Distributed and Parallel Databases
https://doi.org/10.1007/s10619-024-07442-8

PECC: parallel expansion based on clustering coefficient for efficient graph partitioning

Chengcheng Shi[1,2] · Zhenping Xie[1,2]

Accepted: 26 May 2024

Abstract
In the pursuit of graph processing performance, graph partitioning, as a crucial preprocessing step, has been widely concerned. Based on an in-depth analysis of Neighbor Expansion (NE) graph partitioning algorithm, we propose Parallel Expansion based on Clustering Coefficient (PECC). Firstly, to address the partition disturbance caused by internal structural changes during the process of vertex neighborhood expansion in the traditional NE algorithm, we perform a formal redefinition of the vertex state during the partitioning process and introduce the concept of clustering coefficient. Then, PECC uses the clustering coefficient as a metric to measure the closeness between vertices and potential partitions. Based on this metric, a novel parallel partitioning strategy in the distributed environment is proposed. This strategy consists of two core steps: the expansion process and the allocation process. Through two steps, PECC can effectively improve the operating efficiency of programs and significantly reduce the partitioning time. In addition, to ensure data consistency during parallel expansion, we adopt a distributed locking engine to solve concurrency management problems. Our evaluations on large real-world graphs show that in many cases, PECC achieves a balance between partitioning quality and computational efficiency. Finally, we show that PECC integrated on GraphX outperforms the built-in native algorithms.

Keywords Graph partitioning · Distributed graph computing · Clustering coefficient · Distributed lock

✉ Zhenping Xie
xiezp@jiangnan.edu.cn

[1] School of Artificial Intelligence and Computer Science, Jiangnan University, Wuxi 214122, Jiangsu, China

[2] Jiangsu Key University Laboratory of Software and Media Technology Under Human-Computer Cooperation, Jiangnan University, Wuxi 214122, Jiangsu, China

1 Introduction

With the advent of the big data era, the concept of graph-structured data has attracted attention. From online social networks to protein structure networks, it is natural to model and represent data as a graph structure. Due to the massive scale of graph data, it is difficult to use traditional algorithms to analyse it. According to statistics, as of April 2023, Facebook had 2.989 billion monthly active users.[1] Traditional centralized algorithms often face challenges such as memory limitations, computing performance bottlenecks, and difficulty in achieving efficient parallelism when dealing with large and complex graph data. Therefore, based on the above challenges, graph partitioning, as a crucial preprocessing step, is important. Graph partitioning aims to partition a large-scale graph into several smaller subgraphs suitable for distributed environment processing, so as to alleviate the processing pressure of a single machine. For example, in bioinformatics, graph partitioning is used to partition massive protein networks into subgraphs, helping the identification of protein complexes or predicting protein functions [1]. Furthermore, distributed graph computing systems, including Pregel [2], PowerGraph [3], GraphScope [4, 5], and Plato [6, 7], integrate efficient graph partitioning algorithms to achieve effective storage, management, and parallel computation of graph data. As a result, graph partitioning algorithms are a focal point of our research.

Based on the different partitioning objects, graph partitioning algorithms can be classified into vertex partitioning algorithms and edge partitioning algorithms [8, 9]. However, in the real world, most graphs, such as social networks and web graphs, follow a power-law distribution, where most vertices have relatively few neighbors, while a few vertices have many neighbors [10]. In distributed graph systems with vertex partitioning algorithms, power-law distribution can lead to load imbalance between worker machines. Due to the linear relationship between computational complexity and the number of cut edges, the execution time required for each partition may vary significantly. In recent years, researchers have demonstrated that the edge partitioning model performs better on many large real-world graphs, which has sparked great interest in edge partitioning algorithms [11]. In our study, we focus on studying the edge partitioning model in the distributed environment.

