Future Generation Computer Systems 101 (2019) 804–818

Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

TSH: Easy-to-be distributed partitioning for large-scale graphs

Ning Wang[a], Zhigang Wang[a,*], Yu Gu[b], Yubin Bao[b], Ge Yu[b]

[a] College of Information Science and Engineering, Ocean University of China, China
[b] School of Computer Science and Engineering, Northeastern University, China

HIGHLIGHTS

- Re-formulate graph partitioning for efficient combination in Pregel-like systems.
- Discover graph locality in many real-world graphs.
- Design a distributed streaming graph partitioning algorithm based on locality.

ARTICLE INFO

Article history:
Received 1 December 2017
Received in revised form 20 August 2018
Accepted 27 June 2019
Available online 22 July 2019

Keywords:
Distributed iterative computation
Large-scale graph
Streaming partitioning
Locality

ABSTRACT

The big graph era is coming with strong and ever-growing demands on parallel iterative analysis. But, before that, balanced graph partitioning is a fundamental problem and is NP-complete. Till now, there have been several streaming heuristic solutions with a single full scan over the input graph. However, some of them cannot be easily parallelized to further accelerate partitioning for large-scale graphs due to complicated heuristics; while others can be run in parallel but incur expensive communication costs during iterative computation.

This paper presents Target-vertex Sensitive Hash (*TSH*), an easy-to-be distributed partitioning method. We first analyze the locality property naturally provided by the original input graph, which has not yet been considered by existing work. We then exploit such locality to simplify the heuristic rule. The simplified rule is implemented by a two-step framework where target vertices of edges are first logically pre-divided without accessing any graph data and then, based on the distribution of target vertices, streaming partitioning is physically performed in parallel. *TSH* provides the capability of quickly dividing large-scale graphs because of parallelization, as well as optimizes communication overheads due to the utilization of locality. Using a broad spectrum of real-world graphs, we conduct extensive performance studies to confirm the effectiveness of *TSH* over up-to-date competitors.

1. Introduction

Graphs have been studied in general applications in the cyber world, like intelligent transportation systems, social networks, bioinformatics, Internet, and scientific computation [1]. Many graph analysis algorithms are naturally iterative so as to refine the final solution step by step, such as PageRank, Shortest Path, and Connected Components. However, the graph data volume is increasingly growing, that poses great challenges to efficient iterative computation. For example, Google has indexed hundreds of billions of webpages.[1] Until November 2017, there are over 2.07 billion monthly active users on Facebook.[2] In order to handle such complex analysis over large graphs, distributed systems have been developed [2–10], all of which expose simple yet expressive APIs for easily programming various algorithms. Users can submit their own graph analysis jobs concurrently. Any job is typically processed by several workers in parallel which run on physical machines in a cloud computing cluster. That maximally unleashes the power of the given cluster.

Graph partitioning is a key component in all of the existing distributed systems. Distributed systems typically partition an input graph onto a group of workers as sub-graphs and then perform iterative update over them. More specifically, at each iteration, a graph algorithm exchanges intermediate results (also called messages) along edges. Then the number of cut edges across sub-graphs can reflect the network communication overhead to some extent and hence the computation efficiency. Also, the computation workload of each worker is supposed to be balanced. Otherwise, the slow, straggling worker can slow down the overall performance. Thus, as studied before, a high-quality partitioning method should decrease the number of cut edges,

* Corresponding author.
E-mail addresses: wangning_neu@163.com (N. Wang), wangzhiganglab@gmail.com (Z. Wang), guyu@mail.neu.edu.cn (Y. Gu), baoyubin@mail.neu.edu.cn (Y. Bao), yuge@mail.neu.edu.cn (G. Yu).

1 https://www.google.com/search/howsearchworks/crawling-indexing/.
2 https://zephoria.com/top-15-valuable-facebook-statistics/.

https://doi.org/10.1016/j.future.2019.06.033
0167-739X/© 2019 Elsevier B.V. All rights reserved.

as well as balance the workload among subgraphs. However, besides the traditional quality factors mentioned above, now we also need to focus on the partitioning efficiency. Based on our experience, the available cloud compute resources usually dynamically change with the submission of graph jobs, and the addition and removal of physical machines. Most of systems, like Pregel [2] developed by Google, quantify available resources by worker slots. They typically provide a *setWorkerNum* API for users to flexibly set the number of workers when submitting their own jobs. The number of workers might not be a constant when running analysis jobs over the same graph but at different times. Therefore, graph partitioning is required by every job. That is, different jobs cannot share partitioning results to amortize partitioning costs. To summarize, balanced workload, optimized communication costs, and high partitioning efficiency, can improve the overall performance of iterative graph analysis, all of which should be considered when partitioning graphs.

However, graph partitioning is NP-complete [11,12]. Existing heuristic solutions basically fall into two categories: offline represented by Metis [13] and streaming/online by LDG [14], Fennel [15] and FG [16]. Offline partitioning can significantly decrease the number of cut edges, and then optimize the communication overhead. It thereby attracts a lot of attention both in industry and in science. However, it requires to frequently traverse vertices, yielding expensive costs especially when processing large-scale graphs. Previous experiments have shown that the partitioning runtime may exceed the iterative computation runtime [15]. Hence, a cost-effective alternative is to accelerate the partitioning procedure by compromising the partitioning quality, i.e. streaming partitioning. It is completed by performing a single full pass over vertices and edges. This can be done along loading input graph. Hence, we also call it as online partitioning through this paper. However, streaming partitioning is usually run on a single machine so as to update heuristic information in a centralized fashion. That limits the scalability. Note that Hash as another streaming solution can be run in parallel but the quality is particularly poor in terms of communication optimization.

Hence, a naturally desirable goal for graph partitioning is to pursue a streaming mechanism that offers high partitioning quality and can be performed in an efficient distributed fashion. This paper explores a path to such a target. In particular, we re-define graph partitioning because most of Pregel-like systems have a built-in combination function to optimize communication overheads. That is, every worker combines messages sent to the same target vertex into a single one before flushing them. Communication overheads indeed are dominated by the number of messages after performing all possible combining operations. In another word, they depend on how many target vertices edges on each worker link to, instead of cut edges across workers. Existing definitions ignore the combining effect. The re-defined partitioning shifts the optimization core from cut edges to target vertices. Previous efforts have shown that the former cannot be optimized without high overheads. However, it is feasible for the latter. We discover that real-world graphs have some locality, that is, many outgoing edges from different source vertices usually link to common target vertices. By leveraging this locality, we design a Target-vertex Sensitive Hash (*TSH*) partitioning algorithm. Specifically, we assume that vertices as target vertices are pre-partitioned into disjoint subsets, each of which is associated with one sub-graph. Pre-partitioning is performed in a simple fashion for efficiency and the result is regarded as the heuristic rule. Accordingly, *TSH* can decide the placement of each vertex arriving in a streaming fashion to optimize the actual number of target vertices in all sub-graphs. But more importantly, it can be performed in parallel for efficiency since the heuristic rule has been fixed. Our experiments show that although the heuristic rule is not dynamically updated as traditional solutions, it still provides comparable communication optimization effect because of the locality property in real-world graphs.

The major contributions are summarized as below.

- We re-formulate graph partitioning for Pregel-like systems by explicitly taking their built-in combining function into account. The new definition emphasizes decreasing the number of target vertices.
- We discover the graph locality and our further investigation reveals that it exists in many real-world graphs.
- Based on the re-formulated partitioning and the locality, we design a novel streaming partitioning algorithm *TSH*. It can be efficiently performed in a distributed fashion, as well as optimize the communication overheads across balanced sub-graphs.
- Extensive experimental studies explore the performance features of our proposal. We demonstrate that *TSH* can offer comparable quality to the state-of-the-art centralized streaming partitioning solution but run faster than it.

The remainder of this paper is organized as follows. Section 2 provides necessary background regarding distributed graph processing systems and then formally gives the definition of graph partitioning. Section 3 introduces the graph locality and uses a lot of preliminary experimental results to show that it is widespread. Section 4 presents the details of our new partitioning algorithm *TSH*. Section 5 reports extensive evaluation results. Section 6 highlights related work. Finally, Section 7 concludes this paper.

2. Problem definition

This section first uses the Pregel system developed by Google to demonstrate how to perform iterative graph computation in parallel. We then re-formulate the goal of graph partitioning in our context.

2.1. Iterative graph computation on Pregel-like systems

As shown in Fig. 1, Pregel runs on a cloud computing cluster with multiple physical machines where one is selected as Master to manage the whole system and others are responsible for underlying computation. Graph processing jobs are concurrently submitted in multi-tenant environments, each of which is divided into K (such as 3 in Fig. 1) workers and then scheduled onto selected machines for parallel computation. Note that one machine can run several workers as a group at the same time, but in this paper, we assume that a group consists of one worker at most to avoid potential resource contention. Workers need to load input graph from file systems such as HDFS [17] and then partition data into K sub-graphs. The subsequent iterations are separated by global barriers. Within one iteration, workers perform local updates and communicate with each other to exchange intermediate results (messages). These computation workloads should be balanced across workers. Otherwise, fast workers will wait for the slow, straggling ones at barriers.

Note that the K values of jobs can be very different from each other, which is decided by many factors like available resources and urgency. Hence, jobs individually perform the partitioning process based on their corresponding K values, even though they take the same graph as input. The partitioning runtime is thereby counted into the overall runtime of a job since it cannot be amortized.

In particular, an edge is a cut edge if its endpoints are placed onto different workers. Messages are transmitted along cut edges in a batch fashion, in order to make full use of the network idle time and reduce the overhead of building communication

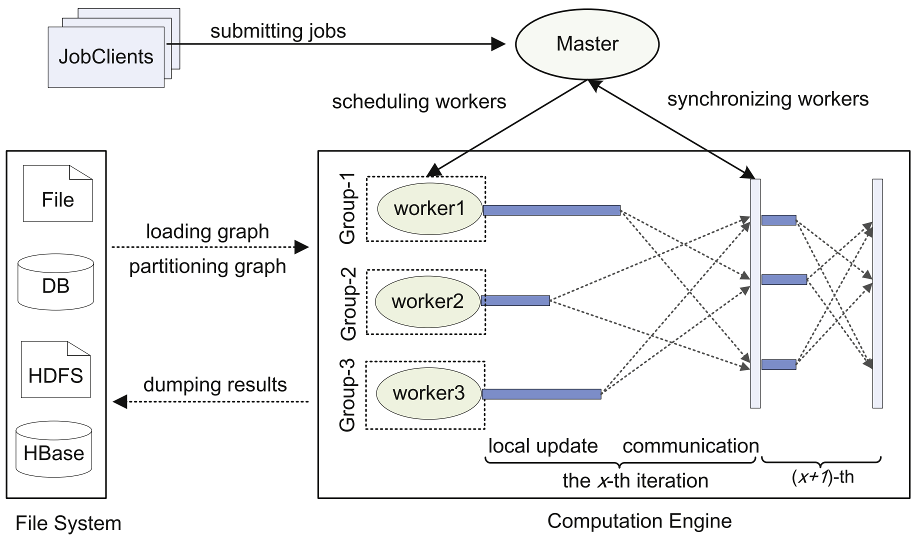


Fig. 1. Parallel graph computation in Pregel.

connections. Pregel immediately flushes messages at the sender side once the number of buffered messages exceeds a pre-defined batch size (also called the sending buffer size). Before that, however, messages sent to the same target vertex can be combined into a single one for efficiency. Clearly, the communication volume is actually dominated by the total number of such target vertices on each worker.

2.2. K-way graph partitioning for Pregel-like systems

We model a graph as a directed graph $G = (V, E)$ where V is a set of vertices and E is a set of edges (pairs of vertices). $|V|$ and $|E|$ denote the number of vertices and edges, respectively. Given a directed edge $(v, u) \in E$, v is the source vertex and u is the target vertex. We conventionally represent G in adjacency lists by grouping edges around sources. This input format is widely used by Pregel-like systems because many graph algorithms send the source vertex value as a message to target vertices along outgoing edges.

