Distributed and Parallel Databases (2020) 38:1–29
https://doi.org/10.1007/s10619-019-07256-z

An investigation of big graph partitioning methods for distribution of graphs in vertex-centric systems

Nasrin Mazaheri Soudani[1] · **Afsaneh Fatemi**[1] · **Mohammadali Nematbakhsh**[1]

Published online: 6 February 2019

Abstract
Relations among data entities in most big data sets can be modeled by a big graph. Implementation and execution of algorithms related to the structure of big graphs is very important in different fields. Because of the inherently high volume of big graphs, their calculations should be performed in a distributed manner. Some distributed systems based on vertex-centric model have been introduced for big graph calculations in recent years. The performance of these systems in terms of run time depends on the partitioning and distribution of the graph. Therefore, the graph partitioning is a major concern in this field. This paper concentrates on big graph partitioning approaches for distribution of graphs in vertex-centric systems. This briefly discusses vertex-centric systems and formulates different models of graph partitioning problem. Then, a review of recent methods of big graph partitioning for these systems is shown. Most recent methods of big graph partitioning for vertex centric systems can be categorized into three classes: (i) stream-based methods that see vertices or edges of the graph in a stream and partition them, (ii) distributed methods that partition vertices or edges in a distributed manner, and (iii) dynamic methods that change partitions during the execution of algorithms to obtain better performance. This study compares the properties of different approaches in each class and briefly reviews methods that are not in these categories. This comparison indicates that The streaming methods are good choices for initial load of the graph in Vertex-centric systems. The distributed and dynamic methods are appropriate for long-running applications.

Keywords Graph partitioning · Vertex-centric systems · Big graphs · Distributed computing

✉ Afsaneh Fatemi
a_fatemi@eng.ui.ac.ir

Nasrin Mazaheri Soudani
mazaheri@eng.ui.ac.ir

Mohammadali Nematbakhsh
nematbakhsh@eng.ui.ac.ir

[1] Department of Computer Engineering, University of Isfahan, Isfahan, Iran

1 Introduction

With the advancement of technology, the speed of content creation is increasing and data sets with high volume, high change velocity and high variety are created. Web pages, linked data, social networks' data and protein interactions' data are some examples of these sets. They are named big data [44,53,58].

Relations among data entities in big data sets can be modeled by graphs. Implementation and execution of algorithms on the structure of these graphs is very important in different fields. Examples of such algorithms are calculation of page ranks in the web [96], community detection and advertisement propagation in social networks [18], calculation of maximum flow and routing in communication networks [37] and protein communication analysis for diagnosis of diseases [63].

Since these big graphs have such high volume of data, their calculations should be made in a distributed manner. For example, the graph of Facebook users has 2.34 billion vertices and 346 billion edges as of 2018 [25]. Twitter has 326M active users and 208 follow per user monthly [4].

The vertex-centric computing model has been introduced for computations on big graphs in recent years. In this model, graph structure is partitioned and distributed over a cluster of computers. Each vertex or edge may have some related data values. A graph algorithm is described with a user-defined function. Each machine runs this function recursively for its active vertices. With execution of this function for a vertex, the data of adjacent vertices or edges may be read and its data may be updated and be exposed to its adjacent vertices or edges. Each vertex has a local view of the graph structure. It has access to the data of adjacent vertices or edges. This model is named Think Like A Vertex (TLAV) [52].

Although there are a variety of computational models for graphs, this model is more popular, because the description of graph algorithms is easier with this model. Today, many graph computing systems such as GraphLab [51], PowerGraph [30], GraphX [98], Giraph [17] and PowerLyra [15] has been developed based on this model.

Graph partitioning has a considerable effect on the efficiency of computation in the vertex-centric systems. The loads of machines depend on the number of their active vertices or edges. Also, the amount of communication among them depends on the replication number of vertices or edges. Graph partitioning should balance loads of different machines and minimize their inter-communication.

This is a balanced k-way graph partitioning problem. This problem has been studied extensively and there are many algorithms [13,28,29] for solving it. But, a number of new challenges for distribution of graphs in vertex-centric systems exists, which makes this problem as important recent research topic.

- Traditional graph partitioning algorithms need random access to graph vertices and have poor locality. They cannot be used for partitioning big graphs [67].
- The structure of real world graphs may change during the time. Therefore, the incremental graph partitioning algorithms that can adapt their results with graph changes are more desirable [88].
- All vertices of the graph may not be active in all steps of an application. Therefore, load balancing on different machines may not be guaranteed in vertex-centric

systems, even with optimal partitioning. A graph partitioning algorithm should consider the characteristic of the running application [88].
– Most of the computational clusters have heterogeneous architecture and their nodes have various computing and communication abilities. A graph partitioning approach should consider the heterogeneity in hardware [56,100].
– The implementation details of various vertex-centric systems can affect the efficiency of graph partitioning. For example, graph systems may be synchronous or asynchronous. They can use either message passing or shared memory for communication among vertices [34].

Recent works for big graph partitioning on vertex-centric systems are often based on three main approaches: (i) stream-based methods that partition graphs serially in a single pass and make permanent partition assignment the first time they examine each vertex or edge [80], (ii) distribute methods that partition graphs in a distributed manner [86] and (iii) dynamic graph partitioning methods that monitor loads and communications of machines during the execution of algorithms and change partitions based on this information.

The approaches regarding big graph partitioning are investigated and compared in this paper. Although graph partitioning has many applications in different fields, this paper only focuses on the application of graph partitioning in the distribution of big graphs in vertex-centric systems. The comparisons are based on real world graphs, such as web pages' or social networks' graphs.

The comparisons indicate that streaming methods are good choices for loading and partitioning the graph in a pipeline because they are incremental and have linear time and space complexity. Distributed methods need separate partitioning phase. They can be used for long-running jobs or when several graph algorithm should be executed in one partitioned graph. Dynamic methods are appropriate for long-running jobs. They are also appropriate for multiphase algorithms with different communication and calculation patterns for each phase on a heterogeneous distributed cluster.

The remainder of this paper is organized as follows: The balanced k-way graph partitioning problem is defined in Sect. 2. Some common vertex-centric systems and their properties are compared in Sect. 3. Stream-based and distributed approaches for static graph partitioning are discussed in Sects. 4 and 5. Methods of dynamic graph partitioning are discussed in Sect. 6. Some different partitioning approaches are considered in Sect. 7 and the paper is concluded in Sect. 8.

2 Balanced k-way graph partitioning

The most common model of balanced k-way graph partitioning is the edge cut(EC) model. In this model, the vertices of graph are placed in different partitions. The objective is to balance the size of partitions and minimize the number of cut edges (edges among vertices in different partitions).

Assume that $G = (V, E)$ is a graph where, V is the set of vertices and E is the set of edges, it is desired to find the set of partitions $P = \{P_1, P_2, \ldots, P_k\}$ on the vertices

V that are pairwise disjoint and the union of which is equal to V. These partitions must meet the flowing two conditions:

$$\min_{P} \left|\left\{e|e = (v_i, v_j) \in E, v_i \in P_x, v_j \in P_y, x \neq y\right\}\right| \tag{1}$$

$$s.t.\ \frac{\max_i |P_i|}{\frac{1}{k}\sum_{i=1}^{k} |P_i|} \leq \epsilon \tag{2}$$

where k is the number of partitions and $\epsilon \geq 1$ is a constant number that identifies the acceptable imbalance [97].

In vertex-centric systems which apply the EC partitioning, the load of a machine may subject to the number of its vertices or edges. Hence, the size of a partition in the second condition can be either the number of vertices or edges. Some works evaluate the workload of a partition by the sum of its active vertices and edges [39,111]. A method for estimating the number of active vertices and edges during the execution of an algorithm in vertex-centric systems is presented in [39].

There might be another condition on the minimum size of partitions in addition to these two conditions. If the computing powers of different machines vary, the second condition should be changed such that the load of each machine will be less than its maximum computing power [106–108].

The edges constitute the communication channels in these systems. The number of cut edges indicates external communication overhead between different machines. When communication costs vary among different machines, each cut edge is assigned different weight. Consequently the partitioning algorithm should minimize the sum of these weights [100,107,108].

For vertex centric systems that store graph structure on the secondary memory, the speed of reading data from secondary storage is usually less than the network speed. So, decreasing the number of cut edges is no longer a primary concern in these systems. They need balanced size partitions with high speed sequential storage access [69].

If the graph represents the dependencies among tasks in a parallel system, partitioning of the graph should meet other constraint. There must not be a cycle among the yield partitions because they should be executed one after another [62].

The other model of graph partitioning is the vertex cut (VC) model. In this model edges are placed in different partitions. Each vertex is copied in different partitions containing its adjacent edges. The size of partitions in terms of the number of edges should be balanced and the number of copied versions of vertices should be minimized. The objective here is to find the set of partitions $P = \{P_1, P_2, \ldots, P_k\}$ on the edges that must meet the flowing two conditions:

$$\min_{P} \frac{1}{|V|} \sum_{v \in V} |P(v)| \tag{3}$$

$$s.t.\ \max_{p_i} |\{e \in E | P(e) = p_i\}| \leq \epsilon \frac{|E|}{k} \tag{4}$$

where $P(v)$ is the set of partitions that vertex v is copied in and $P(e)$ is the partition with edge e.

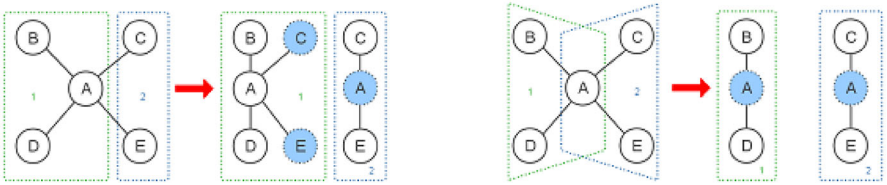

Fig. 1 EC (left side) and VC (right side) methods for partitioning a sample graph [97]

In vertex-centric systems which apply the VC partitioning, after each vertex data is updated, all of its copied versions should be updated. Therefore, the number of copied versions of vertices indicates the external communication overhead among different machines [97]. The difference of these two models is shown in Fig. 1.