Recent researches have demonstrated that NE has been identified as the state-of-the-art edge partitioning algorithm [12, 13]. NE is the first to provide such bound for edge partitioning algorithms on general graphs and applying this bound to random power-law graphs greatly improves the previous bounds of expected replication factor. It uses a heuristic approach to optimally select vertices from the adjacency set of each partition, followed by an expansion of the neighborhood. However, even high-quality edge partitioning algorithms face the following challenges [14]. Firstly, NE causes internal structural changes during partition expansion, thereby increasing the difficulty of partitioning. Secondly, NE is poor in terms of scalability. Finally, ensuring synchronization and collaboration between partitions in a distributed

[1] https://datareportal.com/social-media-users.

environment is another important consideration. To address these challenges, we propose PECC, a distributed graph partitioning algorithm. We introduce the clustering coefficient to measure the closeness between vertices and partitions. In a distributed environment, we expand partitions in parallel using heuristic strategies, which significantly improves program parallelism and reduces partitioning time. The main contributions of this work are as follows:

- We investigate the problem of the state-of-the-art distributed partitioning algorithms and present Parallel Expansion based on Clustering Coefficient (PECC), a distributed graph partitioning algorithm. In many cases, PECC outperforms the state-of-the-art graph partitioning algorithms.
- To better understand the internal workflow of the algorithm, we perform a formal classification of vertices based on states of vertices during partitioning process. We introduce the clustering coefficient to perform a quantitative analysis of the partition structure, thereby identifying groups of vertices with similar local connectivity properties.
- We propose a novel parallel partitioning strategy that divides the partitioning process into two stages: expansion and allocation. In addition, by introducing a distributed lock engine, we effectively address the concurrency management problem that arise during the parallel expansion process.
- We describe the implementation of the algorithm in detail. At the same time, we provide a comprehensive evaluation of the partitioning quality and running time in comparison with the state-of-the-art graph partitioning algorithms. Furthermore, we show that PECC integrated on GraphX outperforms the built-in native algorithms.

This paper is structured as follows. In the following Section we summarize the research work in recent years. In Sect. 3 we present a formulation of the problem. In Sect. 4 we outline the process of the NE algorithm. The proposed method is presented in Sect. 5. Section 6 provides a comprehensive evaluation. In Sect. 7 we conclude our study and look forward to future research.

2 Related work

This section first introduces notable distributed graph processing systems, and subsequently provides an overview of existing partitioning algorithms.

Distributed graph processing systems: Single machine graph processing systems are unable to cope with the immense graph data encountered today, both computationally and in terms of storage capacity. Therefore, several distributed large-scale graph processing systems have emerged, such as Pregel [2], PowerGraph [3], GraphX [15], GraphBuilder [16], and PowerLyra [17]. Pregel adopts a vertex-centered model and utilizes user-defined functions to aggregate information and calculate vertex values. It operates on the Bulk Synchronous Parallel (BSP) model, proceeding through a sequence of super-steps [18]. PowerGraph improves the partitioning quality for power-law distributed graphs by introducing vertex cutting. It replicates cut vertices across partitions, effectively tackling load imbalance from high-degree vertices. GraphX is an embedded graph computing framework based on Spark [19]. It effectively combines the computational models of Pregel and adopts the vertex-cut partitioning strategy. FBSGraph partitions graph data in a path-based manner on a distributed platform, thereby accelerating the propagation speed of vertex states [20]. ReGraph attempts to speed up convergence and improve scalability in distributed graph computing, utilizing techniques like repartitioning and graph compression [21].

Balanced graph partitioning (BGP): The goal of the *k*-way balanced graph partitioning is to partition a graph into *k* subgraphs while minimizing the number of cuts. The classical graph partitioning algorithms, such as METIS [22], Scotch [23] and their parallel variants [24, 25] adopt multi-level partitioning strategy, which have been shown to achieve high partitioning accuracy, and satisfy the load balancing conditions. Zhang et al. [12] proposed NE, a heuristic edge partitioning algorithm based on neighbor expansion, which effectively utilizes the structural information of the local graph. Based on NE, Mayer R proposed HEP, a hybrid partitioning model, which introduces graph pruning and lazy edge removal to fine-tune the trade-off between memory consumption and partitioning quality [13]. However, these methods have been proven to lack scalability with increasing graph size, especially when dealing with skewed degree distributions. This is where distributed graph computing comes in. XtraPuLP [26] is the state-of-the-art distributed-memory graph partitioner that extends PuLP [27] with a scalable label propagation technique. Spinner [28] is a large-scale graph partitioner, but it uses hash-based initial random allocation, resulting in poor quality of the final partition. Sheep [29] is a distributed graph partitioning algorithm, where transforms the input graph into the elimination tree via MapReduce [30] operations. In addition, it only works well on tree-like graphs. Strategic considerations regarding data placement and system architecture play a key role in improving the performance of algorithms when dealing with large-scale graphs in distributed environments. To support heterogeneous machines, Zeng et al. [31] proposed WindGP, a scalable framework of graph partitioning, which simplifies the metric and balances the computation cost according to the characteristics of graphs and machines. At the same time, they proposed best-first search scheme to generate partitions with high cohesion.