Before parallel computation, the Graph Partitioning process in Pregel (*GPP*) should quickly partition G into K sub-graphs $G_i = (V_i, E_i)$, where $E_i = \{(v, u)|v \in V_i\}$ and $\forall\ 1 \le i < j \le K, V_i \cap V_j = \emptyset$. Let $|G_i|$ be the workload associated with G_i. V_i^{tar} is the set of target vertices to which cut edges in E_i link, i.e., $V_i^{tar} = \{u|(v, u) \in E_i \wedge u \notin V_i\}$. As analyzed in Section 2.1, a partition output by *GPP* should balance $|G_i|$ and minimize the summation over $|V_i^{tar}|$ with i ranging from 1 to K. Eq. (1) mathematically gives the goals of *GPP*. Here, C_p stands for the partitioning runtime and ρ is the workload balance factor with ideal value 1.0. In fact, Eq. (1) defines the (K, ρ)-balanced graph partitioning. However, different from traditional definition, now the optimization goal w.r.t. network cost is to reduce $\sum_{i\in[1,K]} |V_i^{tar}|$ rather than the number of cut edges. Theorem 1 tells us that *GPP* is also NP-Complete.

$$\begin{cases} \min C_p \\ \max\limits_{i\in[1,K]} \{|G_i|\} \le \rho \cdot \dfrac{|G|}{K} \\ \min \sum\limits_{i\in[1,K]} |V_i^{tar}| \end{cases} \tag{1}$$

Theorem 1. *For $K \ge 3$, $(K, 1)$-GPP is NP-Complete.*[3]

Proof. Inspired by Ref. [11], we have this claim by a reduction from 3-Partition, a proved NP-Complete problem [12]. Assume that we have a set A of $3K$ elements and a polynomial bound $B \in Z^+$. Further, $\forall a \in A$, a size $s(a)$ satisfies $B/4 < s(a) < B/2$ and $\sum_{a\in A} s(a) = K \cdot S$. The 3-Partition problem aims to answer whether A can be partitioned into K disjoint subsets $A_1, A_2, \dots, A_K$ such that $\sum_{a\in A_i} s(a) = B$ for $1 \le i \le K$.

Before demonstrating the reduction, we first construct a graph G with $3K$ small graphs, each of which corresponds to an element a in A. Notice that one small graph w.r.t. a has $s(a)$ vertices or edges, that depends on how we measure the computation workload in *GPP*. Because a and B are both polynomially bounded, the graph construction can be performed in polynomial time.

Now if we can find a solution for a 3-Partition instance, we can also employ such solution to perfectly solve the $(K, 1)$-balanced *GPP* problem in G with $\sum_{i\in[1,K]} |V_i^{tar}| = 0$, i.e., zero network cost in iterative computation. More specifically, each partitioned sub-graph has exactly 3 small graphs with total B vertices if $|G_i| = |V_i|$ or edges if $|G_i| = |E_i|$, and then no edge is cut between any two sub-graphs. Conversely, if we cannot find such solution, there exists one cut edge at least in the partition of G, i.e., $\sum_{i\in[1,K]} |V_i^{tar}| \ge 1$. Hence, $(K, 1)$-GPP is also NP-Complete when $K \ge 3$. □

In adjacency lists, because all outgoing edges of v like (v, u) share the same source, we store an edge only by the corresponding target like u, to reduce storage space. That is, one list *adj* is given by $\{v, \Gamma_v\}$, where v is the source and Γ_v is the set of targets, i.e. $\Gamma_v = \{u|(v, u) \in E\}$. Then *GPP* should partition every adjacency list $\{v, \Gamma_v\}$ as a whole into K sub-graphs.

Recall that the traditional streaming graph partitioning [14,15] requires one-pass data access, which decreases C_p. However, it is usually performed in a centralized way to accurately maintain heuristic rules and then reduce the number of cut edges for efficient graph computation. The scalability is poor when partitioning very large graphs. On the other hand, the distributed method with good scalability, like Hash, abandons heuristic rules and hence generates a lot of communication costs in graph computation. Neither of them can well solve our *GPP* problem. However, we find that the goal of optimizing $\sum_{i\in[1,K]} |V_i^{tar}|$ and the locality in real-world graphs (see Section 3) enable us to simplify heuristic rules (see Section 4) so that they can be performed in parallel.

[3] For $K = 2$, $(2, 1)$-partitioning is equivalent to the Minimum Bisection problem. It is still NP-Complete with our network cost optimization by referring to Ref. [12]. We omit the proof since for Pregel-like systems, K is much greater than 2 in most cases.

3. Locality of real-world graphs

The complex relationship among real-world entities can be generally modeled by graphs with some locality. This section analyzes the locality and then demonstrates it over many real-world graphs. The locality is the foundation of our new partitioning solution introduced in Section 4.

3.1. Theoretical analysis

Intuitively, entities as vertices have similar behaviors, i.e., they have many edges linking to the common target vertices. Taking social networks as an example, if A and B are friends, then some of their friends overlap, i.e., they have a lot of common neighbors. Grouping A and B into one sub-graph of course increases the probability of combining outgoing messages sent to their neighbors, that decreases the communication overhead.

Given any two vertices $v, u \in V$, their similarity naturally can be measured by the size of the intersection of Γ_v and Γ_u. In particular, we use the well-known Jaccard coefficient $J(\Gamma_v, \Gamma_u)$ as the metric which normalizes the intersection size by the size of the corresponding union, as shown in Eq. (2). We are aware that another metric, triangle [18], can also capture the intersection of neighbors. However, a triangle requires v is adjacent to u, that is over-strict for combination.

$$J(\Gamma_v, \Gamma_u) = \frac{|\Gamma_v \bigcap \Gamma_u|}{|\Gamma_v \bigcup \Gamma_u|} \tag{2}$$

Fig. 2 gives an example graph with 7 vertices and 11 edges. Here, $\Gamma_1 = \{2, 3, 4\}$ and $\Gamma_5 = \{2, 3, 4, 6, 7\}$. They share the target vertices $\{2, 3, 4\}$ and hence $J(\Gamma_1, \Gamma_5) = 0.6$. On the other hand, the intersection of Γ_3 and Γ_4 is empty and then $J(\Gamma_3, \Gamma_4) = 0$.

Locality: The *locality* of a graph G, L_G, is given based on the Jaccard similarity. $\forall v \in V$, we first compute its vertex similarity in G by averaging $J(\Gamma_v, \Gamma_u)$ for all $u \in V/\{v\}$. The vertex similarity shows how likely messages from v can be combined with messages from other vertices. Then L_G is the $|V|$-normalized summation over all vertex similarities, as shown in Eq. (3). Clearly, a big L_G value indicates that a lot of target vertices are shared by edges, that is, many messages can be combined in Pregel if data are placed in a reasonable way. We now analyze the computational complexity of L_G. First of all, because of the symmetry, $J(\Gamma_v, \Gamma_u) = J(\Gamma_u, \Gamma_v)$ and they are respectively involved in computing the vertex similarity of v and u. This motivates us to employ an array of size $\mathbf{n} = |\mathbf{V}|$ to accumulate $J(\Gamma_u, \Gamma_v)$ for u if v's vertex similarity is firstly computed, and vice versa. Accordingly, Eq. (3) only needs to call the subroutine of Eq. (2) $\frac{n(n-1)}{2}$ times. More specifically, a simple yet efficient implementation of Eq. (2) is to sort Γ_v and Γ_u using merge sort, and then compute their intersection and union by traversing elements in the two sorted sets. That leads to a computational complexity of $O(n \log n)$. Together, the final computational complexity of L_G is $O(n^3 \log n)$. It is so high that we cannot validate the locality for very large graphs.

We should stress that even though v and u are not stored adjacently, we still need to compute $J(\Gamma_v, \Gamma_u)$. This is because messages in Pregel-like systems are usually sent in a batch fashion. The system sets a sending threshold. All messages sent by a worker will firstly be cached in the local sending buffer. When the number of buffered messages exceeds the threshold, these messages are immediately flushed out. By this way, the system can make full use of the network idle time and reduce the overhead of building connections. Assume that v and u are far from each other in the local storage. Theoretically, they are updated with poor time locality since vertices are scanned in the storage order. However, messages from v possibly can be combined with those from u due to the existence of sending buffer.

$$L_G = \frac{1}{|V|} \sum_{v \in V} \Big(\frac{1}{|V| - 1} \sum_{u \in V/\{v\}} J(\Gamma_v, \Gamma_u) \Big) \tag{3}$$

Clustered-locality: Our further analysis finds that vertices as sources in adjacency lists may naturally have been clustered in some way and then stored adjacently in the original input file. Together with locality based on the vertex-similarity, we call this phenomenon as *clustered-locality*. It can be quickly measured and can also be used to approximately estimate the locality metric L_G. More importantly, we can utilize it to enhance the performance of our partitioning method.

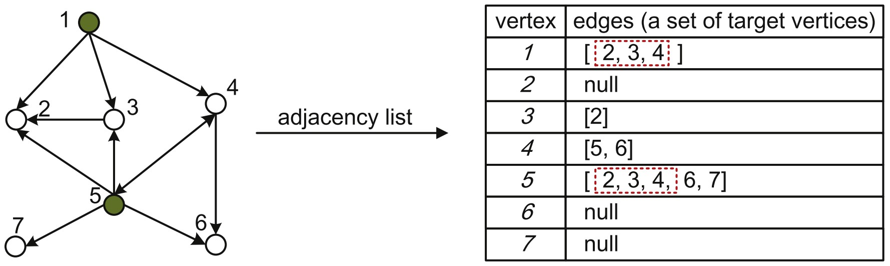
We now enumerate some reasons that make vertices clustered in real graphs. For example, to our knowledge, graphs are usually constructed by traversing real-world entities in a Bread-First Search (BFS) fashion [19]. Given a currently traversed vertex v as source, BFS will traverse target vertices newly found in v's outgoing edges. Such target vertices are regarded as new sources from which new BFS procedures start. The target vertices newly found in one BFS traverse are then stored adjacently (clustered). As another example, vertices in a connected component can also be clustered when graph data are collected.

For a graph with good locality, edges of many vertices as sources will link to these adjacent vertices as targets. As a concrete example, the graph shown in Fig. 2 is BFS-generated. Starting with the source vertex "1", new target vertices "2", "3" and "4" are detected. They are then traversed immediately and stored adjacently. Later, when vertex "5" is traversed, we can observe that some of its edges also link to adjacent vertices "2", "3", and "4". This property motives us to measure clustered-locality by evaluating the adjacent degree of target vertices in every adjacency list, i.e., approximate locality $\tilde{L}_G$. Clearly, if most target vertices in every list are adjacent, the similarity between any two vertices is potentially increased, i.e., L_G is large. Thus, $\tilde{L}_G$ can be used to approximately estimate L_G.

We next focus on quickly computing $\tilde{L}_G$. Assume that vertex ids are numbered consecutively in the natural storage order. The difference between two vertex ids is then inversely proportional to the adjacent degree. Motivated by this, given a vertex v and the set of its k target vertices Γ_v ($k = |\Gamma_v|$), we can measure the adjacent degree $f(v)$ in the following way: (1) sorting entries in Γ_v in ascending order of ids to get the sequence $sort(\Gamma_v) = (u_1, u_2, \ldots, u_k)$; (2) summing up the id difference between u_i and u_{i+1}, i.e., $Id(u_{i+1}) - Id(u_i)$, through a loop from $i = 1$ to $i = (k-1)$; (3) normalizing the summation by $(k-1)$. Note that $f(v) = 0$ when $k \leq 1$. Eq. (4) mathematically shows how to compute $f(v)$. Accordingly, we compute $\tilde{L}_G$ by firstly summing up adjacent degrees of all vertices and then normalizing the summation by $|V|$, as shown in Eq. (5). Let d stand for the maximum out-degree for vertices in V and $m = |E|$. When using sort_merge to compute $sort(\Gamma_v)$, the computational complexity of $\tilde{L}_G$ is $O(m \log d)$. Because $m \ll n^2$ and $d \leq n$ ($n = |V|$), $O(m \log d)$ is smaller than $O(n^3 \log n)$ w.r.t. L_G.

$$f(v) = \begin{cases} \frac{k-1}{\sum_{i=1}^{k-1} \left(Id(u_{i+1}) - Id(u_i) \right)}, & k > 1 \text{ and } u_i \in sort(\Gamma_v) \\ 0, & k \leq 1 \end{cases} \tag{4}$$

$$\tilde{L}_G = \frac{1}{|V|} \sum_{v \in V} f(v) \tag{5}$$



vertex	edges (a set of target vertices)
1	[2, 3, 4]
2	null
3	[2]
4	[5, 6]
5	[2, 3, 4, 6, 7]
6	null
7	null

Fig. 2. Similarity between two vertices.