There exists a one-to-one mapping between the VC and EC partitioning. The number of copied versions of the vertices in a VC partitioning is less than its corresponding EC (with creating ghost vertices for each of the cut edges). According to the available surveys, the VC partitioning outperforms the EC one for natural graphs such as social networks and the web structure. These graphs have a lot of low degree and a few high degree vertices [47,71,72,93]. Existence of a high degree vertex in a machine may disturb load balancing [30]. The VC model can cut high degree vertices and distribute them on different machines [2,45,48,82].

In [10], the outcomes of these two models are compared statistically in terms of efficiency of vertex-centric systems. It is found that the expected communication costs of systems with both models are equal without combining messages while the communication cost of the system with the VC is less than the other with combining messages.

All of these obtained results are based on power-law model for natural graphs but degree distribution of vertices in natural graphs is not exactly compatible with power-law model [38]. Therefore, using better models for comparing the effects of different partitioning approaches is necessary.

By removing high degree vertices from a natural graph, it turns to some isolated dense clusters that can be partitioned efficiently by EC model because high degree vertices increase inter-dependability among different clusters of nodes. Hence, one can first remove these vertices; then partition the generated sub-graphs by an EC partitioning method, and finally adds the removed vertices to the yield partitions [49].

Both models of graph partitioning problem are NP-hard [12]. There are various approximate algorithms for graph partitioning. The most popular approaches are multi-level algorithms like Metis. These algorithms have three phases. First, the initial graph is coarsened iteratively by clustering vertices and replacing each cluster with one weighted vertex. Next, the coarsened graph is partitioned. Finally, the yield partitions map back and refine iteratively. These algorithms produce near optimal partitions. Another approach of graph partitioning is spectral partitioning that bisects the graph based on the different metrics such as Fiedler vectors, conductance and modularity iteratively. All of these approaches are investigated in [13].

Traditional approaches of graph partitioning need random access to graph vertices. They have poor locality and cannot execute in distributed manner. Therefore, they can

not be applied for big graph partitioning [67]. In this paper, recent approaches of big graph partitioning for distribution of graphs in vertex-centric systems are discussed.

3 Vertex-centric systems

Google introduced the Pregel [52] system for distributed graph computations in 2010. This system has master-slave architecture. The slaves make computations and the master coordinates them. This system is based on the Bulk Synchronous Parallel (BSP) [87] model. Computations are made in successive supersteps in parallel. After each superstep, the machines are synchronized with one another. Graph algorithms are defined with a user-defined function named "compute". Each slave machine executes this function for each one of its active vertices in each superstep. Executing this function for a vertex might update its data, send messages to its adjacent vertices or inactivate the vertex. Each inactive vertex is activated again when receives a message. Computations continue until all vertices become inactive [52].

The graph structure is in the main memory of slave machines during the computations. The graph is partitioned in the EC manner. Communications among vertices are made by message sending and push method. The graph topology may be changed during the execution [52]. Giraph [17] is an open-source implementation of this system. This system has been implemented on Hadoop [92] (the most common ecosystem of the big data computations) as a map-only task.It applies the infrastructure of such as HDFS [9].

Definition of "combine" function is optional for the user. If this function is defined, messages that should be sent to a vertex from the vertices in another partition are combined and only one message is sent to it. This reduces the communication overhead.

Pregel+[101] uses two other techniques for reducing the number of messages. This system keeps one mirror of each high degree vertices v in any slaves, in which some vs neighbors exist. When, such a vertex wants to send a message m to all outgoing neighbors, it sends m to its mirrors. Then, each mirror forward m to v's neighbors which exist in the same slave as the mirror. In addition to mirroring, this system introduces a request-response paradigm instead of vertex to vertex communication channel which Pregel uses.

Pregel uses push mode for communication. This means that each vertex can send messages to outgoing neighbors and cannot request the last updates from incoming neighbors. Vertices update their values based on the messages they have received. Gemini system [111] applies message passing communication in both push and pull modes. The authors of Gemini have argued that if the number of active vertices is high during the execution of an algorithm, the pull mode for updating vertices' values has better performance than push mode.

An asynchronous vertex-centric graph system, named GraphLab, is introduced in [51]. In this model, each machine executes a user-defined function recursively and without synchronization with other machines for each active vertex. A locking mechanism prevents concurrent execution of this function on adjacent vertices. This system, like Giraph, applies EC partitioning.

Gather-Apply-Scatter (GAS) model for vertex-centric computations is introduced in PowerGraph. In this model, each recursion of computation for a vertex v is executed in three phases: Firstly, the value of adjacent vertices or edges are read and aggregated in gather phase. Secondly, aggregated values are applied for updating the value of v in apply phase. Finally, the values of adjacent vertices and edges are updated in scatter phase [30].

PowerGraph, like GraphLab, applies shared-memory for communication. It applies the VC model for graph partitioning. This system can be applied in synchronous or asynchronous manner. In synchronous version, a synchronization step should be observed after each one of the gather, apply and scatter phases of each superstep. Consequently, the synchronization overhead of PowerGraph is more than that of Giraph. Its asynchronous version is similar to GraphLab [30].

The VC partitioning in PowerGraph may cause the adjacent edges of low degree vertices to become distributed in different machines, which can increase the communication overhead. The PowerLyra introduced in [15] applies a hybrid cut graph partitioning which places all incoming edges of each low degree vertices in a machine. Therefore, its gather phase can be executed in this machine locally, which leads to a decrease in the communication overhead [15].

GraphLab, PowerGraph and PowerLyra do not use message sending for communication. Each vertex can reach the values of adjacent edges or vertices with the assistant of shared memory. Shared memory approach has lower performance due to the lack of message batching for external communication among workers [35,94].

Distributed locking used in asynchronous systems has high overhead that can decrease their performance. Checking the termination of algorithms is difficult in asynchronous systems. They are not appropriate for graph mutation and multi phase algorithms. On the other hand, synchronous systems, based on BSP, have high communication overhead in global synchronization phase. Fast workers should wait for the slowest ones to reach global barriers. Messages, sent in a superstep, are not considered until the next superstep [35]

The Barrierless Asynchronous Parallel (BAP) model [35] allows workers to consider messages of the next super step as they arrive. Updating vertices' data based on the most recent messages causes faster convergence and shorter computation time. This model, instead of using global barriers, uses local barriers in workers to decrease communication overhead. It allows users to define global barriers after each separate phase of the algorithm or for graph mutations.

All mentioned systems are general and can be used for any graph computation. Tux2 [95] system is introduced for graph computations in machine learning application. General systems usually consider homogeneous set of vertices but graphs in machine learning application often have different types of vertices. For example, in a recommender system, two types of vertices represent sets of users and items. Tux2 supports bipartite graphs with different data structure for each type of vertex set.

Synchronous general systems has strict synchronization step but machine learning applications are usually robust against some inconsistencies. Tux2 applies the Stale Synchronous Parallel (SSP) model [21]. This model has a stale parameter that indicates how far ahead any task can progress from the slowest task.

Tux2 introduces MEGA computing model. The model has four user-defined functions. Exchange function, called for each edge, can change the value of that edge and its adjacent vertices. Apply function synchronized the value of each vertex by its replicas. Global sync function conducts shared computations and updates shared values among partitions. Finally, Mini-Batch function indicates the sequence of execution of other functions in each round.

Chronos [36] is a graph computation system for temporal graph mining applications. These applications should consider several snapshots of a graph in a time period. Chronos uses temporal locality for storing snapshots in memory. That is, several version of a vertex are placed close to each other. Computations on different snapshots are done simultaneously. Therefore, messages, sent by different versions of a vertex, can be grouped and sent altogether.

Mentioned systems store total graph structure in the main memory of the slave machines during the execution of an algorithm. Chaos [69] is a distributed graph computing system that stores graph structure in the secondary memory. Each slave has some mini partitions. It sequentially loads a mini partition in its main memory and performs its computations.

Pregelix [11] uses vertex-centric API but stores vertices, edges and messages like relations in a relational database. It does computation by performing several SQL queries on the relations. Pregelix can store and retrieve data in or out of memory efficiently.

GraphMat [81] also presents vertex-centric API but converts computations to adjacency matrix multiplication. In this way, existing hardware for efficient matrix multiplication, such as GPU, can be used for graph computations.

Implementation of graph operations by vertex-centric API is hard for non-programmer users. Gradoop [42] is a graph system that provides several popular graph operators. Users can define their graph analytical programs using a declarative domain specific language named Grala. Gradoop supports heterogeneous graph data model. Different properties of these systems are summarized in Table 1.

4 Stream-based partitioning methods

In stream-based methods of graph partitioning, vertices or edges of graph are examined in a stream successively. Decisions regarding the assignment of each edge or vertex to partitions are made online as soon as it appears in the stream. This decision is based on the locations of previous edges or vertices and does not change later. It is assumed that there is no information about vertices or edges that will arrive in the flow of the stream [80].

Stream-based methods have some advantages: (i) their time complexity is linear based on the number of vertices and edges [80], (ii) provided that the entire structure of graph is not available at the beginning, they can be applied, (iii) graph partitioning in these methods is incremental (hence, if the graph structure is changed and new vertices or edges are added, there is no need for repartitioning from scratch), and (iv) they can be implemented in parallel [30,70].