Streaming algorithms for graph partitioning (SAGP): The aforementioned partitioning methods are primarily designed for offline scenarios and are too resource-intensive to partition a large graph. Recently, there has been a growing interest in designing algorithms and frameworks that can handle massive graph data in a streaming manner. Several noteworthy methods are summarized in this context. A streaming vertex-cut partitioning algorithm, High Degree Replicated First (HDRF), was proposed by Petroni et al. [14]. It uses a greedy vertex-cut method that prioritizes replicating high-degree vertices to minimize unnecessary replication. Mayer et al. [32] proposed Adwise, which improves the partitioning quality by pre-buffering the local structural information of the graph and selecting the optimal edge for allocation. The partitioning strategy is based on HDRF and takes into consideration the clustering degree of the new edge neighborhood, which tends to cluster the local edges in the same partition. Kong et al.

[33] proposed CLUGP, CLUstering-based restreaming Graph Partitioning, which maps generated clusters to vertex-cut partitions by modeling the process by game theories. Hajidehi et al. [34] proposed CUTTANA, a streaming partitioner, a novel buffering technique that prevents premature assignment of vertices to partitions, allowing for more informed decisions based on a more complete view of the graph. It utilizes a scalable coarsening and refinement technique, which enhances the intermediate assignments. On the other hand, the conventional iterative computing approach is not suitable for addressing the challenges of incremental computation in dynamic graphs. To address this problem, Tang et al. [35] proposed IncGraph, an incremental computing model for dynamic graph. It operates in three stages: preprocessing, incrementing, and merging, focusing on iterative updates based on vertex changes and prior graph iterations.

3 Problem definition

Let $G = (V, E)$ be an undirected and unweighted graph with a set of vertices $V = \{v_1, v_2, \dots, v_n\}$ and a set of edges $E \subseteq V \times V$. The vertex set involved in E is defined as $V(E)$. The goal of the edge partitioning is to divide the set of edges E into P partitions, where $P > 1, P \in \mathbb{N}$. Each partition also has an edge set $E_k (k \in \{1, 2, \dots, P\})$, such that $\cup_{k=1,\dots P} E_k = E$ and $E_i \cap E_j = \emptyset$ where $(i, j \in \{1, 2, \dots, P\}, i \neq j)$.

For edge partitioning, the workload (amount of computation) of a partition is roughly linear in the number of edges located in that partition, and the replicas of the vertices incur communication for synchronization. So the graph partitioning problem considers three factors: (i) The number of vertex replications are minimized. (ii) The number of edges across partitions are balanced. (iii) The sum of the input graph loading time and the run-time required by partitioning algorithms is minimized. Let $V\left(E_k\right)$ be the set of partitions that each vertex is replicated. The number of vertex replication is normalized as follows:

$$\frac{1}{|V|} \sum_{k \in P} \left| V\left(E_k\right) \right| \tag{1}$$

By minimizing the replication factor, the amount of synchronization between the distributed machines is minimized. Therefore, the optimization problem of a balanced $|P|$-way edge partition of G is defined by Eq. (2).

$$\min \frac{1}{|V|} \sum_{k \in P} \left| V\left(E_k\right) \right|, \quad \text{s.t. } \max_{k \in P} \left| E_k \right| < \alpha \frac{|E|}{|P|} \tag{2}$$

where $\left|E_k\right|$ and $|P|$ are the size of the edge set of the partition and the number of partitions, respectively. And the imbalance factor, $\alpha \geq 1.0$, is a constant parameter.