3.2. Evaluation

We validate that *locality* and *clustered-locality* indeed exist by showing the ratio of L_G and $\tilde{L}_G$ over a real graph to $L_G(rd)$ and $\tilde{L}_G(rd)$ over a corresponding random graph (termed "rd"), respectively.

As listed in Table 1, real graphs used in this paper are collected from a broad class of applications. Specifically, WikiVote[4] contains the Wikipedia voting data where vertices represent users and an edge from vertex v to vertex u represents that user v elects user u as one of administrators of the free encyclopedia; Email-EU[5] is a network generated using email data from an European research institution where one vertex corresponds to an email address and an edge between v and u is created if v has sent at least one message to u; Slashdot,[6] Epinions,[7] Livej,[8] and Orkut[9] are social networks; while, Berkeley[10] and Wikipage[11] are web crawls; Amazon[12] is a product co-purchasing network where vertices represent products and there exists an edge between two products if they are frequently co-purchased. On the other hand, a random graph has the same number of vertices and edges with the corresponding real graph. We generate a random graph without locality by adding edges using the following procedure: for a vertex with identifier v, $1 \leq v \leq |V|$, we add $d_{avg} = \frac{|E|}{|V|}$ edges connecting it to vertices chosen uniform randomly from the interval $[1, |V|]$. Note that in the original input file of every graph, one adjacency list is a line and the source vertices of all lists have already been numbered consecutively in their storage order. We do not make any change such as re-ordering lists based on BFS. When computing L_G, $\tilde{L}_G$, $L_G(rd)$ and $\tilde{L}_G(rd)$, adjacency lists are streamed in the natural storage order originally provided in the input file.

Table 1 summarizes graph features and metric values. $\frac{\tilde{L}_G}{\tilde{L}_G(rd)} > 1.0$ indicates that real graphs outperform random graphs in terms of *clustered-locality*. On the other hand, for L_G, we do not test it for large graphs Livej, Wikipage and Orkut, because of the high computational complexity. However, we generally observe that $\frac{L_G}{L_G(rd)} > \frac{\tilde{L}_G}{\tilde{L}_G(rd)}$ over every small graph except Amazon. Together with $\frac{\tilde{L}_G}{\tilde{L}_G(rd)} > 1.0$, we can reasonably infer that $\frac{L_G}{L_G(rd)}$ is consistently greater than 1.0. That is, real graphs have better *locality* than random graphs. The validation of *locality* and *clustered-locality* is a key basis to design our new partitioning method in Section 4.

4 http://snap.stanford.edu/data/wiki-Vote.html.
5 http://snap.stanford.edu/data/email-EuAll.html.
6 http://snap.stanford.edu/data/soc-Slashdot0811.html.
7 http://snap.stanford.edu/data/soc-Epinions1.html.
8 http://snap.stanford.edu/data/soc-LiveJournal1.html.
9 http://law.di.unimi.it/webdata/orkut-2007/.
10 http://snap.stanford.edu/data/web-BerkStan.html.
11 http://haselgrove.id.au/wikipedia.htm.
12 http://snap.stanford.edu/data/amazon0302.html.

4. Target-vertex sensitive hash partitioning

This paper proposes a Target-vertex Sensitive Hash (*TSH*) to parallelize streaming partitioning by leveraging the graph locality. We now formally give the heuristic rule, followed by its distributed implementation.

4.1. Heuristic rule

The biggest difference between existing streaming solutions [14–16] like *LDG* and our *TSH* is that the former try to reduce the number of cut edges to improve the communication efficiency, while the latter achieves this goal by increasing the probability of combining messages. Specifically, *LDG*, for example, uses the distribution of source vertices in already assigned adjacency lists as heuristic information. It assigns a newly streamed adjacency list $\{v, \Gamma_v\}$ to the sub-graph that has the most common sources with out-neighbors as targets in Γ_v. Clearly, the distribution table in *LDG* changes when new adjacency lists as well as the corresponding source vertices are assigned. It is really expensive to synchronize the change across distributed workers. On the other hand, *TSH* improves the combination hit ratio by grouping adjacency lists with the most common out-neighbors into the same sub-graph. Then Pregel can combine messages sent to the same target vertices as much as possible. As shown in Table 1, real graphs usually have good locality—many lists share a lot of common out-neighbors. Thus, we can logically assume a distribution of target vertices as heuristic to guide data placement. That is, a new adjacency list is assigned to the sub-graph with the most common out-neighbors/targets in the corresponding logical distribution table.

Motivated by this, we design a two-step partitioning framework for *TSH* where in the first step an expected distribution of target vertices is logically computed and in the second step adjacency lists are physically loaded and grouped.

We call the first step as pre-partitioning since it happens before the input graph is physically partitioned. Its output is K disjoint subsets of V, i.e., $\{T_1, T_2, \ldots, T_i, \ldots, T_K\}$, where $\bigcup_{i\in[1,K]} T_i = V$. T_i corresponds to the sub-graph G_i, indicating that we expect adjacency lists with (almost) the same target vertices in T_i to be assigned into G_i. Then messages sent to these target vertices can be effectively and efficiently combined. Later we will give two pre-partitioning implementations.

TSH feeds a series of adjacency lists and the pre-partitioning result into the second step. One list *adj* consisting of a source v and its target set Γ_v is a line in the original input file, as shown in Fig. 2. Lists are streamed along loading data in Pregel and then assigned to sub-graphs/workers. A list is never moved after it has been assigned. Hence, *TSH* is a streaming partitioning method. When *adj* arrives at the time instance t, we naturally hash it into G_i with T_i sharing a maximum number of targets with Γ_v. Eq. (6) gives a mathematical description. Here, *gid* is the id of the selected sub-graph. $w(t, i)$ is the real-time remaining workload capacity of sub-graph G_i at time t. It works as a penalty function

Table 1
Locality of real-world graphs ($K = \times 10^3$, $M = \times 10^6$).

Graphs	$\|V\|$	$\|E\|$	$\tilde{L}_G$	$\tilde{L}_G(rd)$	$\frac{\tilde{L}_G}{\tilde{L}_G(rd)}$	L_G	$L_G(rd)$	$\frac{L_G}{L_G(rd)}$
WikiVote	7.1K	103K	4.6×10^{-3}	2.5×10^{-3}	1.9	1.7×10^{-3}	2.5×10^{-4}	6.7
Slashdot	77K	905K	4.8×10^{-4}	3.6×10^{-4}	1.4	3.8×10^{-4}	3.0×10^{-5}	12.5
Email-Eu	265K	420K	1.0×10^{-2}	2.2×10^{-5}	466.5	3.8×10^{-3}	3.9×10^{-6}	964.8
Epinions	76K	509K	1.3×10^{-4}	9.4×10^{-5}	1.4	3.2×10^{-4}	1.9×10^{-5}	16.3
Amazon	262K	1.2M	2.1×10^{-2}	7.2×10^{-5}	287.1	2.4×10^{-5}	8.2×10^{-6}	2.9
Berkeley	685K	7.6M	6.1×10^{-2}	4.3×10^{-5}	1412.0	6.8×10^{-3}	4.5×10^{-6}	1507.0
Livej	4.8M	69M	8.2×10^{-3}	5.1×10^{-6}	1607.8			
Wikipage	5.7M	130M	2.9×10^{-3}	5.0×10^{-6}	580.0			
Orkut	3.1M	223M	3.4×10^{-4}	3.2×10^{-5}	10.6			

to balance workloads among sub-graphs. Note that in Pregel, the majority of iterative computation workload is to generate messages and then send them along edges. We thereby use the number of edges $|E_i|$ in a sub-graph G_i to stand for $|G_i|$ and $|E_i| = \sum_{v \in V_i} |\Gamma_v|$. Then $w(t,i)$ is computed by Eq. (7), where $|E_i^t|$ is the real-time workload of G_i at t and $C = \frac{|E|}{K}$.

$$gid = \arg\max_{i \in [1,K]} \{|\Gamma_v \cap T_i| \cdot w(t,i)\} \tag{6}$$

$$w(t,i) = 1 - \frac{|E_i^t|}{C} \tag{7}$$

In a sense, $\{T_1, T_2, \dots, T_K\}$ is the key to having the heuristic rule perform well because it directly decides the size of intersection $T_i \cap \Gamma_v$—the combination gain. On the other hand, we should quickly build $\{T_1, T_2, \dots, T_K\}$ as the time spent on pre-partitioning is a fraction of C_p. One possible solution is Hash, i.e. "id%K", but target vertices often shared by source vertices cannot be effectively clustered. Take the graph in Fig. 2 as an example. Assume that $K = 3$. Then we have $T_1 = \{3, 6\}$, $T_2 = \{1, 4, 7\}$, and $T_3 = \{2, 5\}$. Targets $\{2, 3, 4\}$ shared by sources "1" and "5" are divided into three different subsets. Ignoring the affect of $w(t,i)$, list $adj_5 = \{5, \Gamma_5 = \{2, 3, 4, 6, 7\}\}$ will be randomly put into G_1 or G_2 since $|\Gamma_5 \cap T_1| = |\Gamma_5 \cap T_2| = 2$. Similarly, $adj_1 = \{1, \Gamma_1 = \{2, 3, 4\}\}$ is randomly assigned among all three sub-graphs because the intersection size is always 1. The two lists are grouped into the same sub-graph with low probability ($\frac{1}{3}$) even though their targets are highly similar to each other. Another preferred solution is to evenly and continuously partition vertices, termed Range. It preserves the *clustered-locality* of graphs but requires that vertices are numbered consecutively in the storage order. In Fig. 2, the pre-partitioning result under *Range* is $T_1 = \{1, 2, 3\}$, $T_2 = \{4, 5, 6\}$, and $T_3 = \{7\}$. adj_1 belongs to G_1 and adj_5 belongs to G_1 or G_2. The two lists can be placed into G_1 in high probability ($\frac{1}{2}$).

Unlike the dynamically changed distribution of sources in *LDG*, the logical distribution of targets in *TSH* is fixed during partitioning. This simplifies the parallel implementation because every worker can get a copy of the table before loading data and then partition data in local stream without maintenance cost on the table. We should also stress that although *TSH* keeps the logical distribution table unchanged, it optimizes the communication cost by explicitly utilizing graph locality, i.e., the intersection computation in Eq. (6). For example, *Worker*$_1$ loads an adjacency list $\{x, \Gamma_x\}$ and then sends it to sub-graph G_2 if $(|\Gamma_x \cap T_i| \cdot w(t,i))$ gets the largest value at $i = 2$, for $1 \le i \le K$. Here, T_i indicates the assumed set of targets in sub-graph G_i, instead of source vertices. T_i does not change during partitioning. And after partitioning, we expect that adjacency lists with most common neighbors in T_i can be grouped into G_i. Assume that there is no locality, that is, targets of any source v are chosen uniform randomly from the interval $[1, |V|]$, resembling the generation of random graphs mentioned in Section 3.2. Then *TSH* cannot effectively group adjacency lists no matter how to pre-partition target vertices. We verify this in Section 5.5. Further, neither *TSH-H* nor *TSH-R* makes assumptions on the streaming order of lists. Otherwise, users need to re-sort lists before running our methods. That increases the pre-processing cost and hence C_p, violating the condition of minimizing C_p shown in Eq. (1).