Table 1 The properties of some vertex-centric systems

Systems	Partitioning	Communication	Synchronization	Application	Memory
Pregel [52]	Edge cut	Message passing (push)	Sync (BSP)	Global	In memory
GraphLab [51]	Edge cut	Shared memory	Async	Machine learning	In memory
PowerGraph [30]	Vertex cut	Shared memory	Sync (BSP) and Async	Global	In memory
PowerLyra [15]	Hybrid cut	Shared memory	Async	Global	In memory
Gemini [111]	Vertex cut	Message passing (push–pull)	Sync (BSP)	Global	In memory
Chaos [69]	Vertex cut	Shared memory	Sync (BSP)	Global	Out of memory
Tux2 [95]	Vertex cut	Shared memory	Sync (SSP)	Machine learning	In memory
Pregelix [11]	Edge cut	Message passing (push)	Sync (BSP)	Global	In or Out of memory
GraphMat [81]	Edge cut	Message passing (push)	Sync (BSP)	Global	In memory
Chronos [36]	Edge cut	Shared memory	Async	Temporal graph mining	In memory
Giraph Unchained [35]	Edge cut	Message passing (push)	Async (BAP)	Global	In memory

They have also two disadvantages. Firstly, the resulting graph partitioning depends on the order of the vertices or edges in the stream [84]. This order can be random or based on DFS or BFS graph traversal [20] from a custom vertex. Secondly, they do not guarantee a good approximate solution of graph partitioning. For example, provided that the graph is sparse and the vertex stream is of a random order, it is expected that the first $o(\sqrt{n})$ vertices in the stream will have no common edge. Hence, their placement, regardless of the heuristic, will be random.

4.1 Stream-based edge cut graph partitioning

Some heuristics of stream-based edge cut graph partitioning is introduced and compared in [80]. The Random heuristic produces balanced partitions with no concern on the number of cut edges. The expected ratio of cut edges in this method is $1 - \frac{1}{k}$, where k is the number of partitions. Pregel and GraphLab systems use this heuristic for graph partitioning by default.

The Linear Deterministic Greedy (LDG) heuristic, for placement of each new vertex v of the stream, ranks the partitions based on the number of their v's neighbors and inserts v in the partition with the highest rank. A weight proportional to the free capacity of each partition is multiplied by its rank for obtaining balanced partitions. The triangular heuristic ranks each partition based the number of triangles where v is one of their vertices in the partition [80]. The heuristic is based on local clustering coefficient factor [91]. The LDG heuristic obtains the best results among all mentioned heuristics [79].

Real world graphs are often collected by crawlers. Then, adjacent vertices are stored and visited close to each other in the stream. The authors of [111] apply a simple chunk-based graph partitioning based on this assumption. This method divides the stream of vertices into k contiguous vertex chunks.

A new definition of k-way balanced graph partitioning is discussed in [85]. This definition has no hard constraints on balancing the size of partitions. The balance factor and the number of cut edges are applied for evaluating the quality of partitioning. Each of them can be reduced in favor of the other. Therefore, the size of the obtained partitions may be unbalanced.

The quality of a set of partitions is evaluated through the sum of an intra-partition function (for obtaining balanced partitions) and an inter-partition function (for reducing the number of cut edges). Each new vertex in stream is placed in a partition that maximizes quality [85].

By considering different functions for intra- and inter- partition factors, different heuristics can be obtained. Intra-partition factor is usually considered as the number of vertices' neighbors placed in their partitions and inter-partition factor is usually considered as the sum of the results of a convex function on the partitions' sizes multiplied by -1. Provided that partitions' sizes are equal, this sum will be minimized and intra-partition function is maximized. This method is named Fennel. The expected ratio of cut edges in this method is $\frac{\log k}{k}$ [85].

Tsourakakis [84] proves that after arriving the first $\log^{6+\epsilon} n$ vertices of the stream, each new vertex has $\log^6 n$ neighbors among previous vertices with high probability. They present a semi-stream method for graph partitioning, which postpones the assignments of these vertices to partitions until the next vertices' arrival. For each next vertex v in the stream, the number of common v's neighbors with each previous vertex is calculated and the vertex with the most common neighbors is placed in a partition with v. This method, unlike Fennel and LDG methods, considers two-step neighbors for placing vertices. In this paper, it is argued that using two-step neighbors yields the best results among all other neighbors. These results are based on the random planted graphs with hidden partitions [8,19,59].

Echbarthi and Kheddouci [27] have applied Metis heuristic for stream-based partitioning. This method merges vertices of each partition together and builds k weighted vertices at each iteration. Then, it loads a portion of next vertices of the stream. It builds a weighted graph connecting these new vertices to the k weighted vertices. This weighted graph is partitioned with Metis algorithm to obtain new partitions. This process is performed until the end of the stream.

The methods that apply a stream-based partitioning recursively with considering logs of previous recursions, are called restream methods. Applying the partitioning log information improves the partitioning results. Two restream methods, based on LDG and Fennel heuristics, are mentioned by Nishimura and Ugander [64]. These methods calculate the number of neighbors of vertices in each partition and the load of each partition based on the most recent partition assignment, either from the previous or the current passes. The restream version of LDG does not guarantee convergence since its results depend on the order of vertices in the stream. This non-convergence is applicable in dynamic graphs because the partitioning results can be adapted to the graph changes. The restream version of Fennel is convergent. This convergence does not depend on the stream order. Hence, it is not applicable for dynamic graphs [64].

Another method for graph partitioning based on log information is presented in [99]. This method can be adapted for loading a graph into Pregel system. First, a hyper graph is constructed from the original graph based on the log information. Then, this hyper graph is partitioned with a stream-based method. The hyper graph has hyper edges among vertices that were in the same partition and active vertices that were accessed by a vertex in successive supersteps in addition to the edges of original graph. Consequently, the vertices that are adjacent to a hyper edge are assigned to a partition with high probability [99]. Although, this method is not stream-based, it applies stream-based methods. The main drawback of this method is the overhead for constructing the hyper graph.

The partial restream method [26] partitions the first loaded portion of graph with a multipath restream method. The rest of graph is partitioned with a common streaming heuristic. This method has lower runtime while the number of cut edges may be higher than that of the whole restream methods.

All of the mentioned methods try to construct balance partitions. They assume that all machines have the same computing powers. A stream-based partitioning method for non-uniform environment is presented in [100]. The features of all mentioned methods are compared in Table 2.

Table 2 The comparison of edge cut streaming heuristics

Methods	Architecture awareness	Log awareness	Neighbor	Iteration
Random [80]	✘	✘	One step	One iteration
LDG [80]	✘	✘	One step	One iteration
Fennel [85]	✘	✘	One step	One iteration
Restream [64]	✘	✓	One step	Multiple iterations
Partial-restream [26]	✘	✘	One step	Multiple iterations for a portion of graph
LogGP [99]	✘	✓	One step	Two iterations
Streaming metis [27]	✘	✘	One step	Multiple iterations for each window of stream
EGyPT [84]	✘	✘	Two steps	One iteration
Capacity-aware [100]	✓	✘	One step	One iteration

4.2 Stream-based vertex cut graph partitioning

The simplest manner of vertex cut graph partitioning is the random method that inserts an equal number of edges in different partitions. The yield partitions are balanced while the number of copied versions of vertices may be high. The expected number of vertex copies in this method is $\frac{k}{|V|}\sum_{\forall v \in V}(1-(1-\frac{1}{k})^{D(v)})$, where $D(v)$ is the degree of vertex v and k is the number of partitions [41].

The torus or Grid heuristic allows each vertex to copy only in a limited subset of partitions [41]. Each pair of the vertices should have at least one common partition in their subsets. Each new edge (u, v) in the stream is placed in a common partition of allowed subsets of v and u. The maximum of vertex copies in this method is $1.5\sqrt{(|v|)+1}$. This method is named two-dimensional because it arranges partitions in a two dimensional table and applies a hash function to determine allowed subsets of the vertices. The restriction of this method is that the number of partitions should be a power of two.

PowerGraph system applies a greedy heuristic for stream-based VC partitioning. It has four rules for assigning each new edge (v, u): (i) provided that v and u have already been copied and there are some partitions containing both, the edge is placed in one of these partitions, (ii) if v and u have already been copied but there is no partition containing both, the edge is placed in one of the partitions that have the vertex (among v and u) with fewer unseen edges, (iii) if one of v and u has already been copied, the edge is placed in one of the partitions that has its copies, and (iv) if neither v and u have already been copied, the edge is placed in the smallest partition [30].

PowerGraph does not guarantee the production of balanced partitions. For example, if the stream order is based on DFS graph traversal, all edges will be placed in one partition. The degrees of vertices should be estimated for applying the second rule. Graphbuilder [41] applies another rule instead of the second rule. This new rule places the edge in the smallest partition among the partitions that contain copies of vertices.

Fennel and LDG heuristics have been adapted for VC partitioning in [10]. These methods rank partitions and select the partition with the highest rank for placing each new edge (u, v). In the Fennel-based method, the intra-partition rank of each partition indicates by the number of vertices among v and u that have been previously assigned to this partition. Its intra-partition rank is calculated like that of the common Fennel method.

Since, the above methods do not consider vertex degrees, the adjacent edges of low degree vertices may be placed in different partitions. It can generate many vertex copies in natural graphs because they have many low-degree vertices. Petroni et al. [66] present a method named HDRF for vertex cut partitioning based on this argument. This method subtracts the normalized degree of a vertex from the ranks of partitions containing its copies. Therefore, the ranks of the partitions that have copies of the lower degree vertex would be higher than that of the partitions with the higher degree ones in equal conditions. Applying this method needs the estimation of the vertices' degrees.

Natural graphs usually have high skewness in incoming degrees of their vertices, whereas outgoing degrees do not have this property. Therefore, incoming degrees can be a good estimation of the total degree. The method presented in [97] estimates and applies only the incoming degrees of vertices to decrease time complexity.

A hybrid cut partitioning method is presented in [15]. This method applies edge cut for low degree vertices and vertex cut for high degree ones. Consequently, the number of copies of low degree vertices would be low. This is not a stream-based method but it similarly assigns edges to partitions only once.

When vertex-centric systems execute algorithms like page rank, only copies of a vertex with some outgoing edges should be informed of its data updates. A method that reduces the number of copies with outgoing edges is presented in [105]. This method changes the PowerGraph [30] rules so that each new edge (u, v) is inserted in a partition containing a copy of u with only outgoing edges or a copy of v with only incoming edges.

The communication cost among different machines may be different in vertex-centric systems. Mayer and Tariq et al. [56] propose a method based on the PowerGraph rules. This method inserts different replica of vertices in machines with low communication cost.

Stream-based methods can be used for inserting new edges in dynamic graphs during the execution of an algorithm in vertex-centric systems. In this situation, the rank of the partitions can be calculated dynamically based on the speed and the free capacity of the machines. A method based on the LDG heuristic is presented in [14] which multiplies the rank of each partition by the weight proportional to the speed of the machine containing it in the last superstep.