At the same time, we also consider edge balance (EB) and vertex balance (VB) as metrics to measure the load balance among partitions, as given by Eqs. (3) and 4.

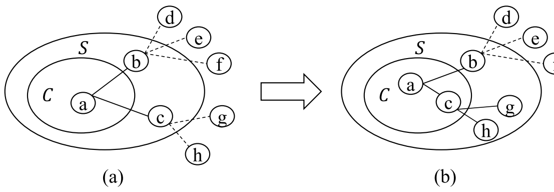

Fig. 1 An illustration of expansion step. **a** Is the initial partition state. **b** Is the state after partitioning expansion

$$EB := \frac{\max |E_k|}{\bar{E}}, \quad \text{s.t.} \bar{E} := \frac{\sum |E_k|}{|P|} \tag{3}$$

$$VB := \frac{\max |V(E_k)|}{\bar{V}}, \quad \text{s.t.} \bar{V} := \frac{\sum |V(E_k)|}{|P|} \tag{4}$$

where, the smaller the EB and VB are, the more balanced the load among partitions is.

4 Neighbor expansion

NE is a heuristic edge partitioning algorithm based on neighbor expansion, which effectively utilizes the structural information of the local graph. NE classifies vertices in each partition into two parts: the core set C and the boundary set S, where $C \subseteq S$. Vertices connected with external vertices in the partition belong to the boundary set S. The core idea of the algorithm's heuristic strategy is to minimize the degree of connection between the vertices in the boundary set S and the external vertices during each partition expansion, thereby minimizing the replication factor. In summary, the heuristic formula of NE is formally defined by Eq. (5):

$$x := \underset{v \in S \setminus C}{\arg\min} |N(v) \setminus S| \tag{5}$$

where, $|N(v) \setminus S|$ represents the number of vertex v connected to external vertices. The iterative nature of neighbor expansion is illustrated in Fig. 1. Initially, vertex a is randomly selected to be assigned to the current partition. Simultaneously, the set of the candidate vertices to be added to the partition is generated from the adjacent edges of the selected vertex. As shown in Fig. 1b, there is 1 vertex in the core set C, 3 vertices in the boundary set S. Next, we select the vertices from the boundary set S that have the least edges connected to external vertices to expand the neighborhood. Now, among the candidate set $S \setminus C = \{b, c\}$, vertex c is selected because $|N(c) \setminus S| = 2 < |N(b) \setminus S| = 3$. Then, vertex c is added to C, and its neighbors g and h are added to S. The adjacent edges (c, g) and (c, h) are allocated to this partition as shown in Fig. 1b. These steps iterate until the partition reaches the upper bound of load.

NE has some noticeable shortcomings when performing heuristic edge partitioning during neighbor expansion. Firstly, NE creates only one partition in each iteration and allocates only one edge to the partition during each expansion. This will lead to lower efficiency when dealing with large-scale graphs. Secondly, NE is highly sensitive to the structure of the graph. There are changes in the local structure, such as the emergence of fragmented or isolated subgraphs during the expansion process, can result in a decline in the final partitioning quality. Finally, in a distributed computing environment, ensuring synchronization and collaboration among partitions, as well as effectively handling the transmission of global information, are important factors to consider for the algorithm to run efficiently. In summary, based on a detailed analysis of NE, we propose a distributed graph partitioning algorithm called Parallel Expansion based on Clustering Coefficient (PECC).

5 Parallel expansion based on clustering coefficient

In order to address the aforementioned problems, this section provides the definition of vertex states and introduces a new heuristic strategy. Following that, we elaborate PECC framework.

5.1 Formalized definition

Definition 1 (*core vertex*). Given a vertex $v \in P_k$, with its set of adjacent vertices denoted as $N(v) = \{v_1, v_2, \cdots, v_n\}$, if it satisfies $C(P_k) = \left\{v | \forall (v, v_i) \in P_k, i \in [1, \cdots, n]\right\}$, then $C(P_k)$ is referred to as the core vertex set of partition P_k. As shown in Fig. 1, vertex a is considered as the core vertex and belongs to the core set.