4.2. Distributed implementation

We now implement *TSH* in Pregel-like systems. *TSH* consists of two procedures: (1) pre-partitioning target vertices into $\{T_1, T_2, \dots, T_K\}$ through *Hash* or *Range*, shown in Algorithm 1, and (2) partitioning the input graph by Eq. (6), shown in Algorithm 2. The former as logical partitioning does not load any input data. We can run it in a centralized way because of the low runtime cost. While, the latter needs to physically load every adjacency list and then call Equation (6) to decide its placement. This motivates us to parallelize the execution for efficiency. Pregel's architecture can facilitate the implementation of *TSH* where Master is responsible for building $\{T_1, T_2, \dots, T_K\}$ and workers concurrently load specified input data for partitioning.

Algorithm 1: Logical pre-partitioning on Master

```
Input : number of sub-graphs $K$, metadata of the input
        graph $D = \{|V|, |E|\}$
Output: pre-partitioning result $\{T_1, T_2, \dots, T_K\}$ and average
        workload $C$
1 pre-partitioning target vertices into $\{T_1, T_2, \dots, T_K\}$ based
  on $D$
2 computing the average workload $C = \frac{D.|E|}{K}$
3 broadcasting $\{T_1, T_2, \dots, T_K\}$ and $C$ to Workers
4 notifying Workers to start physical partitioning
```

4.2.1. Logical pre-partitioning

Algorithm 1 demonstrates how to logically pre-partition data on Master. The goal of this procedure is outputting key information required by Eq. (6), including the distribution of target vertices and the average workload C among workers. The former is used as heuristic information to guide the placement of a newly loaded adjacency list. We do not need to physically load target vertices and then divide them. Instead, we achieve the distribution of target vertices through logical computation (Line 1). Specifically, when *Range* is employed, vertices as targets will be evenly and consecutively divided into K subsets $\{T_1, T_2, \dots, T_K\}$. We achieve this goal by specifying the id range of target vertices in T_i as $[(i-1)\frac{|V|}{K} + \alpha, i\frac{|V|}{K} + \alpha]$, where $1 \le i \le K$ and α is the minimal vertex id in G. The case with *Hash* is even more simple because Master does not perform any computation. Instead, given a target vertex id $u.id$, we can quickly specify to which subset it is divided by $u.id\%K$, whenever necessary. On the other hand, the computation of C is given by $\frac{|E|}{K}$ (Line 2). Then Master can notify workers to physically load input data and partition them after broadcasting the key heuristic information (Lines 3–4). Notably, T_i is a pair of figures $((i-1)\frac{|V|}{K}+\alpha$ and $i\frac{|V|}{K}+\alpha)$ for *Range* or semantic information "id%K" for *Hash*. As analyzed above, Master will not be the performance bottleneck in terms of computation and communication.

Algorithm 2: Physical partitioning on Worker$_i$

```
Input  : metadata $D_i$, pre-partitioning result
         $\{T_1, T_2, \ldots, T_K\}$, average workload $C$, number of
         workers $K$
Output: sub-graph $G_i$
1  $w(t,x) \leftarrow 1, \forall x \in [1,K]$
2  Sender:
3  while there exist adjacency lists specified by $D_i$ do
4      loading a new adjacency list adj={$v$, $\Gamma_v$}
5      $\hat{s} \leftarrow 0$
6      $c \leftarrow \varnothing$
7      foreach $x = 1$ to $K$ do
8          $s_x \leftarrow |\Gamma_v \cap T_x| \cdot w(t,x)$
9          if $\hat{s} < s_x$ then
10             $\hat{s} = s_x$
11             $c = \{x\}$
12         else if $\hat{s} = s_x$ then
13             $c = c \cup \{x\}$
14     gid $\leftarrow$ randomSelect($c$)
15     sending adj to Worker$_{gid}$
16 broadcasting EOF to all workers
17 Receiver:
18 $\Delta \leftarrow 0$ and local_cnt $\leftarrow 0$
19 while #EOF$< K$ do
20     if an adj is received then
21         putting adj into $G_i$
22         update($i, \frac{adj.|\Gamma_v|}{C}$)
23         $\Delta \leftarrow \Delta + \frac{adj.|\Gamma_v|}{C}$
24         if local_cnt++ $\geq \eta$ then
25             invoking Worker$_x$.update($x$,$\Delta$) for
               $x \in [1,K] \wedge x \neq i$
26             $\Delta \leftarrow 0$ and local_cnt $\leftarrow 0$
27 Function void update(int x, int dec):
28     $w(t,x) \leftarrow w(t,x) - dec$
```

4.2.2. Physical partitioning

Algorithm 2 gives the implementation of physical partitioning on *Worker$_i$*. Note that on each worker, there exist two computation threads working as *Sender* and *Receiver*. *Sender* loads partial input graph specified by the metadata D_i and then sends every loaded adjacency list to the target worker/sub-graph selected by Eq. (6). Meanwhile, *Receiver* continuously receives the adjacency list from any possible *Worker* and then puts it into the local sub-graph G_i.

Recall that given an adjacency list *adj*, Eq. (6) needs to compute assignment scores of all workers to select the optimal target worker/sub-graph. This requires *Worker$_i$* to maintain all penalty functions where $w(t,i)$ as the local one is managed by *Worker$_i$* and others are replicas of functions managed by remote workers. At the very beginning, no graph data is assigned. Thus, $\forall x \in [1,K]$, $|E_x| = 0$. Based on Eq. (7), *Worker$_i$* initializes all penalty function values as 1 (Line 1). Afterwards, the *Sender* and *Receiver* threads are launched.

Sender: Once *Sender* loads a new list *adj*, it first resets the maximal assignment score $\hat{s}$ and the candidate set of target workers c (Lines 5–6). The two variables are then re-computed for *adj* (Lines 7–13). Specifically, following Equation (6), for each candidate worker *Worker$_x$*, its assignment score s_x is given by multiplying $|\Gamma_v \cap T_x|$ by $w(t,x)$. The former indicates how many target vertices are commonly shared by *adj*.Γ_v and T_x, aiming at increasing the probability of combining messages and hence minimizing the communication cost. While, the latter as a penalty function can balance the workload among workers. For example, if *Worker$_x$* is overloaded at time t, its $w(t,x)$ will decrease based on Eq. (7). The resulting score s_x gets small to reduce the probability of assigning *adj* to *Worker$_x$*. We generally select the worker with the highest score as the target to trike a good balance between communication optimization and workload balance (Lines 9–11). If several workers have the same score s_x and $s_x = \hat{s}$, we randomly select one from these candidates (Lines 12–14). Finally, a worker will broadcast an EOF flag to notify other workers that it has already processed all of local adjacency lists.

More specifically, computing the intersection size $|\Gamma_v \cap T_x|$ depends on the pre-partitioning policy. When *Range* is used, we compute it by judging whether each target vertex in *adj*.Γ_v falls into the id range of T_x. The size increases by one if yes. If *Hash* is employed, we make the judgment by the hashing result of a target vertex id. Besides, the penalty function value $w(t,x)$ is read-only in *Sender* and is used when computing the assignment score s_x (Line 8). The update on $w(t,x)$ is performed by *Receivers* of the local or remote workers. We will show the update policy in the following introduction to *Receiver*.

Receiver: Because *Senders* have already considered the communication cost and workload balance before sending adjacency lists, the only task of *Receiver* is to passively receive data until all *Senders* finish the loading operation, i.e., the number of EOFs received is equal to K (Line 19). A newly received *adj* will be put into the local sub-graph G_i, which immediately decreases the remaining capacity $w(t,i)$ by *adj*.$|\Gamma_v|/C$ since we use the number of edges as the workload metric (Lines 20–22).

Notably, there exist $(K-1)$ replicas of $w(t,i)$ on other workers. Theoretically, after updating $w(t,i)$, *Worker$_i$* should immediately synchronize these replicas so that other workers can utilize the up-to-date $w(t,i)$ when deciding the placement of their locally loaded adjacency lists. However, frequently performing synchronization requires a lot of network resources. A compromise solution is to synchronizes replicas of its $w(t,i)$ in a batch fashion (Lines 23–26). That is, *Worker$_i$* first locally accumulates changes w.r.t. $w(t,i)$ into Δ, and then updates the replica of $w(t,i)$ on *Worker$_x$* by *Worker$_x$.update()* if the batch size *local_cnt*, i.e., the number of already received lists, exceeds a user-specified threshold η. We will experimentally give a proper threshold in Section 5.3.

5. Performance studies

Now we study the performance of our proposals by comparing them with state-of-the-art partitioning techniques.

Compared Solutions: The main competitors include four well-known streaming methods: *Hash*, *LDG* [14], *Fennel* [15], and *FG* (Fractional Greedy) [16]. *Hash* is a fast distributed heuristic where workers load adjacency lists in parallel and then compute the target sub-graphs by taking the source vertex id of a list modulo K. By contrast, *LDG*, *Fennel* and *FG* are all centralized but employ different polices to place streamed adjacency lists and balance the workload. For *TSH*, we have two variants: *TSH-H* with *Hash* pre-partitioning and *TSH-R* with *Range* pre-partitioning. Besides, some important non-streaming solutions with input graph accessed multiple times are also tested, like the BFS-based Greedy Graph Growing technique (*GGG*) used in METIS [20], the partial re-streaming variant of *FG* [16] (*PartFG*), and *XtraPuLP* that supports multiple objective metrics and multiple constraints. Among them, *GGG* and *PartFG* are centralized while *XtraPuLP* is distributed. In particular, *PartFG* re-streams a portion of the input graph multiple times and the rest of graph is streamed only once. Compared with re-streaming the whole graph [21], *PartFG* can minimize the runtime while providing acceptable partitioning quality. *XtraPuLP* is based on iterative label propagation technique.

We implement *Hash*, *LDG*, *Fennel*, *FG*, *GGG* and *PartFG* by own in Java, and directly use the open-source *XtraPuLP* written in C++ on GitHub (https://github.com/HPCGraphAnalysis/PuLP). When analyzing the partitioning runtime, we parallelize the centralized *LDG*, *Fennel*, *FG*, *GGG* and *PartFG* to perform a fair comparison with distributed *Hash*, *XtraPuLP* and our *TSH*. The parallel variants are called *pLDG*, *pFennel*, *pFG*, *pGGG* and *pPartFG*, respectively.

We next briefly introduce how to parallelize centralized solutions. For *LDG*, *Fennel*, *FG* and *PartFG*, they decide the placement of an adjacency list based on the distribution of lists already loaded. In distributed environments, adjacency lists are loaded in parallel but their placements must be broadcasted periodically so that the subsequent lists can be correctly placed. Clearly, a short broadcasting interval enables each sub-graph to be aware of the most recent change on the distribution of vertices, but incurs expensive communication costs. We experimentally select a reasonable interval such that the broadcasting cost is minimized while the parallel implementations can achieve similar partitioning quality to the corresponding centralized versions. Further, like *TSH*, *LDG*, *Fennel*, *FG* and *PartFG* also penalize the sub-graphs with large size for workload balance. In their parallel variants, the penalty function value is automatically synchronized along broadcasting placements of lists. For *GGG*, the parallel implementation is simply to run BFS in parallel.

General Experiment Setting: All partitioning methods are tested over four graphs listed in Table 1: Berkeley, Livej, Wikipage, and Orkut. We integrate all partitioning methods except *XtraPuLP* into BC-BSP [8], an open-source Java-based clone of Pregel. Assume that we launch K workers in BC-BSP for parallel computation. For centralized partitioning methods, only one worker works in the partitioning phase to get K sub-graphs. By contrast, parallel variants can fully utilize the total K workers. After partitioning, we run a real application PageRank with 30 iterations on BC-BSP to show the impact of partitioning on performance. For *XtraPuLP*, we first run it to get a partition and then feed the partition into BC-BSP for iterative computation. In BC-BSP, the sending buffer size on each worker is measured by the number of buffered messages. Our testing cluster has 21 machines where one of them works as Master. All machines are connected by Gigabit Ethernet to a switch. Each has 2 Intel Core i3-2100 CPUs, 8GB RAM, and a 500GB disk with 7,200 RMP. Notice that each machine except Master runs a single worker to avoid resource contention. Unless otherwise specified, the input graph is always divided into $K = 20$ sub-graphs which are evenly placed on the total 20 workers. Finally, for streaming methods, adjacency lists are streamed in the natural storage order in the input graph file.