Adwise [57] is a semi-stream vertex cut partitioning. This method considers a window of w next edges of the stream and computes the score of each edge related to each partition. Then, it assigns the best edge to the best partition and refills the window again so that it contains w edges. The size of w is adaptive and can be changed during the execution based on the acceptable delay of partitioning algorithm.

The method ranks a partition p related to an edge (u, v) like HDRF except that it also considers an additional clustering factor. This factor represents the number of u

Table 3 The comparison of vertex cut streaming heuristics

Methods	Balance partitions	Reduce communication	Heterogeneous communication	Heterogeneous capacity	Graph
Random [30]	✓	✘	✘	✘	–
Torus [41]	✘	✓	✘	✘	–
PowerGraph [30]	✘	✓	✘	✘	–
Balanced VC [10]	✓	✓	✘	✘	–
HDRF [66]	✓	✓	✘	✘	Power-law
s-PowerGraph [97]	✓	✓	✘	✘	Power-law
ginger [15]	✓	✓	✘	✘	Power-law
EDAP [105]	✘	✓	✘	✘	–
GrapH [56]	✘	✓	✓	✘	–
Imbalance cluster [14]	✓	✓	✘	✓	–

and vs neighbors in p. Considering this factor, the yield partitions have higher locality. It decreases the number of replicas of vertices. All of these heuristics are tabulated in Table 3.

4.3 Parallel and distributed execution of stream-based methods

There are three main approaches for parallel and distributed implementation of stream-based methods. In the coordinate approach, there is a shared table accessible to all machines. This table stores partitioning information. Each machine partitions a sub-stream by using the information of this table and updates it [30]. Reading and updating shared table for each vertex or edge has high communication overhead. Instead, each sub-partitioner can collect a number of next vertices or edges of its sub-stream in a window with a certain size. Then, it can pull the needed information for partitioning them in one batch from the shared table [70].

In the oblivious approach, each machine partitions a sub-stream independently without any information of other sub-streams. The oblivious approach has lower time complexity while the number of cut edges or copied versions of vertices with this method may be higher than that of the coordinated approach [30]. The authors of [57] suggest that if there are r worker machines, each machine divides its sub-stream into $\frac{k}{r}$ partitions, where k is the number of desired partition in total graph. Based on experimental results, this approach reduces the number of cut edges or vertices replicas.

A different approach is introduced in [77] that assigns each partition to a worker machine. There is a coordinator that receives stream of vertices and broadcasts the information of each vertex v to workers. Each worker calculates the rank of its partition related to v and sends this rank to the coordinator. Then, coordinator assigns v to the partition with the highest rank. This method can be implemented asynchronously in common distributed systems, such as spark [103] and hadoop [92].

Table 4 The complexity of streaming heuristics

Methods	Time	Space	State
Random EC [80]	o(n)	o(1)	None
LDG [80]	o(kn+m)	o(n)	Partition assignments
Fennel [85]	o(kn+m)	o(n)	Partition assignments
Restream [64]	o(r*(kn+m))	(n)	Partition assignments
EGyPT [84]	o(n*polylog(n))	o(n)	Partition assignments
Random VC [30]	o(m)	o(1)	None
Torus [41]	o(m+n)	o(k)	Size of partitions
PowerGraph [30]	o(km+n)	o(n)	Partition assignments
HDRF [66]	o(km+n)	o(n)	Partition assignments & degree of vertices
s-PowerGraph [97]	o(km+n)	o(m+n)	Partition assignments & degree of vertices
ginger [15]	o(kn+m)	o(n)	Partition assignments & degree of vertices

4.4 performance of stream-based methods

Several experimental evaluations have been done on the efficiency of stream-based graph partitioning approaches in vertex-centric systems [1,34,89]. Common existing graph systems such as GraphLab, PowerGraph, PowerLyra support stream-based methods as main approach for partitioning and loading graphs.

The results of experiments show that a good choice of partitioning strategy depends on different factors: (1) the properties of the graph such as its degree distribution; (2) the type of computation and algorithms; (3) the properties of hardware such as the number of workers or heterogeneity in communication or computation; (4) details of the implementation of the graph system [34,89].

According to the results of [1], algorithms that store state of the partitioning (for example previous partition assignments or partial degree of vertices) deliver higher quality partitions according to the number of cut edges or vertices than hash-based method. These heuristics are not suitable for online applications that work with unbounded stream of vertices or edges due to the need for global state.

Another experiment indicates that vertex cut heuristics which store state, performs better for graphs with high-diameter and low-degree distribution, such as road networks (according to the total partitioning and execution time). Torus algorithm does not store state. It incurs higher replication factor and lower execution time when compared to the stateful heuristics. Therefore, Torus approach is appropriate for short jobs and heuristic strategies are suitable for long-running jobs on heavy-tailed and power-law graphs. Hybrid PowerLyra algorithm is more efficient for natural applications, which gather from one direction and scatter in the other [89]. The time and space complexity of these heuristics are listed in Table 4.[1]

[1] $n = |V|$ & $m = |E|$ & $r =$ *number of iterations*.

5 Distributed partitioning methods

Vertex-centric systems can be used for distributed execution of graph partitioning algorithms, though the execution time of graph partitioning algorithms, like other algorithms, depends on the initial partitioning and distribution of the graph. This is a chicken and egg problem according to [54]. However, a random or simple stream-based partitioning method can be adapted for initial graph loading and better partitioning result with lower cut edges or vertex copies can be obtained with a distributed partitioning algorithm.

Most of the distributed methods of graph partitioning are based on label propagation method [31,50,110]. This method considers k labels that represent partitions. First, each vertex selects a random label and sends it to its neighbors. Next, each vertex ranks the labels by considering neighbors' labels and selects the label with the highest rank for itself and sends it to its neighbors again. It is repeated until the labels of vertices stop changing and the algorithm converges.

The advantage of label propagation method is that if the graph structure is changed in dynamic graphs, there is no need to repartition the graph from scratch. In this situation, exchanging and updating the labels can continue until the convergence is obtained again. Moreover, if there is a need to increase the number of partitions, a similar method can be used. Assume that the current number of partitions is k and it is desired to add j new partitions. In this case, each vertex should change its label to one of the new labels with the probability $\frac{j}{(k+j)}$. Then, label propagation algorithm should be restarted to adapt the partitioning to new assignments [55].

Various methods differ in ranking the labels. For example, in [55], each vertex v ranks each label l based on the number of v's neighbors with label l. A penalty function related to each label is subtracted from its rank to balance the size of partitions. Its value is equal to the sum of the degrees of vertices with this label. Therefore, if the number of edges in a partition is high, the value of its penalty function will be high as well. The main drawback of this approach is that if the size of a partition is small, many vertices may choose it concurrently and its size may become very large. To decrease the effect of this drawback, each vertex changes its label to candidate label only with the probability $p < 1$. This probability should be inversely proportional with the partition size.

Sometimes there is a need to consider different constraints on the minimum and maximum sizes of the partitions. A method considering these constraints was performed by Ugander and Backstrom [86] where the number of vertices that can move from each partition to other partitions in each round is determined by solving a linear programming problem. The time complexity of solving this problem depends on the number of partitions.

The graph partitioning problem, as a multi-objective and multi-constrained problem, is defined by Slota and Madduri et al. [78]. It presents a three-phase method for solving this problem. All of them are based on the label propagation method. The output of each phase is the initial partitioning of the next phase. Some of these constraints and objectives are considered in each phase. This method has better results as compared to a one-phase methods, which considers all constraints and objectives together.

The method presented in [67] applies another approach instead of using the penalty function to balance the partitions. In this method, the random initial partitions should be balanced. Each vertex in each recursion selects one of its neighbors. Then, these two vertices exchange their labels. By exchanging labels, the partitions remain balanced. Labels of two adjacent vertices will be exchanged if the sum of ranks of new labels has the most increments related to their previous labels. This algorithm applies the simulated annealing method [40] to avoid falling into local optimum. If there is a need for more constraints on the size of the partitions, one can consider them on initial label allocation. The VC version of this method is presented in [68]. Here, concurrent exchange of labels may increase the number of cut edges. To overcome this drawback, a method is presented by Chen and Li [16] which only allows two partitions to exchange their vertices in each time.

Mofrad and Melhem et al. [61] apply the reinforcement learning for distributed graph partitioning. Their algorithm, named Revolver, assigns an agent to each vertex. Agents iteratively select partitions for vertices based on their probability distribution. The partition assignments are evaluated by label propagation. Agents refine their probability distributions based on the feedbacks. Selections and feedbacks continue to get convergence.

Edges are assigned to partitions through amount of funds in the VC method presented in [33]. In this method, first, a random vertex and an equal amount of funds are assigned to each partition. Next, each partition, in each round, makes an offer to obtain an edge based on the number of its adjacent vertices. Each edge is sold to the partition that makes the largest offer. The smallest partition gets more unit of fund at the end of each round. This algorithm has three phases where the first two phases can be executed in a distributed manner but the third one should be executed in a centralized manner. The time complexity of this phase depends only on the number of partitions [32,33].

There are some distributed approaches for executing the phases of multi-level graph partitioning methods [60,90,104]. A label propagation method for coarsening phase of these algorithms is presented in [104]. This method is based on vertex-centric computational model. Another label propagation method is introduced in [60] that can be used for both coarsening and refinement phases. It is based on sub-graph centric model.

Aydin and Bateni et al. [5] present a distributed partitioning method based on the Map-Reduce [46] computing model on Hadoop [92]. The result of this method can be used for initial distribution of big graphs in vertex-centric systems. This method first maps vertices of graph in a line such that vertices with the most affinity are close to each other. This ordering can be obtained from an agglomerative hierarchical clustering of the graph. Next, it improves ordering by swapping vertices. Finally, it cuts this line to K balanced partitions and does some local optimization in cut points of partitions to improve the quality of partitioning. The properties of all mentioned methods are summarized in Table 5.