Definition 2 (boundary vertex). Given a vertex $v \in P_k$, with its set of adjacent vertices denoted as $N(v) = \{v_1, v_2, \cdots, v_n\}$, if it satisfies$S(P_k) = \left\{v | \exists (v, v_i) \notin P_k, i \in [1, \cdots, n]\right\}$, then $S(P_k)$ is referred to as the boundary vertex set of partition P_k. As shown in Fig. 1, vertex b and c are considered as the boundary vertex.

Definition 3 (*external vertex*). Given a vertex v, if it satisfies $O(P_k) = \{v | v \notin P_k, k \in [1, \cdots, P]\}$, then $O(P_k)$ is referred to as the external vertex set of partition P_k. As shown in Fig. 1, vertex d,e,f,g and h are considered as the external vertex.

In the field of graph partitioning, what we pay more attention to is the degree of connection between vertices and the whole partition, rather than their local neighborhood. Therefore, we introduce the concept of clustering coefficient. Given the definitions of the core vertex set $C(P_k)$ and the boundary vertex set $S(P_k)$, the clustering coefficient is defined as follows.

Definition 4 (*clustering coefficient*). Given a vertex $v \in P_k$, with its set of adjacent vertices denoted as $N(v) = \{v_1, v_2, \cdots, v_n\}$ and the core vertex set denoted as $C(P_k) = \left\{v | \forall (v, v_i) \in P_k, i \in [1, \cdots, n]\right\}$. To represent the degree of closeness between vertex v and partition P_k, the cluster efficient is defined by Eq. (6):

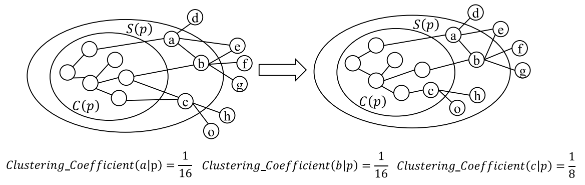
S(p)
C(p)
a
b
c
d
e
f
g
h
o
Clustering_Coefficient(a|p) = 1/16
Clustering_Coefficient(b|p) = 1/16
Clustering_Coefficient(c|p) = 1/8

Fig. 2 An illustration of heuristic strategy for selecting best exploration vertex

$$Cluster\_Coefficient\left(v|P_k\right) = \frac{N(v) \cap C\left(P_k\right)}{N(v)^2} \tag{6}$$

Based on clustering coefficient, we propose a new heuristic strategy to add unallocated edges to P_k without increasing the number of replication. The formula for selecting the best exploration vertex v_{best} is as follows:

$$v_{best} := \underset{v \in S(P_k) \setminus C(P_k)}{\arg\max}\ Cluster\_Coefficient(v|P_k) \tag{7}$$

This formula explains the decision-making process for selecting boundary vertices with higher *Clustering_Coefficient* as the best exploration vertex. Given a vertex v belonging to the boundary vertex and its adjacent vertex set $N(v)$, if there are only a few edges connected between vertex v and core vertex set $C(P_k)$ in the partition, then *Clustering_Coefficient* of vertex v will be close to 0. This reflects a relatively sparse connection environment in the partition. When selecting the best exploration vertex, the algorithm gives priority to boundary vertices with a higher *Clustering_Coefficient*, so as to ensure that the allocated edges help to improve the connection tightness of the whole partition and reduce the number of replication.