Evaluation Metrics: This paper cares about both partitioning quality and efficiency. Specifically, we evaluate partitioning quality from two perspectives: the communication cost in iterative computation and the load balance factor across sub-graphs. Let $|M|$ be the number of messages actually transmitted via network when computing PageRank. We use the ratio of $|M|$ to $|E|$ to measure the communication cost (*com-ratio*). On the other hand, Eq. (1) tells us that the load balance factor ρ is given by $\rho = \max\{|G_i|\}/\frac{|G|}{K}$, $i \in [1, K]$. $|G_i|$ can be the number of vertices $|V_i|$ or edges $|E_i|$. For efficiency, we analyze the partitioning runtime C_p, iterative PageRank computation runtime C_c and their sum—overall runtime C. Clearly, for all metrics except ρ, lower values are preferred. For ρ, it is expected to be close to 1.0.

Experiment Organization: In the following, we first analyze the performance of streaming partitioning methods in terms of partitioning quality (Section 5.1) and efficiency (Section 5.2), and then explore the impact of different batch sizes on *TSH* (Section 5.3). We next test the scalability in different K settings (Section 5.4). We also demonstrate the importance of locality by analyzing the partitioning quality over a random graph (Section 5.5). We then compare *TSH* with complex non-streaming partitioning techniques (Section 5.6). Finally, we verify that *TSH* can quickly output a partition with good locality (Section 5.7).

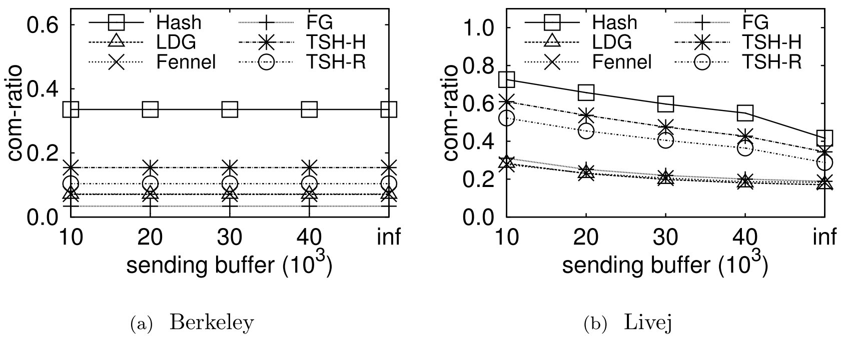
5.1. Partitioning quality

In this group of experiments, we analyze partitioning quality of streaming methods. Note that although *TSH* focuses on edge-based balance, we still report both vertex- and edge-based ρ values for a complete comparison. The batch size threshold η used in *TSH-H* and *TSH-R* is set as $\eta = 10^3$ to reduce the synchronization cost. We will give a detailed discussion about η in Section 5.3.

Fig. 3 reports the communication ratios over different graphs. Because $|M|$ relies on how many messages can be buffered at the sender side, we vary the buffer size from 10^4 to *inf* in experiments, where *inf* indicates that the buffer is big enough to store all messages from a sub-graph. In this case, all messages generated by one sub-graph but sent to the same target can be combined into a single one, yielding the best combination effect.

LDG, *Fennel* and *FG* consistently perform best because of the centralized update for heuristic rules. In particular, the three solutions have the similar behaviors because the combination function in Pregel narrows their performance gap in terms of reducing the number of cut edges. *TSH-H* works better than *Hash* as the latter completely ignores the distribution of already placed adjacency lists. By preserving *clustered-locality*, *TSH-R* further outperforms *TSH-H*. In particular, compared with distributed *Hash* and *TSH-H*, *TSH-R* respectively reduces the communication ratio by 69% and 38% at most (from 28% and 14%). On the other hand, it offers a comparable performance to the best competitors, i.e., the centralized *LDG*, *Fennel* and *FG*. The only drawback of *TSH-R* is that it requires to consecutively number vertex ids in the storage order. Further, we observe that the performance gap among all solutions is narrowed with the increase of the sending buffer size. This is because a large buffer size naturally increases the probability of combining messages, and hence weakens the effect of partitioning. The similar trend can also be observed when K increases as shown in Fig. 6(a) in the scalability test (Section 5.4). That explains why *Fennel*, *LDG* and *FG* perform similarly when $K = 20$ in terms of com-ratios in Fig. 3.

Fig. 4 demonstrates the edge- and vertex-based load balance factors. *Hash* is better than others for both factors because of the random data placement policy. Benefiting from the *Hash*-based pre-partitioning, *TSH-H* has similar performance to *Hash*. Differently, *LDG*, *Fennel*, *FG* and *TSH-R* focus on balancing the edge-based workload. They then perform poorly from the perspective of the vertex-based balance factor. Recall that in Pregel, the computation workload of many graph algorithms like PageRank is mainly dominated by the number of edges. Hence, as we will show later, *LDG*, *Fennel*, *FG* and *TSH-R* can still improve the computation efficiency, even though the distribution of vertices is skew.



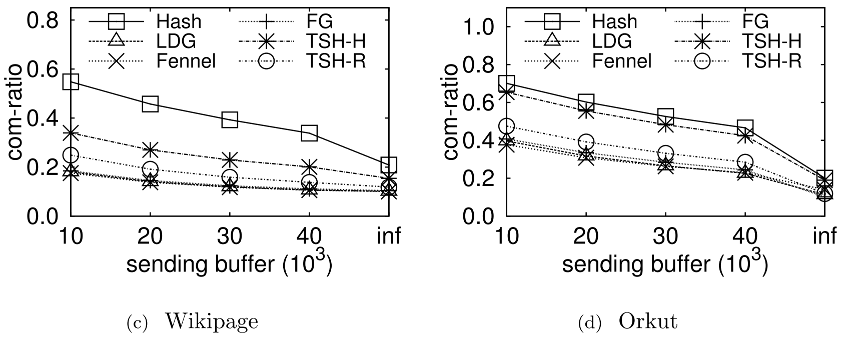


Fig. 3. The communication ratios (com-ratios) over different graphs ($K = 20$).

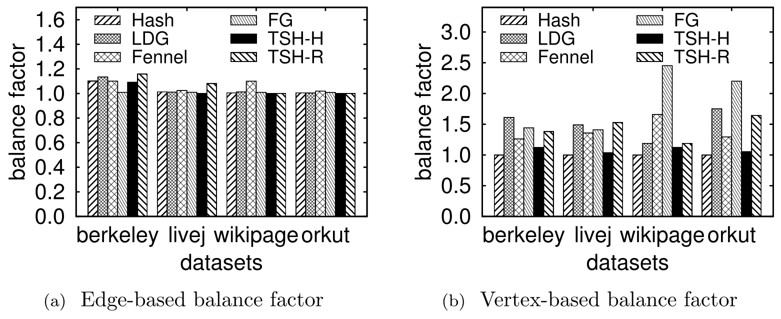


Fig. 4. The balance factors w.r.t. edges and vertices ($K = 20$).

5.2. Partitioning efficiency

We then explore the efficiency features with the sending buffer size fixed to 10^4. Table 2 gives the detailed reports of C_p, C_c, and their sum C. Because the runtime of Berkeley is less than 1 s, we omit it for brevity.

For C_p, we can easily find that *Hash*, *TSH-H* and *TSH-R* have almost the similar performance. All of them run 2.6–8.4 times faster than the centralized *LDG*/*Fennel*/*FG*. In particular, the parallel execution of the latter brings marginal benefit because of broadcasting costs. For C_c, *TSH-R*, (p)*LDG*, (p)*Fennel*, and (p)*FG* reduce the computation runtime by roughly 16%, when compared with *Hash*. The reduction of runtime in BC-BSP is generally below that of communication ratios (28%–54%) shown in Fig. 3(b)-Fig. 3(d). Ref [22] also finds this phenomenon and makes an explanation. That is, there exist two kinds of messages on a worker: remote messages sent to other workers and local ones sent to that worker. Many systems including BC-BSP allocate more computation resources for the former than the latter. Clearly, the local message processing becomes a performance bottleneck if a good balanced graph partition is provided. Thus, the reduction of communication ratios cannot be fully transferred into the performance improvement. Note that under *LDG*, *Fennel* and *FG*, the C_c values are almost the same because of their similar communication ratios as shown in Fig. 3. We finally analyze the overall runtime C. *Hash* generally beats (p)*LDG*, (p)*Fennel* and (p)*FG*, since the communication gain of the latter is offset by the expensive partitioning cost. Further, *TSH-R* strikes a good balance between partitioning quality (C_c) and efficiency (C_p). It achieves the most significant gain. The runtime improvement is up to 16% (from 9%) compared against *Hash* and 19% (from 15%) compared against *pLDG*, *pFennel* and *pFG*.

5.3. Determining the batch size threshold of TSH

TSH-R usually exhibits prominent performance in partitioning quality and efficiency. However, it requires to periodically synchronize the penalty function to well balance workload. Fig. 5 depicts the impact of different synchronization intervals, i.e., batch size thresholds η, on the overall runtime C, communication ratio, and edge-based load balance factor.

First of all, increasing the batch size significantly drops the times of synchronizing penalty functions and then reduces the overall runtime, as shown in Fig. 5(a). Second, Fig. 5(b) shows that the communication ratio can be slightly optimized when the batch size is extremely large. A large synchronization delay is equivalent to relaxing the balance constraint. Then partitioning adjacency lists will heavily depend on the intersection of T_i and

Table 2
Runtime (second; sending buffer = 10^4; $K = 20$).

Solutions	Livej			Wiki			Orkut		
	C_p	C_c	C	C_p	C_c	C	C_p	C_c	C
Hash	10.7	110.3	121.0	14.1	173.0	187.1	18.1	265.1	283.2
LDG	50.8	90.4	141.2	88.2	141.8	229.9	130.7	217.3	348.0
Fennel	53.1	89.4	142.5	90.7	138.2	228.9	140.8	213.4	354.2
FG	51.4	88.8	140.2	88.4	138.4	226.8	129.5	212.1	341.6
pLDG	39.0	91.2	130.2	60.4	140.9	201.3	76.3	215.0	291.3
pFennel	41.9	90.2	132.1	64.7	136.9	201.6	84.4	214.1	298.5
pFG	38.4	92.6	131.0	60.9	139.5	200.4	78.9	213.0	291.9
TSH-H	19.6	95.6	115.2	19.9	150.7	170.6	15.8	227.9	243.7
TSH-R	19.0	91.1	110.1	17.1	145.8	162.9	15.5	221.8	237.3

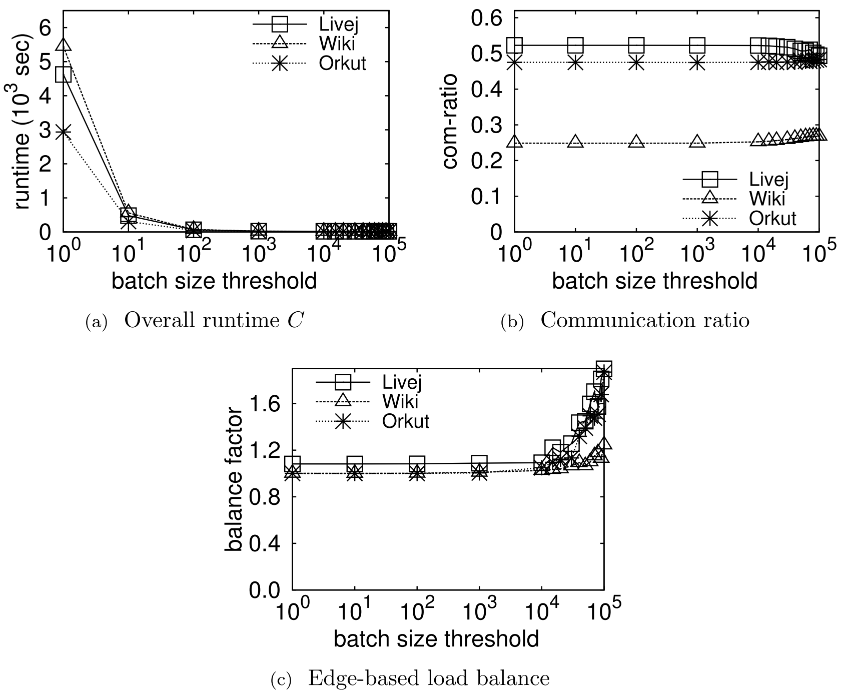


Fig. 5. The impact of η on *TSH-R* (sending buffer = 10^4; $K = 20$).