Table 5 The comparison of distributed graph partitioning methods

Methods	Computing model	Approach	Graph changes	Partition number change	Multi constraint	VC/EC
Spinner [55]	TLAV	Label propagation	✓	✓	✘	EC
Blp [86]	TLAV	Label propagation	✓	✓	✓	EC
Pulp [78]	TLAV	Label propagation	✓	✓	✓	EC
Ja-be-ja [67]	TLAV	Label exchange	✓	✓	✓	EC
Ja-be-ja-vc [68]	TLAV	Label exchange	✓	✓	✓	VC
BS [16]	TLAV	Label exchange	✓	✓	✓	EC
Deep [33]	TLAV	Currency distribution	✘	✘	✘	VC
Ml-blp [60]	TLAV	Label propagation	✘	✘	✘	EC
Linear embedding [5]	MR	Linear embeding	✓	✓	✓	EC
Revolver [61]	TLAV	Reinforcement learning	✘	✘	✘	VC

6 Dynamic graph partitioning methods

Most big graphs, such as social networks' graphs, have dynamic structure and some vertices or edges may be added to or removed from them as time goes on. Moreover, during the execution of some algorithms, such as graph traversal algorithms, all vertices or edges are not active in all supersteps. Hence, even a good initial graph partitioning cannot guarantee load balancing of machines.

To avoid load imbalance caused by algorithmic skewness, the initial partitioning should distribute vertices that are activated in the same superstep among machines. This is possible when the runtime characteristic of algorithm is predictable. Graph traversal algorithms, such as BFS, have this feature [109]. If this is impossible, one can use a dynamic repartitioning method to guarantee load balancing.

Methods of dynamic graph repartitioning in vertex-centric systems monitor loads and communications of machines during the execution and based on this information, select some vertices for migration among machines. The differences of them are in the manner of selecting candidate vertices for migration, choosing destination partition for them and exchanging them. All of these methods can be applied in synchronous vertex-centric systems. There exists no dynamic graph partitioning method for asynchronous systems to the knowledge of the authors here.

XDGP method [88] divides the available capacity of each partition by the number of other partitions at the end of each superstep in order to determine their quotas for sending vertices to it. Each vertex selects the partition containing most of its neighbors as its candidate partition. If the quota does not exceed the required amount, it will migrate to this partition with probability $p < 1$. One of the drawbacks is that, the adjacent vertices may exchange their partitions at the same time so that the number of cut edges increases. By considering the probability p, this drawback is reduced.

The method seeks to decrease the number of cut edges for decreasing the communication overhead while all cut edges may not be used for conducting communications during the execution of an algorithm. Hence, it is better that, instead of cut edges, the communications among different machines are considered for decision about vertex migration.

GPS method [73] considers this factor. Each pair of the machines exchanges the equal number of their vertices that has the most communication with each other in each superstep. This method considers only external communication of vertices to make decision on vertex migrations.

Xpregel method [7] considers both external and internal communications. This method allows only a machine to move its vertices to other machine in each superstep. It prevents adjacent vertices from moving at the same time. Master machine, at each superstep, selects the machine with the most vertices or edges as a candidate for moving vertices. This machine monitors the incoming and outgoing messages of its vertices and based on this information, it selects target partitions for moving them. There is no limitation on the number of vertices that can move. Thus, the result partitions may not be balanced.

Putting the same number of vertices in different partitions does not guarantee load balancing, because some vertices may not be active in some supersteps and also active vertices do not have the same computational time. Catchw method [76] considers

computational load instead of the number of vertices. Here, each machine monitors its computational load in each superstep and sends it to other machines. Computational load is calculated by the sum of the numbers of its active vertices and edges. Active edges are considered because the running time of each active vertex usually is linear to the number of messages it receives or sends via these edges. The main drawback of this method is that if all machines has balanced computational load, no vertex will be selected for moving even if external communication cost is very high.

In Mizan [43], each machine monitors the number of input messages, output messages to other machines, and total running time for its active vertices in each superstep and sends this information to other machines. Each machine calculates the correlation between each of these factors and time and selects the factor with the highest correlation as the objective factor for moving vertices. Here, each overutilized machine (based on the objective factor) is paired with an underutilized one and moves some of its vertices to the former. Pairing the machines prevents overutilization after vertex migrations. One drawback is that the selection of machines for pairing is made without considering communications among them and this leads to an increase in the amount of communications.

Above methods make decisions on vertex migration based on the information of previous superstep. However, load and communication patterns of the next superstep may be different from the previous superstep. The LogGP method [99] estimates the communication and computation costs of the next superstep and moves vertices based on this estimation. Only vertices need external communication in the next superstep, that their two-step neighbors are not in the same partition with them . Hence, the number of them on each machine is considered as the estimation of its communication load.

All of the mentioned methods assume uniform network communication costs among partitions. Most of the existing parallel and distributed architectures have heterogeneous communication costs. Aragon [106] is an architecture aware repartitioning method that refines the partitioning of graph according to non-uniform network communication costs. For each pair of partitions, Aragon moves vertices with the most gain between them. Gain is defined by reduction in communication costs between partitions related to the underlying network. The vertex migration overhead is considered in computing gain. Aragon only considers communication costs for repartitioning and does not consider balance factor. Partitions are not adapted to graph changes during the execution of algorithms. Paragon [108] is the parallel version of Aragon.

Planar [107] is a dynamic repartitioner that is triggered when there is imbalance among partitions. It has three phases. In the first phase it computes the gain of moving each vertex to other partitions similar to Aragon and selects the partition with the most gain as its destination. In the second phase it uses an architecture aware quota based allocation to balance partitions. These two phases move vertices logically. Physical migration of them is implemented by a migration service like Zoltan [22] in the third phase.

Moving a vertex in Pregel like systems can be implemented by moving its adjacency list, data and incoming messages in current superstep. For decreasing the overhead of this process, The GPS [73] system uses the differed migration method. Moving a vertex in this method is implemented in two supersteps. In the first superstep, all

Table 6 The comparison of dynamic methods

Methods	Migration metric	Balance method	Implementation	EC/VC	Architecture aware
Xdgp [88]	Num of cut edges	Using quotas	Differed migration with broadcast	EC	✘
Gps [73]	Num of output messages	Swapping vertices	Differed migration with broadcast	EC	✘
Xpregel [7]	Num of output and input messages	Repartitioning the largest partition	Differed migration with broadcast	EC	✘
Catchw [76]	Computation time	Using quotas	Immediate migration with look up table [83]	EC	✘
Mizan [43]	Num of output and input messages and computation time	Pairing small and large partitions	Differed migration with Distributed hash table [6]	EC	✘
LogGP [99]	Computation time of next super step	Repartitioning large partitions	Immediate migration with Look up table [83]	EC	✘
Aragon [106]	Communication and migration costs	pairing partitions	Centralized Immediate migration	EC	✓
Planar [107]	Communication and migration costs	Using quotas	Using a migration service like Zoltan [22]	EC	✓
GrapH [56]	Communication and migration costs	Pairing partitions	Immediate migration with broadcast	VC	✘

adjacent vertices are informed about this moving and update their adjacency lists but Vertex does not move. In the next superstep, the new data of the vertex and also its adjacency list are moved to new machines. There is no need to move messages between machines using this method.

The essential issue with vertex migration is how to inform other vertices about this migration. One way is to send messages to adjacent vertices and inform them. Communication overhead of this method is high. Moreover, non-adjacent vertices cannot communicate with the vertex any more. Applying a distributed hash table for positioning vertices can overcome this issue since only this table should be updated when a vertex moves [43,73]. A dynamic method is presented in [56] that moves edges, instead of vertices, among partitions It chooses two random partitions after each superstep and moves candidate edges between them. It applies a locking mechanism on the vertices adjacent to candidate edges to avoid inconsistency. These dynamic methods are tabulated in Table 6.

7 Different methods with other approaches

A parallel multi-level partitioning method is presented in [3]. PEs in this method has access to all graph structure stored in a shared memory. Each PE applies a size constraint label propagation clustering for coarsening phases on a bucket of vertices. Then,

each PE executes KAHIP [75] partitioning algorithm on the coarsened graph independently. The best partitioning result is selected in this phase. Both size constraint label propagation and k-way multi try local search [75] methods are applied for refinement phase.

A serial algorithm for graph partitioning is presented in [65] that has low execution time and consequently can be used for big graph partitioning. This method has three phases. It applies the balanced sized modularity based graph clustering method in the first phases to produce more clusters than the number of desired partitions. Then, it applies a greedy first fit technique for picketing problem to merge partitions until the number of clusters becomes the number of desired partitions in the second phases. Finally, the final edge cut partitioning is converted to the vertex cut by producing a vertex replica for each cut edge.

The authors of [24] propose a method that constructs a super graph with a small number of vertices and edges by clustering the original graph. It partitions the super graph based on the properties of the application that should be executed. These properties are estimated by the execution of compute function with random messages. This partitioning is used for distribution of graph in vertex-centric system and does not change dynamically during the execution of algorithm. So, this method does not monitor the communication and computation loads of machines dynamically. This method has low computational overhead because the size of the super graph is smaller than the original graph. One issue about this method is how to estimate the properties of the compute method.

Dynamic assignment of partitions to virtual machines in cloud environment is considered by Dong and Zhang et al. [23]. Their method moves partitions among virtual machines after each superstep in order to decrease the number of active virtual machines. This decreases the total cost that should be paid for active virtual machines.

A method that is based on vertex replication instead of vertex migration is presented by Yang and Yan et al. [102]. Here, the communications and computations of different partitions are monitored. If some vertices in different partitions have heavy communications with each other, a new partition is created and placed in a low load machine. These vertices are replicated and placed in this new partition. This method is applied to response queries in limited region of graphs.

During the execution of some algorithms in vertex-centric systems, a small number of vertices are sometimes active in successive supersteps. If the number of active vertices is less than a threshold, they can be moved to the master machine for reducing the synchronization overhead [74].

8 Conclusion

One of the applications of big graph partitioning is the balanced distribution of a graph in a cluster for executing graph algorithms. The vertex-centric model is the most popular paradigm of graph systems which has been introduced in recent years.

In this paper, the recent approaches of big graph partitioning are considered and compared with each other and their advantages and disadvantages are mentioned.

All of these approaches can be used for partitioning and distributing big graphs in vertex-centric systems in a sense that the performance of these systems is improved.

Recent partitioning approaches for vertex-centric systems are categorized into three classes in this paper. (i) Stream-based methods that consider each vertex or edges once. (ii) Distributed methods that partition graphs in a distributed manner. They have high scalability and can produce higher quality partitioning results than that of stream-based methods. (iii) Dynamic methods that change partitions dynamically during the execution of other algorithms. They change the initial partitions by considering the communication and computation loads of machines.

The efficiency of graph partitioning approaches in a graph System depends on variable factors: (i) the properties of the graph, such as its degree distribution; (ii) the hardware specification; (iii) the implementation of the graph system; (iv) the communication and computation pattern of application. There is no graph partitioning algorithm which performs well in all situations.

There are some suggestions for selecting an appropriate partitioning algorithm:

- Stream-based methods are good choices for loading the graph because they do not need the whole graph in the memory for partitioning. The stream of vertices or edges can load and partition in a pipeline simultaneously. Therefore, popular graph systems implement streaming methods as default partitioning approaches.
- State-full stream-based heuristics, which store and apply the partitioning history or partial degree of vertices, output better partitions as compared to stateless hash-based methods, but they are not applicable for unbounded streams.
- Distributed graph partitioning methods need separate partition phase. They are suitable when the graph is partitioned once and calculations are done several times.
- Among distributed approaches, those approaches that are based on label propagation, like spinner, BLP and Pulp, are simple and flexible. The yield partitions can be adapted to the changes in graph structure or the number of partitions.
- Dynamic partitioning approaches have the best outputs in terms of load balancing and reducing communication costs because they consider the runtime characteristic of algorithms and hardware heterogeneity.
- The overhead of migration of vertices or edges among partitions is high. Therefore, dynamic algorithms, which consider migration costs for decision on the movement of vertices or edges, are more functional. GrapH and Planar are two examples of these methods.
- The VC partitioning produces more balance partition for natural graphs with power-law degree distribution, as compared with the EC one.

There are a number of interesting directions for future works:

- Higher length walks: Most existing streaming and distributed graph partitioning approaches consider one-step neighbors. Extending existing graph partitioning heuristics with considering higher length walks may improve the quality of partitioning.
- Using other clustering approaches: Most existing distributed graph partitioning algorithms are based on label propagation method. Label propagation method is basically a graph clustering algorithm that has been extended for graph partitioning. The result of this method depends on the initial label assignments. Using

other approaches of graph clustering like random walk based may cause better partitioning results.
- Architecture awareness: There is no distributed approach that has architecture awareness. All of the mentioned methods consider uniform communication costs among different partitions. As mentioned in Sect. 6, architecture awareness is essential.
- Dynamic vertex cut graph partitioning: Most existing approaches of dynamic repartitioning are based on edge cut model. Whereas, many vertex-centric distributed frameworks use vertex cut model.
- Dynamic partitioning for asynchronous systems: Existing methods of dynamic repartitioning are for synchronous systems. Migration of vertices or edges for asynchronous systems has more complexity than synchronous ones because it needs concurrency control methods such as locking to avoid inconsistency. There is a need for asynchronous dynamic methods.
- Algorithmic awareness: Even an optimal graph partition cannot guarantee the performance of vertex-centric systems for executing all algorithms, because graph algorithms have different communication and computation patterns. As a suggestion for future works, graph algorithms can be categorized based on communication and computation patterns and special graph partitioning methods can be presented for each category.

References

1. Abbas, Z., Kalavri, V., Carbone, P., Vlassov, V.: Streaming graph partitioning: an experimental study. Proc. VLDB Endow. **11**(11), 1590–1603 (2018). https://doi.org/10.14778/3236187.3236208
2. Abou-Rjeili, A., Karypis, G.: Multilevel algorithms for partitioning power-law graphs. In: Proceedings 20th IEEE International Parallel Distributed Processing Symposium (2006). https://doi.org/10.1109/IPDPS.2006.1639360
3. Akhremtsev, Y., Sanders, P., Schulz, C.: High-quality shared-memory graph partitioning (2017). arXiv:1710.08231
4. Aslam, S.: Twitter by the numbers: stats, demographics and fun facts (2018). https://www.omnicoreagency.com/twitter-statistics/
5. Aydin, K., Bateni, M., Mirrokni, V.: Distributed balanced partitioning via linear embedding. In: Proceedings of the Ninth ACM International Conference on Web Search and Data Mining, pp. 387–396. ACM, New York (2016)
6. Balakrishnan, H., Kaashoek, M.F., Karger, D., Morris, R., Stoica, I.: Looking up data in p2p systems. Commun. ACM **46**(2), 43–48 (2003). https://doi.org/10.1145/606272.606299
7. Bao, N.T., Suzumura, T.: Towards highly scalable pregel-based graph processing platform with x10. In: Proceedings of the 22Nd International Conference on World Wide Web (WWW '13) Companion, pp. 501–508. ACM, New York (2013). https://doi.org/10.1145/2487788.2487984
8. Bollobs, B.: Random Graphs. Springer, New York (1998)
9. Borthakur, D.: HDFS Architecture Guide. Apache Hadoop Project (2008)
10. Bourse, F., Lelarge, M., Vojnovic, M.: Balanced graph edge partition. In: Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '14), pp. 1456–1465. ACM, New York (2014). https://doi.org/10.1145/2623330.2623660
11. Bu, Y., Borkar, V., Jia, J., Carey, M.J., Condie, T.: Pregelix: big(ger) graph analytics on a dataflow engine. Proc. VLDB Endow. **8**(2), 161–172 (2014). https://doi.org/10.14778/2735471.2735477
12. Bui, T.N., Jones, C.: Finding good approximate vertex and edge partitions is np-hard. Inf. Process. Lett. **42**(3), 153–159 (1992). https://doi.org/10.1016/0020-0190(92)90140-Q

13. Buluç, A., Meyerhenke, H., Safro, I., Sanders, P., Schulz, C.: Recent advances in graph partitioning. In: Kliemann, L., Sanders, P. (eds.) Algorithm Engineering. LNCS, pp. 117–158. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-49487-6-4
14. Cao, Y., Rao, R.: A streaming graph partitioning approach on imbalance cluster. In: 2016 18th International Conference on Advanced Communication Technology (ICACT), pp. 360–364 (2016). https://doi.org/10.1109/ICACT.2016.7423392
15. Chen, R., Shi, J., Chen, Y., Chen, H.: Powerlyra: differentiated graph computation and partitioning on skewed graphs. In: Proceedings of the Tenth European Conference on Computer Systems (EuroSys '15), pp. 1:1–1:15. ACM, New York (2015). https://doi.org/10.1145/2741948.2741970
16. Chen, T., Li, B.: A distributed graph partitioning algorithm for processing large graphs. In: 2016 IEEE Symposium on Service-Oriented System Engineering (SOSE), pp. 53–59 (2016)
17. Ching, A.: Giraph: production-grade graph processing infrastructure for trillion edge graphs. In: ATPESC, vol. 14 (2014)
18. Ching, A., Edunov, S., Kabiljo, M., Logothetis, D., Muthukrishnan, S.: One trillion edges: graph processing at Facebook-scale. Proc. VLDB Endow. **8**(12), 1804–1815 (2015). https://doi.org/10.14778/2824032.2824077
19. Condon, A., Karp, R.M.: Algorithms for graph partitioning on the planted partition model. Random Struct. Algorithms **18**(2), 116–140 (2001)
20. Cormen, T.H.: Introduction to algorithms. MIT, Cambridge (2009)
21. Cui, H., Cipar, J., Ho, Q., Kim, J.K., Lee, S., Kumar, A., Wei, J., Dai, W., Ganger, G.R., Gibbons, P.B., et al.: Exploiting bounded staleness to speed up big data analytics. In: USENIX Annual Technical Conference, pp. 37–48 (2014)
22. Devine, K., Boman, E., Heaphy, R., Hendrickson, B., Vaughan, C.: Zoltan data management services for parallel dynamic applications. Comput. Sci. Eng. **4**(2), 90–97 (2002)
23. Dindokar, R., Simmhan, Y.: Elastic partition placement for non-stationary graph algorithms. In: 2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid), pp. 90–93 (2016). https://doi.org/10.1109/CCGrid.2016.97
24. Dong, F., Zhang, J., Luo, J., Shen, D., Jin, J.: Enabling application-aware flexible graph partition mechanism for parallel graph processing systems. Concurr. Comput. Pract. Exp. **29**(6), e3849 (2017). https://doi.org/10.1002/cpe.3849, e3849 cpe.3849
25. Donnelly, G.: 75 super-useful Facebook statistics for 2018 (2018). https://www.wordstream.com/blog/ws/2017/11/07/facebook-statistics
26. Echbarthi, G., Kheddouci, H.: Fractional greedy and partial restreaming partitioning: New methods for massive graph partitioning. In: 2014 IEEE International Conference on Big Data (Big Data), pp. 25–32 (2014). https://doi.org/10.1109/BigData.2014.7004368
27. Echbarthi, G., Kheddouci, H.: Streaming metis partitioning. In: 2016 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining (ASONAM), pp. 17–24 (2016). https://doi.org/10.1109/ASONAM.2016.7752208
28. Elsner, U.: Graph partitioning—a survey. Technical Report SFB393/97-27 (1997)
29. Fjllstrm, P.O.: Algorithms for Graph Partitioning: A Survey, vol. 3. Linkping University Electronic Press, Linkping (1998)
30. Gonzalez, J.E., Low, Y., Gu, H., Bickson, D., Guestrin, C.: Powergraph: distributed graph-parallel computation on natural graphs. In: Proceedings of 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI), vol. 12, pp. 17–30 (2012)
31. Gregory, S.: Finding overlapping communities in networks by label propagation. New J. Phys. **12**(10), 103,018 (2010)
32. Guerrieri, A., Montresor, A.: Distributed edge partitioning for graph processing (2014). arXiv:1403.6270
33. Guerrieri, A., Montresor, A.: DFEP: Distributed Funding-Based Edge Partitioning, Springer, Berlin, pp. 346–358 (2015)
34. Guo, Y., Hong, S., Chafi, H., Iosup, A., Epema, D.: Modeling, analysis, and experimental comparison of streaming graph-partitioning policies. J. Parallel Distrib. Comput. **108**, 106–121. https://doi.org/10.1016/j.jpdc.2016.02.003. Special Issue on Scalable Computing Systems for Big Data Applications (2017)
35. Han, M., Daudjee, K.: Giraph unchained: barrierless asynchronous parallel execution in pregel-like graph processing systems. Proc. VLDB Endow. **8**(9), 950–961 (2015). https://doi.org/10.14778/2777598.2777604