Γ_i as seen in Eq. (6), in order to combine messages as much as possible. The side effect is of course the skewed distribution of edges, as plotted in Fig. 5(c). Hence, in experiments above, we use $\eta = 10^3$ as the default value to balance the partitioning cost and the load balance factor. *TSH-H* uses the same setting.

5.4. Scalability

We next study how partitioning quality and efficiency scale when varying the K values, i.e., the number of workers. The built-in fair scheduler in BC-BSP guarantees that no matter how many workers there are, they can be evenly scheduled onto 20 physical machines. Here we call the sending buffer size in BC-BSP as *SB* for short. When analyzing the partitioning quality, we exclude *pLDG*, *pFennel* and *pFG* in figures for brevity because they provide similar performance to their centralized versions. Without loss of generality, all experiments are run over Wikipage.

Fig. 6(a) and (b) plot the communication ratio ($\frac{|M|}{|E|}$) as a function of K. The results include 2 different values for *SB*: 10^4 and *inf*. On both settings, *TSH-R* significantly outperforms *Hash* and *TSH-H*, and consistently provides a comparable performance to *LDG*, *Fennel* and *FG*. Note that the variation of K may affect $|M|$, the number of messages actually transmitted via network. Intuitively, with increasing K, the total number of messages generated by all workers will increase because more edges become cut edges. $|M|$ generally increases. However, the number of messages generated by a single worker may decrease since the sub-graph assigned to it becomes small. Given a fixed-size sending buffer on each worker, that potentially increases the probability of combining messages, leading to a reduction of $|M|$. As a result, the variation of $|M|$ is non-deterministic. Then we can explain why the ratio temporarily drops around $K = 60$ in Fig. 6(a). In particular, for larger values of K such as $K \geq 100$, the number of messages generated by a worker is so low that the sending buffer can hold all messages to fully combine them. In this scenario, the combination effect is fixed. $|M|$ thereby consistently increases with K. Fig. 6(b) also verifies our analysis because *SB=inf* always guarantees a full combination. Besides, sub-figure (a) reveals that *Fennel* beats all other solutions when $2 \leq K \leq 10$, with a clear performance gap. However, the gap narrows with growing K. This is mainly due to the enhanced effect of combination as explained above, which weakens the effect of partitioning.

We then investigate the partitioning quality in terms of edge-based load balance factor. We omit the vertex-based factor because Fig. 4(b) and Table 2 have shown that a moderately skewed distribution of vertices does not affect the overall runtime of PageRank. Fig. 6(c) shows that with growing K, the factors of all solutions slightly increase, because the average workload $\frac{|G|}{K}$ becomes smaller and then is more sensitive to the real workload distribution. However, the increased factors are still acceptable (< 1.15).

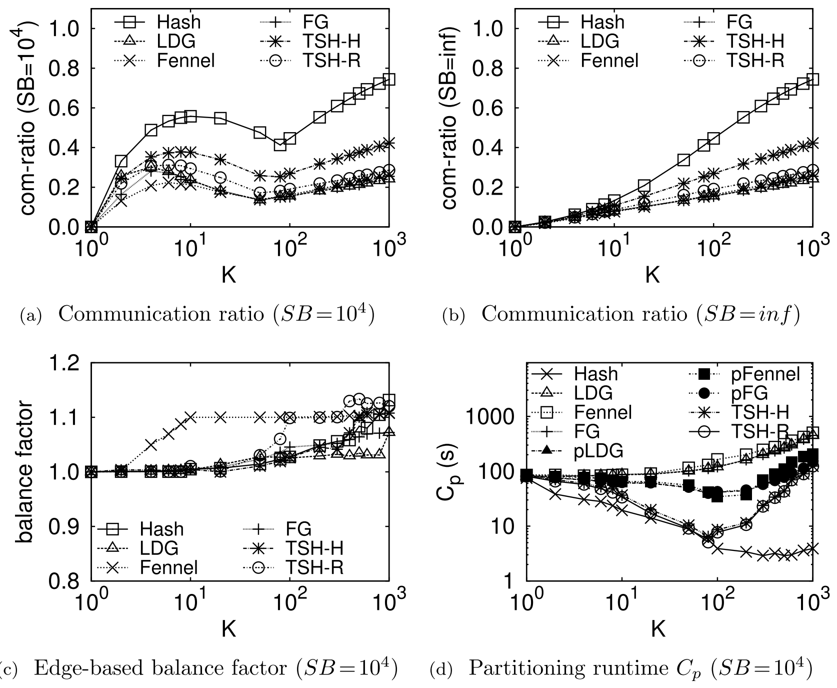


Fig. 6. Scalability (Wikipage).

We next focus on the partitioning efficiency, i.e., C_p. Fig. 6(d) shows the results of all solutions including the parallel variants of *LDG*, *Fennel* and *FG*. The trends of C_p against K fall into three categories. (1) The C_p values of centralized *LDG*, *Fennel* and *FG* monotonously increase with K. This is mainly because a large K increases the number of candidate target sub-graphs when deciding the placement of a newly loaded adjacency list, increasing the computation complexity. (2) By contrast, C_p simply drops with K for *Hash* due to the parallel execution. (3) Different from the two cases mentioned above, the variation of C_p is more complex for parallel partitioning methods *pLDG*, *pFennel*, *pFG*, *TSH-H* and *TSH-R*. It first drops like the trend in *Hash* for the same reason. When K further increases, however, the number of adjacency lists on each worker decreases. Then we need to use a short interval to synchronize the penalty function value to achieve a good balance factor. The resulting C_p value becomes large. Because *pLDG*, *pFennel* and *pFG* need to additionally synchronize the placements of adjacency lists, they consistently cost more time than *TSH-H* and *TSH-R*. The final case motivates us to smartly select a proper partitioning solution between *TSH-R* and *Hash* based on K, to dynamically optimize C_p. We plan to investigate this issue as future work.

5.5. Comparison over random graphs

Fig. 7 repeats the experiments in Fig. 3(c) but on a random graph with the same number of vertices and edges in Wikipage. It is generated as described in Section 3.2. Comparing Fig. 7 with Fig. 3(c), all partitioning solutions perform poorly on the random graph. In particular, *TSH-R* and *TSH-H* are even roughly equivalent to *Hash*. This is mainly due to the absence of locality. In another word, the locality property is significantly important for *TSH* to implement distributed streaming partitioning with high quality.

5.6. Comparison against non-streaming partitioning methods

We now compare our best solution *TSH-R* against complex non-streaming techniques *GGG*, *PartFG*, and *XtraPuLP*. Here, the sending buffer is always set to 10^4. Further, because *GGG* is

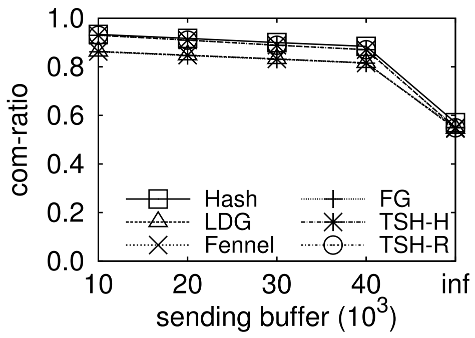
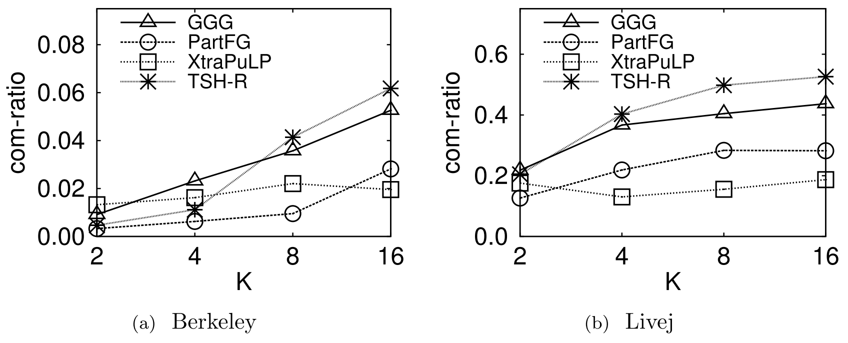


Fig. 7. Communication ratio over a random graph ($K = 20$).

a 2-way partitioning method, we solve the K-way partitioning problem by recursively invoking *GGG*. Then after $\log K$ passes, we obtain K sub-graphs. The K values in this group of experiments then range in $\{2, 4, 8, 16\}$. When running *PartFG*, we follow the author's default parameter setting, that is, half of the graph is re-streamed 10 times. Different from other competitors, *XtraPuLP* is more complex because of its multiple objectives (minimizing the global cut edges and the maximal cut edges of any sub-graph) and multiple constraints (vertex- and edge-based balance). The partitioning quality and efficiency largely depend on which objective and constraint are selected and/or set. To perform a fair comparison, we set the target vertex- and edge-based balance factors as the same values reported by *TSH-R*. Also, after many trials, we find that both of the two objective metrics should be used for good quality. Besides, in *XtraPulP*, we disable the thread level parallelism in each worker since other solutions now do not support that.

Quality: Fig. 8 investigates the variation of the communication ratios over different graphs. We analyze the behaviors of all competitors in the following. (1) *TSH-R* vs. *GGG*: Both are trying to utilize the locality provided by BFS. *TSH-R* should work well if the input graph is crawled by BFS. However, the condition cannot be strictly guaranteed in real world. Instead, *GGG* grows a sub-graph



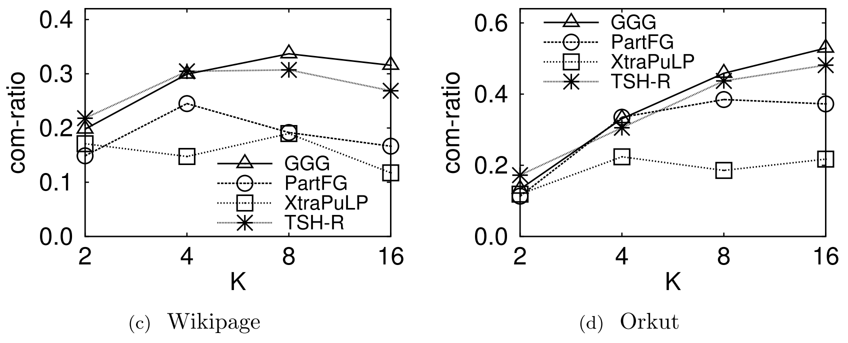


Fig. 8. The com-ratios of non-streaming methods (sending buffer = 10^4).