36. Han, W., Miao, Y., Li, K., Wu, M., Yang, F., Zhou, L., Prabhakaran, V., Chen, W., Chen, E.: Chronos: A graph engine for temporal graph analysis. In: Proceedings of the Ninth European Conference on Computer Systems (EuroSys '14), pp. 1:1–1:14. ACM, New York (2014). https://doi.org/10.1145/2592798.2592799
37. Hendawi, A.M., Bao, J., Mokbel, M.F.: iRoad: a framework for scalable predictive query processing on road networks. Proc. VLDB Endow. **6**(12), 1262–1265 (2013). https://doi.org/10.14778/2536274.2536291
38. Hoque, I., Gupta, I.: Lfgraph: Simple and fast distributed graph analytics. In: Proceedings of the First ACM SIGOPS Conference on Timely Results in Operating Systems (TRIOS '13), pp. 9:1–9:17. ACM, New York (2013). https://doi.org/10.1145/2524211.2524218
39. Hu, K., Zeng, H.J.W.W.G.: Partitioning big graph with respect to arbitrary proportions in a streaming manner. Future Gen. Comput. Syst. **80**, 1–11 (2018). https://doi.org/10.1016/j.future.2017.06.027
40. Hwang, C.R.: Simulated annealing: theory and applications. Acta Appl. Math. **12**(1), 108–111 (1988). https://doi.org/10.1007/BF00047572
41. Jain, N., Liao, G., Willke, T.L.: Graphbuilder: Scalable graph etl framework. In: First International Workshop on Graph Data Management Experiences and Systems (GRADES '13), pp. 4:1–4:6. ACM, New York (2013). https://doi.org/10.1145/2484425.2484429
42. Junghanns, M., Kiessling, M., Teichmann, N., Gomez, K., Petermann, A., Rahm, E.: Declarative and distributed graph analytics with gradoop. Proc. VLDB Endow. **11**(12), 2006–2009 (2018). https://doi.org/10.14778/3229863.3236246
43. Khayyat, Z., Awara, K., Alonazi, A., Jamjoom, H., Williams, D., Kalnis, P.: Mizan: A system for dynamic load balancing in large-scale graph processing. In: Proceedings of the 8th ACM European Conference on Computer Systems (EuroSys '13), pp. 169–182. ACM, New York (2013). https://doi.org/10.1145/2465351.2465369
44. Kim, G.H., Trimi, S., Chung, J.H.: Big-data applications in the government sector. Commun. ACM **57**(3), 78–85 (2014). https://doi.org/10.1145/2500873
45. Lang, K.: Finding good nearly balanced cuts in power law graphs. Technical Report YRL-2004-036, Yahoo! Research Labs (2004)
46. Lee, K.H., Lee, Y.J., Choi, H., Chung, Y.D., Moon, B.: Parallel data processing with MapReduce: a survey. SIGMOD Rec. **40**(4), 11–20 (2012). https://doi.org/10.1145/2094114.2094118
47. Leskovec, J., Kleinberg, J., Faloutsos, C.: Graph evolution: densification and shrinking diameters. ACM Trans. Knowl. Discov. Data **1**(1), 2 (2007). https://doi.org/10.1145/1217299.1217301
48. Leskovec, J., Lang, K.J., Dasgupta, A., Mahoney, M.W.: Community structure in large networks: natural cluster sizes and the absence of large well-defined clusters. Internet Math. **6**(1), 29–123 (2009)
49. Lim, Y., Lee, W.J., Choi, H.J., Kang, U.: MTP: discovering high quality partitions in real world graphs. World Wide Web, pp. 1–24 (2016)
50. Liu, X., Murata, T.: Advanced modularity-specialized label propagation algorithm for detecting communities in networks. Physica A **389**(7), 1493–1500 (2010). https://doi.org/10.1016/j.physa.2009.12.019
51. Low, Y., Bickson, D., Gonzalez, J., Guestrin, C., Kyrola, A., Hellerstein, J.M.: Distributed GraphLab: a framework for machine learning and data mining in the cloud. Proc. VLDB Endow. **5**(8), 716–727 (2012). https://doi.org/10.14778/2212351.2212354
52. Malewicz, G., Austern, M.H., Bik, A.J., Dehnert, J.C., Horn, I., Leiser, N., Czajkowski, G.: Pregel: A system for large-scale graph processing. In: Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data (SIGMOD '10), pp. 135–146. ACM, New York (2010). https://doi.org/10.1145/1807167.1807184
53. Manyika, J., Chui, M., Brown, B., Bughin, J., Dobbs, R., Roxburgh, C., Byers, A.H.: Big data: the next frontier for innovation, competition, and productivity 2011, vol. 5(33), p. 222 (2015). http://www.mckinsey.com/Insights/MGI/Research/Technology_and_Innovation/Big_data_The_next_frontier_for_innovation
54. Margo, D., Seltzer, M.: A scalable distributed graph partitioner. Proc. VLDB Endow. **8**(12), 1478–1489 (2015). https://doi.org/10.14778/2824032.2824046
55. Martella, C., Logothetis, D., Loukas, A., Siganos, G.: Spinner: Scalable graph partitioning in the cloud (2014). arXiv:1404.3861

56. Mayer, C., Tariq, M.A., Li, C., Rothermel, K.: Graph: heterogeneity-aware graph computation with adaptive partitioning. In: 2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS), pp. 118–128 (2016). https://doi.org/10.1109/ICDCS.2016.92
57. Mayer, C., Mayer, R., Tariq, M.A., Geppert, H., Laich, L., Rieger, L., Rothermel, K.: Adwise: adaptive window-based streaming edge partitioning for high-speed graph processing (2017). arXiv preprint. arXiv:1712.08367
58. McAfee, A., Brynjolfsson, E., Davenport, T.H., Patil, D., Barton, D.: Big data's: the management revolution. Harv. Bus. Rev. **90**(10), 61–67 (2012)
59. McSherry, F.: Spectral partitioning of random graphs. In: Proceedings of 42nd IEEE Symposium on Foundations of Computer Science, pp. 529–537 (2001)
60. Meyerhenke, H., Sanders, P., Schulz, C.: Parallel graph partitioning for complex networks. IEEE Trans. Parallel Distrib. Syst. **PP**(99), 1–1 (2017)
61. Mofrad, M.H., Melhem, R., Hammoud, M.: Revolver: vertex-centric graph partitioning using reinforcement learning. In: 2018 IEEE 11th International Conference on Cloud Computing (CLOUD), pp. 818–821 (2018). https://doi.org/10.1109/CLOUD.2018.00111
62. Moreira, O., Popp, M., Schulz, C.: Graph partitioning with acyclicity constraints (2017). arXiv:1704.00705
63. Nirmala, G., Dinakaran, K.: Analysis of protein database for semantic similarity using map reduce; a survey. In: Proceedings of IEEE International Conference on Computer Communication and Systems (ICCCS14), pp. 046–050 (2014). https://doi.org/10.1109/ICCCS.2014.7068166
64. Nishimura, J., Ugander, J.: Restreaming graph partitioning: simple versatile algorithms for advanced balancing. In: Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '13), pp. 1106–1114. ACM, New York (2013). https://doi.org/10.1145/2487575.2487696
65. Onizuka, M., Fujimori, T., Shiokawa, H.: Graph partitioning for distributed graph processing. Data Sci. Eng. **2**(1), 94–105 (2017). https://doi.org/10.1007/s41019-017-0034-4
66. Petroni, F., Querzoni, L., Daudjee, K., Kamali, S., Iacoboni, G.: HDRF: Stream-based partitioning for power-law graphs. In: Proceedings of the 24th ACM International on Conference on Information and Knowledge Management (CIKM '15), pp. 243–252. ACM, New York (2015). https://doi.org/10.1145/2806416.2806424
67. Rahimian, F., Payberah, A.H., Girdzijauskas, S., Jelasity, M., Haridi, S.: JA-BE-JA: A distributed algorithm for balanced graph partitioning. In: 2013 IEEE 7th International Conference on Self-Adaptive and Self-Organizing Systems, pp. 51–60 (2013)
68. Rahimian, F., Payberah, A.H., Girdzijauskas, S., Haridi S.: Distributed vertex-cut partitioning. In: Distributed Applications and Interoperable Systems. Springer, Heidelberg, pp. 186–200 (2014)
69. Roy, A., Bindschaedler, L., Malicevic, J., Zwaenepoel, W.: Chaos: Scale-out graph processing from secondary storage. In: Proceedings of the 25th Symposium on Operating Systems Principles (SOSP '15), pp. 410–424. ACM, New York (2015). https://doi.org/10.1145/2815400.2815408
70. Sajjad, H.P., Payberah, A.H., Rahimian, F., Vlassov, V., Haridi, S.: Boosting vertex-cut partitioning for streaming graphs. In: 2016 IEEE International Congress on Big Data (BigData Congress), pp. 1–8 (2016). https://doi.org/10.1109/BigDataCongress.2016.10
71. Sala, A., Cao, L., Wilson, C., Zablit, R., Zheng, H., Zhao, B.Y.: Measurement-calibrated graph models for social network experiments. In: Proceedings of the 19th International Conference on World Wide Web (WWW '10), pp. 861–870. ACM, New York (2010). https://doi.org/10.1145/1772690.1772778
72. Sala, A., Zhao, X., Wilson, C., Zheng, H., Zhao, B.Y.: Sharing graphs using differentially private graph models. In: Proceedings of the 2011 ACM SIGCOMM Conference on Internet Measurement Conference (IMC '11), pp. 81–98. ACM, New York (2011).https://doi.org/10.1145/2068816.2068825
73. Salihoglu, S., Widom, J.: GPS: a graph processing system. In: Proceedings of the 25th International Conference on Scientific and Statistical Database Management (SSDBM), pp. 22:1–22:12. ACM, New York (2013). https://doi.org/10.1145/2484838.2484843
74. Salihoglu, S., Widom, J.: Optimizing graph algorithms on pregel-like systems. Proc. VLDB Endow. **7**(7), 577–588 (2014). https://doi.org/10.14778/2732286.2732294
75. Sanders, P., Schulz, C.: Engineering multilevel graph partitioning algorithms. In: Algorithms—ESA 2011. Springer, Berlin, pp. 469–480 (2011)
76. Shang, Z., Yu, J.X.: Catch the wind: Graph workload balancing on cloud. In: 2013 IEEE 29th International Conference on Data Engineering (ICDE), pp. 553–564 (2013). https://doi.org/10.1109/ICDE.2013.6544855