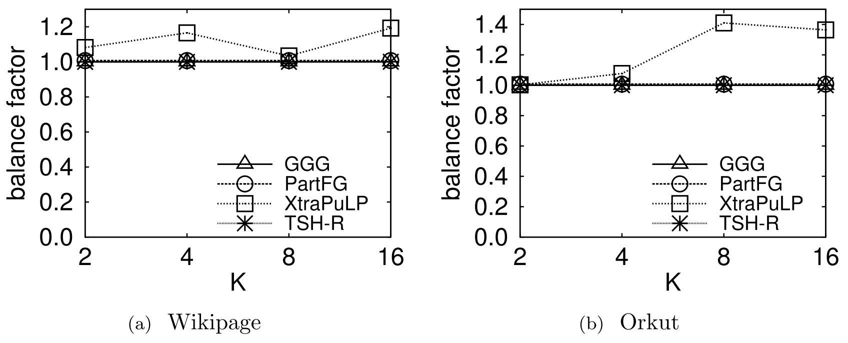


Fig. 9. The edge-based balance factor of non-streaming methods (sending buffer = 10^4; $K = 20$).

around a source vertex by including newly traversed adjacency lists in BFS. In theory, *GGG* should beat *TSH-R* because of its strict BFS traverse. However, it simply terminates when half of edges have been included in the new sub-graph. The lack of penalty function prevents it from including more reasonable data that so far has not been traversed. Hence, for *TSH-R* and *GGG*, one cannot consistently outperform the other. (2) Because of multiple re-streaming operations, *PartFG* has a steady *com-ratio* lower than *GGG* and *TSH-R*. (3) Before analyzing *XtraPuLP*, we first report the edge-based balance factor on some graphs since this metric is important for computation runtime C_c as shown in Section 5.2. In Fig. 9, all methods except *XtraPuLP* can well balance the number of edges across sub-graphs. For the exception, comparing Fig. 8(c) and Fig. 9(a) (or Fig. 8(d) and Fig. 9(b)), we find that *XtraPuLP* automatically strikes a balance between reducing *com-ratio* and guaranteeing the edge-based load balance target. In fact, it is designed to strictly guarantee the user-specified vertex-based balance target but only try its best effort for edge-based balance. That might affect the runtime of real applications as we will show in later. Note that *com-ratio* on Wikipage first increases and then decreases. We have given the explanation (for Fig. 6(a)) in Section 5.4.

Efficiency: Table 3 then reports the runtime performance of non-streaming methods. Now the parallel variants of *GGG* and *PartFG* are included. We use Wikipage as the example graph since the up-to-date *XtraPuLP* achieves good quality considering *com-ratio* and the edge-based balance factor.

As expected, C_p for *GGG* monotonously increases with K because the number of data scans $\log K$ is proportional to K. C_p for *PartFG* is also large but not sensitive to K as *GGG*. Instead, it is decided by the re-streaming times. Further, although C_p is reduced in the parallel *pGGG* and *pPartFG*, it is still greater than our *TSH-R* where the input data is accessed only once. That leads to an overall success on runtime C (up to 26% and 48% improvements respectively compared with *pGGG* and *pPartFG*).

Notice that *XtraPuLP* limits the iteration number of label propagation for fast partitioning. Together with the efficient C++ implementation, its C_p is smaller than that in *pGGG* and *pPartFG* in most cases but still greater than that in *TSH-R*. On the other hand, *XtraPuLP* theoretically can decrease C_c because of its low *com-ratio*. However, when the distribution of edges across sub-graphs is heavily skewed, workers running fast must wait for workers running slowly. The gain achieved by communication reduction

Table 3
Runtime of non-streaming methods (second; Wikepage; sending buffer = 10^4).

Solutions	$K=4$			$K=8$			$K=16$		
	C_p	C_c	C	C_p	C_c	C	C_p	C_c	C
GGG	123.1	652.9	775.9	187.3	335.4	522.7	264.7	172.6	437.2
PartFG	410.5	606.5	1017.0	417.2	241.4	658.6	425.4	138.1	563.6
pGGG	92.4	654.9	747.3	106.1	336.6	442.7	90.4	174.1	264.5
pPartFG	334.6	608.4	943.0	304.6	244.1	548.7	238.1	139.2	377.3
XtraPuLP	107.4	592.7	700.1	97.9	241.8	339.7	61.5	146.4	207.9
TSH-R	57.6	656.1	713.7	41.2	333.5	374.7	25.4	168.6	193.9

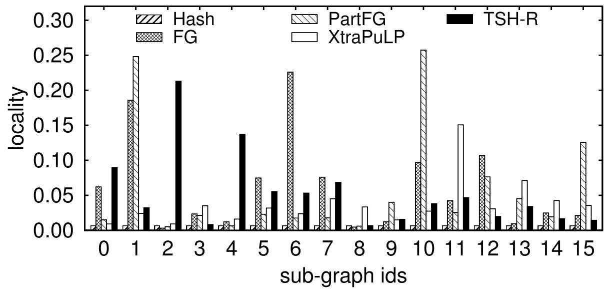


Fig. 10. Locality of each sub-graph (Berkeley, sending buffer = 10^4, $K = 16$).

might be offset by the waiting cost (as shown in the case of $K = 16$). Finally, considering the overall runtime C, *XtraPuLP* beats all other competitors and even our *TSH-R*. For a fair comparison, we also implement *TSH-R* in C++ and find that the Java-version runs roughly 2x slower than the C++ version. This is also validated in Ref. [23]. Thus, *TSH-R* still works better than *XtraPuLP*.

5.7. Locality of partitioned sub-graphs

We finally analyze the locality L_{G_i} for each sub-graph G_i after partitioning. L_{G_i} is computed using Eq. (3) but V is replaced with V_i in G_i. A big L_{G_i} is preferred because it indicates that there are many common out-neighbors and then messages for such neighbors can be combined in high probability, that decreases *com-ratio*. Based on the *com-ratio* performance shown in Figs. 3 and 8, we select *FG*, *PartFG*, *XtraPuLP* and our *TSH-R* as representative advanced methods. We compare them with the baseline *Hash* method. Without loss of generality, all tests are run on Berkeley with sending buffer equal to 10^4 and $K = 16$. Fig. 10 then depicts L_{G_i} for each sub-graph G_i.

Clearly, the sub-graph locality under advanced methods is significantly greater than that under random *Hash*. Further, all advanced methods except *TSH-R* try to decrease the number of cut edges between separated sub-graphs. In another word, for every sub-graph G_i, they aim to increase the number of inner edges that link to source vertices in V_i. That potentially increases the number of common out-neighbors for edges in E_i and hence *com-ratio* gets smaller. However, *TSH-R* directly optimizes *com-ratio* by grouping adjacency lists with the most common out-neighbors. It can quickly partition a graph and provide an acceptable high locality L_{G_i} (w.r.t. low *com-ratio*). Thus, *TSH-R* works best in terms of the overall runtime performance as shown in Tables 2 and 3.

6. Related work

This section first lists representative Pregel-like systems and then summarizes existing partitioning techniques.

6.1. Pregel-like distributed graph computation systems

Distributed graph systems are receiving growing attention with the emergence of highly scalable iterative computation. Pregel [2] and Trinity [24] as early representatives are proposed by Google and Microsoft respectively but both of them are closed-source. To provide publicly available service, many open source implementations have been released, such as Apache Hama [3], Apache Giraph [4], GPS [5] and BC-BSP [8,9]. Besides, some researchers pioneer in iterative analytics on top of general-purpose data-flow frameworks, like PEGASUS [1], Haloop [25] and Spark [26]. All systems above necessitate a full pass over the input graph to load data and then partition data across workers before parallel computation.

6.2. Partitioning algorithms

This section outlines two important research branches for graph partitioning: offline and online, followed by a general overview of dynamic partitioning techniques. The related work dealing with the problem of hypergraphs is also given as hypergraphs capture the target vertex concept naturally.

Offline graph partitioning. There exists a rich literature on graph partitioning. Many approaches traverse vertices to partition data based on the graph topology. They scan the input graph more than one time to gradually refine the partitioning quality. The most well-known *multilevel* partitioning solution gradually coarsens the input graph and then divides the coarsest graph. Finally it projects the partition back towards the original graph. METIS [13,20], Scotch [27] and their parallel variants [28,29] fall into this category. There is yet, however, a costly maximal matching scheme for clustering in the coarsening phase. In particular, METIS [20] employs a greedy growing partitioning policy to divide the coarsest graph. It also utilizes BFS to optimize network communication. However, as shown in Section 5.6, it might work poorly due to the lack of penalty function. Worse, as a 2-way partitioning policy, it must access the input graph $\log K$ times to solve the K-way partitioning problem, increasing the partitioning runtime. MLP [30] replaces matching with connected component computation based on label propagation, yielding a better efficiency. Some approaches solely rely on *iteratively* propagating labels among vertices for parallel graph partitioning [31,32]. Based on label propagation, Slota et al. pay attention to partitioning with multiple constraints and multiple objectives respectively in shared-memory [33,34] and distributed-memory [35] environments. However, their method called *XtraPuLP* cannot strictly guarantee the edge-based balance based on our test. That might affect the iterative computation efficiency due to the blocking cost across workers. Also, iterative label propagation costs more time than our streaming solutions.

Because of multiple data scans, neither *multilevel* nor *iterative* approaches can be run along loading the input graph. They are so-called offline partitioning algorithms and thereby different from our online solution.

Online/streaming graph partitioning. Some researchers assume that graph data arrive streamingly. With the arrival of vertices and/or edges, we can compute the distribution of already placed data and then decide the placement of newly arrived data. Clearly, streaming partitioning requires a single data scan and thereby can be performed along loading data in Pregel-like systems, referred to here as online partitioning. Online approaches largely improve the partitioning efficiency although to a certain extent they compromise the quality. This comprise leads to better overall performance for a graph processing job [15].

Online partitioning is usually performed in a *centralized* fashion so as to accurately keep track of the distribution of arrived data, like LDG [14] and Fennel [15]. The partitioning quality can be further refined by feeding the output of previously performed streaming partitioning into the current execution. We call this full re-streaming if the whole output is used [21] and partial re-streaming [16], otherwise. Some variants customized for random walk algorithms [36] and heterogeneous computation environments [37] have also been explored. However, graphs are rapidly growing in size, that is well beyond the capability of a single machine. Note that Shi et al. parallelize the placement decision of vertices in shared-memory platforms, but data are still streamed serially, leading to a single machine bottleneck [38]. On the other hand, Hash can be easily run in distributed environments but at the expense of extremely poor partitioning quality.

Edge-cut vs. vertex-cut. Till now, all offline and online approaches we summarized partition vertices into disjoint parts while edges with endpoints in two different parts are called cut/external edges and will incur communication costs in iterative computation. However, vertices can be replicated in different parts (cut vertices) to transfer external edges into internal ones (both endpoints are in the same part). That is, vertices are partitioned into overlapped parts and then communication happens between vertices and their replicas. Clearly, the two partitioning policies have totally different goals, i.e., minimizing the number of cut edges and vertices, respectively, and then the heuristics employed are technically orthogonal. Also, performing computation over a vertex-cut based partition requires much more complicated system design to synchronize values of vertex replicas. To our knowledge, only a few systems support that, like PowerGraph [39] and GraphX [6]. Hence, although there exist some distributed online vertex-cut partitioning approaches [39–41], we still need to develop new solutions for the widely used edge-cut based partitioning.

Dynamic graph partitioning. Note that the communication workloads may dynamically change during iterations because vertices converged usually do not communicate with others. The original partitioning result thereby becomes suboptimal. Dynamic partitioning can react to the change by migrating data across workers in real time [5,42,43]. It can be performed over the output of any original partitioning algorithm including *TSH*. Hence, it is complementary to our proposal.

Hypergraph partitioning. Last but not least, the communication cost of a hyperedge in hypergraphs is also dominated by the distribution of target vertices. Theoretically, Pregel-like systems can first transfer the input graph into a hypergraph and then re-load and partition the transferred graph with hypergraph partitioning techniques. However, that yields yet another full pass over input data, increasing C_p compared with streaming solutions. Worse, hypergraph partitioning is now employing offline (*multilevel* [44–47] and *iterative* [48]) or single-machine online [49] heuristics. The former require to access graph data multiple times, that is time-consuming even though some of them can be run in parallel like Parkway [46], Zoltan [47] and Social Hash Partitioner [48]. On the other hand, the latter has poor scalability due to limited compute power.