77. Shi, Z., Li, J., Guo, P., Li, S., Feng, D., Su, Y.: Partitioning dynamic graph asynchronously with distributed fennel. Future Gen. Comput. Syst. **71**, 32–42 (2017). https://doi.org/10.1016/j.future.2017.01.014
78. Slota, G.M., Madduri, K., Rajamanickam, S.: PuLP: scalable multi-objective multi-constraint partitioning for small-world networks. In: 2014 IEEE International Conference on Big Data (Big Data), pp. 481–490 (2014). https://doi.org/10.1109/BigData.2014.7004265
79. Stanton, I.: Streaming balanced graph partitioning algorithms for random graphs. In: Proceedings of the Twenty-Fifth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '14), pp. 1287–1301 (2014). https://doi.org/10.1137/1.9781611973402.95
80. Stanton, I., Kliot, G.: Streaming graph partitioning for large distributed graphs. In: Proceedings of the 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '12), pp. 1222–1230. ACM, New York (2012). https://doi.org/10.1145/2339530.2339722
81. Sundaram, N., Satish, N., Patwary, M.M.A., Dulloor, S.R., Anderson, M.J., Vadlamudi, S.G., Das, D., Dubey, P.: GraphMat: high performance graph analytics made productive. Proc. VLDB Endow. **8**(11), 1214–1225 (2015). https://doi.org/10.14778/2809974.2809983
82. Suri, S., Vassilvitskii, S.: Counting triangles and the curse of the last reducer. In: Proceedings of the 20th International Conference on World Wide Web (WWW '11), pp. 607–614. ACM, New York (2011). https://doi.org/10.1145/1963405.1963491
83. Tatarowicz, A.L., Curino, C., Jones, E.P.C., Madden, S.: Lookup tables: fine-grained partitioning for distributed databases. In: 2012 IEEE 28th International Conference on Data Engineering, pp. 102–113. (2012). https://doi.org/10.1109/ICDE.2012.26
84. Tsourakakis, C.: Streaming graph partitioning in the planted partition model. In: Proceedings of the 2015 ACM on Conference on Online Social Networks (COSN '15), pp. 27–35. ACM, New York (2015). https://doi.org/10.1145/2817946.2817950
85. Tsourakakis, C., Gkantsidis, C., Radunovic, B., Vojnovic, M.: Fennel: streaming graph partitioning for massive scale graphs. In: Proceedings of the 7th ACM International Conference on Web Search and Data Mining (WSDM '14), pp. 333–342. ACM, New York (2014). https://doi.org/10.1145/2556195.2556213
86. Ugander, J., Backstrom, L.: Balanced label propagation for partitioning massive graphs. In: Proceedings of the Sixth ACM International Conference on Web Search and Data Mining (WSDM '13), pp. 507–516. ACM, New York (2013). https://doi.org/10.1145/2433396.2433461
87. Valiant, L.G.: A bridging model for parallel computation. Commun. ACM **33**(8), 103–111 (1990). https://doi.org/10.1145/79173.79181
88. Vaquero, L.M., Cuadrado, F., Logothetis, D., Martella, C.: xDGP: a dynamic graph processing system with adaptive partitioning (2013). arXiv:1309.1049
89. Verma, S., Leslie, L.M., Shin, Y., Gupta, I.: An experimental comparison of partitioning strategies in distributed graph processing. Proc. VLDB Endow. **10**(5), 493–504 (2017). https://doi.org/10.14778/3055540.3055543
90. Wang, L., Xiao, Y., Shao, B., Wang, H.: How to partition a billion-node graph. In: 2014 IEEE 30th International Conference on Data Engineering, pp. 568–579 (2014). https://doi.org/10.1109/ICDE.2014.6816682
91. Watts, D.J., Strogatz, S.H.: Collective dynamics of 'small-world' networks. Nature **393**(6684), 440–442 (1998). https://doi.org/10.1038/30918
92. White, T.: Hadoop: The Definitive Guide. O'Reilly Media, Boston (2012)
93. Wilson, C., Boe, B., Sala, A., Puttaswamy, K.P., Zhao, B.Y.: User interactions in social networks and their implications. In: Proceedings of the 4th ACM European Conference on Computer Systems (EuroSys '09), pp. 205–218. ACM, New York (2009). https://doi.org/10.1145/1519065.1519089
94. Wu, M., Yang, F., Xue, J., Xiao, W., Miao, Y., Wei, L., Lin, H., Dai, Y., Zhou, L.: GraM: Scaling graph computation to the trillions. In: Proceedings of the Sixth ACM Symposium on Cloud Computing (SoCC '15), pp. 408–421. ACM, New York (2015). https://doi.org/10.1145/2806777.2806849
95. Xiao, W., Xue, J., Miao, Y., Li, Z., Chen, C., Wu, M., Li, W., Zhou, L.: Tux2: Distributed graph computation for machine learning. In: Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17), pp. 669–682 (2017)
96. Xiao-Shu, W., Yao, X., Huan, L.: Cloud computing oriented retrieval technology based on big data. In: 2015 IEEE Seventh International Conference on Measuring Technology and Mechatronics Automation, pp. 275–278 (2015). https://doi.org/10.1109/ICMTMA.2015.73

97. Xie, C., Li, W.J., Zhang, Z.: S-PowerGraph: streaming graph partitioning for natural graphs by vertex-cut (2015). arXiv:1511.02586
98. Xin, R.S., Gonzalez, J.E., Franklin, M.J., Stoica, I.: GraphX: a resilient distributed graph system on spark. In: First International Workshop on Graph Data Management Experiences and Systems (GRADES '13), pp. 2:1–2:6. ACM, New York (2013). https://doi.org/10.1145/2484425.2484427
99. Xu, N., Chen, L., Cui, B.: LogGP: a log-based dynamic graph partitioning method. Proc. VLDB Endow. **7**(14), 1917–1928 (2014)
100. Xu, N., Cui, B., Chen, L., Huang, Z., Shao, Y.: Heterogeneous environment aware streaming graph partitioning. IEEE Trans. Knowl. Data Eng. **27**(6), 1560–1572 (2015)
101. Yan, D., Cheng, J., Lu, Y., Ng, W.: Effective techniques for message reduction and load balancing in distributed graph computation. In: Proceedings of the 24th International Conference on World Wide Web (WWW '15), pp. 1307–1317. International World Wide Web Conferences Steering Committee, Republic and Canton of Geneva, Switzerland (2015). https://doi.org/10.1145/2736277.2741096
102. Yang, S., Yan, X., Zong, B., Khan, A.: Towards effective partition management for large graphs. In: Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data (SIGMOD '12), pp. 517–528. ACM, New York (2012). https://doi.org/10.1145/2213836.2213895
103. Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., Franklin, M.J., Shenker, S., Stoica, I.: Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. In: Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation (NSDI'12), pp. 2–2. USENIX Association, Berkeley (2012)
104. Zeng, Z., Wu, B., Wang, H.: A parallel graph partitioning algorithm to speed up the large-scale distributed graph mining. In: Proceedings of the 1st International Workshop on Big Data, Streams and Heterogeneous Source Mining: Algorithms, Systems, Programming Models and Applications (BigMine '12), pp. 61–68. ACM, New York (2012). https://doi.org/10.1145/2351316.2351325
105. Zhao, Y., Yoshigoe, K., Xie, M., Zhou, S., Seker, R., Bian, J.: LightGraph: lighten communication in distributed graph-parallel processing. In: 2014 IEEE International Congress on Big Data, pp. 717–724 (2014). https://doi.org/10.1109/BigData.Congress.2014.106
106. Zheng, A., Labrinidis, A., Chrysanthis, P.K.: Architecture-aware graph repartitioning for data-intensive scientific computing. In: 2014 IEEE International Conference on Big Data (Big Data), pp. 78–85 (2014). doi:https://doi.org/10.1109/BigData.2014.7004375
107. Zheng, A., Labrinidis, A., Chrysanthis, P.K.: Planar: Parallel lightweight architecture-aware adaptive graph repartitioning. In: 2016 IEEE 32nd International Conference on Data Engineering (ICDE), pp. 121–132 (2016a). https://doi.org/10.1109/ICDE.2016.7498234
108. Zheng, A., Labrinidis, A., Pisciuneri, P.H., Chrysanthis, P.K., Givi, P.: Paragon: parallel architecture-aware graph partition refinement algorithm. In: EDBT, pp. 365–376 (2016b)
109. Zheng, A., Labrinidis, A., Faloutsos, C.: Skew-resistant graph partitioning. In: 2017 IEEE 33rd International Conference on Data Engineering (ICDE), pp. 151–154 (2017). https://doi.org/10.1109/ICDE.2017.62
110. Zhu, X., Ghahramani, Z.: Learning from labeled and unlabeled data with label propagation. Tech. Rep., Citeseer (2002)
111. Zhu, X., Chen, W., Zheng, W., Ma, X.: Gemini: A computation-centric distributed graph processing system. In: Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16), pp. 301–316 (2016)