7. Conclusion

This paper investigates the problem of graph partitioning in Pregel-like systems. We re-define graph partitioning by emphasizing the importance of partitioning efficiency and reducing the number of target vertices in each sub-graph. We accordingly propose a new partitioning solution *TSH* to strike a good balance between partitioning quality and efficiency by leveraging the newly found graph locality feature. Extensive experiments confirm the effect of our method.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

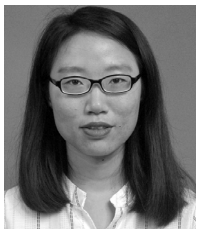
Acknowledgments

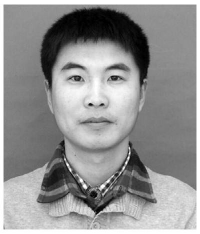
This work was supported by China Postdoctoral Science Foundation Grant (2019M652474 and 2019M652473) and the National Natural Science Foundation of China (61433008, 61872070, 61528203, and 61602103).

References

[1] U. Kang, C.E. Tsourakakis, C. Faloutsos, Pegasus: a peta-scale graph mining system implementation and observations, in: Proc. of ICDM, IEEE, 2009, pp. 229–238.
[2] G. Malewicz, M.H. Austern, A.J. Bik, J.C. Dehnert, I. Horn, N. Leiser, G. Czajkowski, Pregel: a system for large-scale graph processing, in: Proc. of SIGMOD, ACM, 2010, pp. 135–146.
[3] S. Seo, E.J. Yoon, J. Kim, S. Jin, J.-S. Kim, S. Maeng, Hama: an efficient matrix computation with the mapreduce framework, in: Proc. of CloudCom, IEEE, 2010, pp. 721–726.
[4] Apache Giraph. http://giraph.apache.org/.
[5] S. Salihoglu, J. Widom, Gps: a graph processing system, in: Proc. of SSDBM, ACM, 2013, p. 22.
[6] J.E. Gonzalez, R.S. Xin, A. Dave, D. Crankshaw, M.J. Franklin, I. Stoica, Graphx: Graph processing in a distributed dataflow framework, in: Proc. of OSDI, 2014, pp. 599–613.
[7] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, J.M. Hellerstein, Distributed graphlab: a framework for machine learning and data mining in the cloud, PVLDB 5 (8) (2012) 716–727.
[8] Y. Bao, Z. Wang, Y. Gu, G. Yu, F. Leng, H. Zhang, B. Chen, C. Deng, L. Guo, BC-BSP: A BSP-based parallel iterative processing system for big data on cloud architecture, in: Proc. of DASFAA Workshop, Springer, 2013, pp. 31–45.
[9] Z. Wang, Y. Gu, Y. Bao, G. Yu, J.X. Yu, Hybrid pulling/pushing for I/O-efficient distributed and iterative graph computing, in: Proc. of SIGMOD, ACM, 2016, pp. 479–494.
[10] Z. Wang, L. Gao, Y. Gu, Y. Bao, G. Yu, A fault-tolerant framework for asynchronous iterative computations in cloud environments, IEEE Trans. Parallel Distrib. Syst. 29 (8) (2018) 1678–1692.
[11] K. Andreev, H. Racke, Balanced graph partitioning, in: Proc. of the Sixteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA), ACM, 2004, pp. 120–124.
[12] R.G. Michael, S.J. David, Computers and Intractability: A Guide to the Theory of NP-completeness, WH Free. Co., San Fr, 1979, pp. 90–91.
[13] G. Karypis, V. Kumar, Multilevel k-way partitioning scheme for irregular graphs, J. Parallel Distrib. Comput. 48 (1) (1998) 96–129.
[14] I. Stanton, G. Kliot, Streaming graph partitioning for large distributed graphs, in: Proc. of SIGKDD, ACM, 2012, pp. 1222–1230.
[15] C. Tsourakakis, C. Gkantsidis, B. Radunovic, M. Vojnovic, Fennel: streaming graph partitioning for massive scale graphs, in: Proc. of WSDM, ACM, 2014, pp. 333–342.
[16] G. Echbarthi, H. Kheddouci, Fractional greedy and partial restreaming partitioning: new methods for massive graph partitioning, in: Proc. of Big Data, IEEE, 2014, pp. 25–32.
[17] Apache Hadoop. http://hadoop.apache.org/.
[18] S. Thomas, W. Dorothea, Finding, counting and listing all triangles in large graphs, an experimental study, in: Proc. of Experimental and Efficient Algorithms, 4th International Workshop, WEA, 2005, pp. 606–609.
[19] M. Najork, J.L. Wiener, Breadth-first crawling yields high-quality pages, in: Proc. of WWW, ACM, 2001, pp. 114–118.

[20] G. Karypis, V. Kumar, A fast and high quality multilevel scheme for partitioning irregular graphs, SIAM J. Sci. Comput. 20 (1) (1998) 359–392.
[21] J. Nishimura, J. Ugander, Restreaming graph partitioning: simple versatile algorithms for advanced balancing, in: Proc. of SIGKDD, ACM, 2013, pp. 1106–1114.
[22] Y. Shao, B. Cui, L. Ma, Page: a partition aware engine for parallel graph computation, IEEE Trans. Knowl. Data Eng. 27 (2) (2015) 518–530.
[23] A. Kyrola, G.E. Blelloch, C. Guestrin, Graphchi: large-scale graph computation on just a PC, in: OSDI, vol. 12, 2012, pp. 31–46.
[24] B. Shao, H. Wang, Y. Li, Trinity: a distributed graph engine on a memory cloud, in: Proc. of SIGMOD, ACM, 2013, pp. 505–516.
[25] Y. Bu, B. Howe, M. Balazinska, M.D. Ernst, Haloop: efficient iterative data processing on large clusters, PVLDB 3 (1–2) (2010) 285–296.
[26] Apache Spark. http://spark.apache.org/.
[27] F. Pellegrini, J. Roman, Scotch: a software package for static mapping by dual recursive bipartitioning of process and architecture graphs, in: Proc. of High-Performance Computing and Networking, Springer, 1996, pp. 493–498.
[28] G. Karypis, V. Kumar, A parallel algorithm for multilevel graph partitioning and sparse matrix ordering, J. Parallel Distrib. Comput. 48 (1) (1998) 71–95.
[29] C. Chevalier, F. Pellegrini, Pt-scotch: a tool for efficient parallel graph ordering, Parallel Comput. 34 (6) (2008) 318–331.
[30] L. Wang, Y. Xiao, B. Shao, H. Wang, How to partition a billion-node graph, in: Proc. of ICDE, IEEE, 2014, pp. 568–579.
[31] J. Ugander, L. Backstrom, Balanced label propagation for partitioning massive graphs, in: Proc. of WSDM, ACM, 2013, pp. 507–516.
[32] F. Rahimian, A.H. Payberah, S. Girdzijauskas, M. Jelasity, S. Haridi, Ja-be-ja: a distributed algorithm for balanced graph partitioning, in: Proc. of SASO, IEEE, 2013, pp. 51–60.
[33] G.M. Slota, K. Madduri, S. Rajamanickam, Pulp: scalable multi-objective multi-constraint partitioning for small-world networks, in: Proc. of Big Data, IEEE, 2014, pp. 481–490.
[34] G.M. Slota, K. Madduri, S. Rajamanickam, Complex network partitioning using label propagation, SIAM J. Sci. Comput. 38 (5) (2016) S620–S645.
[35] G.M. Slota, S. Rajamanickam, K. Devine, K. Madduri, Partitioning trillion-edge graphs in minutes, in: Proc. of IPDPS, IEEE, 2017, pp. 646–655.
[36] X. Liu, Y. Zhou, X. Guan, C. Shen, A feasible graph partition framework for parallel computing of big graph, Knowl.-Based Syst. 134 (2017) 228–239.
[37] K.-k. Hu, G.-s. Zeng, H.-w. Jiang, W. Wang, Partitioning big graph with respect to arbitrary proportions in a streaming manner, Future Gener. Comput. Syst. (2017) Available online.
[38] Z. Shi, J. Li, P. Guo, S. Li, D. Feng, Y. Su, Partitioning dynamic graph asynchronously with distributed fennel, Future Gener. Comput. Syst. 71 (2017) 32–42.
[39] J.E. Gonzalez, Y. Low, H. Gu, D. Bickson, C. Guestrin, Powergraph: distributed graph-parallel computation on natural graphs, in: Proc. of OSDI, vol. 12, 2012, p. 2.
[40] R. Chen, J. Shi, Y. Chen, H. Chen, Powerlyra: differentiated graph computation and partitioning on skewed graphs, in: Proc. of EuroSys, ACM, 2015, p. 1.
[41] F. Petroni, L. Querzoni, K. Daudjee, S. Kamali, G. Iacoboni, Hdrf: stream-based partitioning for power-law graphs, in: Proc. of CIKM, ACM, 2015, pp. 243–252.
[42] Z. Khayyat, K. Awara, A. Alonazi, H. Jamjoom, D. Williams, P. Kalnis, Mizan: a system for dynamic load balancing in large-scale graph processing, in: Proc. of EuroSys, ACM, 2013, pp. 169–182.
[43] Z. Shang, J.X. Yu, Catch the wind: graph workload balancing on cloud, in: Proc. of ICDE, IEEE, 2013, pp. 553–564.
[44] U.V. Catalyurek, C. Aykanat, Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication, IEEE Trans. Parallel Distrib. Syst. 10 (7) (1999) 673–693.
[45] D. Mehmet, K. Kamer, U. Bora, V.Ç. Ümit, Hypergraph partitioning for multiple communication cost metrics: model and methods, J. Parallel Distrib. Comput. 77 (2015) 69–83.
[46] T. Aleksandar, J.K. William, Parallel multilevel algorithms for hypergraph partitioning, J. Parallel Distrib. Comput. 68 (5) (2008) 563–581.
[47] D.D. Karen, G.B. Erik, T.H. Robert, H.B. Rob, V.Ç. Ümit, Parallel hypergraph partitioning for scientific computing, in: Proc. of IPDPS, IEEE, 2006.
[48] I. Kabiljo, B. Karrer, M. Pundir, S. Pupyrev, A. Shalita, A. Presta, Y. Akhremtsev, Social hash partitioner: a scalable distributed hypergraph partitioner, Proc. of the VLDB Endowment 10 (11) (2017) 1418–1429.
[49] A. Dan, J. Iglesias, M. Vojnovic, Streaming min-max hypergraph partitioning, in: Proc. of NIPS, 2015, pp. 1900–1908.


Ning Wang received her Ph.D. degree in computer software and theory from Northeastern University at China, in 2017. She is currently a lecturer at Ocean University of China. Her current research interest lies in data privacy protection and big data analytics.


Zhigang Wang received the Ph.D. degree in computer software and theory from Northeastern University, China, in 2018. He is currently a lecturer at Ocean University of China. He has been a visiting PhD student in University of Massachusetts Amherst during December 2014 to December 2016. His research interests include cloud computing, distributed graph processing and machine learning.

Yu Gu received the Ph.D. degree in computer software and theory from Northeastern University, China, in 2010. Currently, he is a professor and the PhD supervisor at Northeastern University, China. His current research interests include big data analysis, spatial data management and graph data management. He is a senior member of the China Computer Federation (CCF).

Yubin Bao received the Ph.D. degree in computer software and theory from Northeastern University, China, in 2003. Currently, he is a professor at Northeastern University, China. His current research interests include data warehouse and OLAP, graph data management, and cloud computing. He is a senior member of the China Computer Federation (CCF).

Ge Yu received the Ph.D. degree in computer science from Kyushu University of Japan, in 1996. He is currently a professor and the PhD supervisor at Northeastern University of China. His research interests include distributed and parallel database, OLAP and data warehousing, data integration, graph data management, etc. He is a member of the IEEE Computer Society, IEEE, ACM, and a Fellow of the China Computer Federation (CCF).