An example of selecting the best exploration vertex through Eq. (7) is shown in Fig. 2. During the expansion process, based on the principle of Eq. (7), we select the best exploration vertex from the set of candidate vertices $S(p) \setminus C(p) = \{a, b, c\}$ to strengthen the structural compactness of the partition p. The calculation results show that the score of each candidate vertex follows: $Clustering\_Coefficient(a|p) = \frac{1}{16} = Clustering\_Coefficient(b|p) = \frac{1}{16} < Clustering\_Coefficient(c|p) = \frac{1}{8}$, so the vertex c is selected as the best exploration vertex and add it to the core vertex set $C(p)$, and at the same time allocate the edge (c,h) and edge (c,o) connected with it into the current partition p. However, if it is partitioned based on Eq. (5), that is $|N(a) \setminus S(p)| = 2 = |N(c) \setminus S(p)| < |N(b) \setminus S(p)| = 3$. If it is only based on the indicator for the number of out-of-partition vertices, it is necessary to randomly choose one of the vertices a and c as the best exploration vertex. However, this strategy that only depends on the number of out-of-partition vertices has certain limitations. Although vertex a and vertex c have the same number of out-of-partition vertices in absolute terms, the structural tightness between vertex c and partition p is significantly better than that of vertex a, as shown in Fig. 2. This means that only relying on Eq. (5) may ignore the correlation and connectivity between vertices and the whole structure of the partition, thus leading to the failure of the selected exploration vertex to maximize the internal stability of the partition.

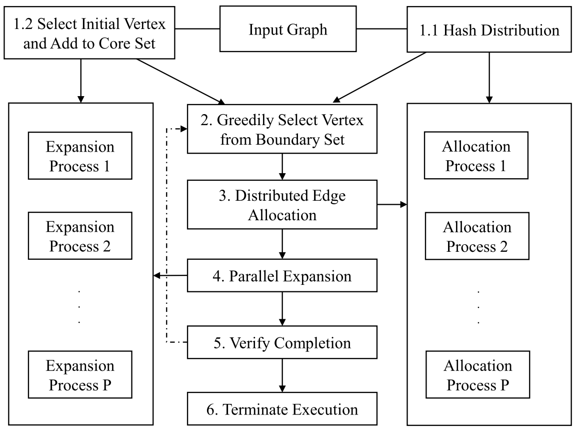

Fig. 3 The workflow of our PECC

In the following sections, we will explain in detail how to effectively utilize these theoretical analysis for optimizing graph partitioning process.

5.2 Parallel expansion

For the balanced $|P|$-way edge partitioning of $G = (V, E)$, the key consideration is to ensure that there are as many partitions as there are machines available. We assume that it is most efficient for each partition to be assigned to one machine [36]. In addition, the parallel expansion is classified as an offline algorithm, which requires full access to the whole input graph from the beginning. Therefore, it is essential to effectively initialize the deployment of the input graph in a distributed main memory system to achieve scalability and efficiency. In our study, map all edges to their corresponding machines by hash function to ensure that adjacent edges are assigned to the same machine [37]. This approach can reduce the possibility of load imbalance and improve the overall performance of the algorithm. At the same time, we use the compressed sparse row (CSR) graph representation [38, 39], which stores the adjacency lists of vertices in the column array. In the CSR representation, the neighborhood information of each vertex is stored in a compact form, with the aim of minimizing memory consumption and improving the efficiency of the algorithm. CSR consists of vertex IDs, an index array, and a column array. The vertex ID serves as a unique identifier for each vertex and is used to locate and identify specific vertices in the graph. Each element in the index array indicates the starting position of each corresponding vertex's neighborhood in the column array. The column array stores the neighborhood information of each vertex in the graph.

Main workflow. The main workflow of PECC is illustrated as Fig. 3. Each expansion and allocation process is assigned to one of the $|P|$ machines. First, the input graph is distributed to some computing nodes by hash function (Step 1.1). Then, based on the initial vertex selection strategy, an initial exploration vertex is selected and added to the core vertex set to ensure that the selected vertex can effectively guide subsequent expansion processes (Step 1.2). During the expansion process, the algorithm selects the best exploration vertex based on a heuristic strategy and broadcasts it to the allocation process, waiting for the corresponding allocation process to distribute the edges (Step 3). The allocation process concurrently synchronizes the allocated edges to the corresponding expansion process, and the expansion process expand the partitions in parallel (Step 4). Finally, the iteration termination condition is verified (Step 5–6). At the end of the computation, each edge is assigned to the $|P|$ partitions.

Selecting initial exploration vertices. Initially, the expansion process in each iteration starts with the selection of a vertex based on the new rule. Algorithm 1 shows the details of the selection strategy. The following operations are performed: firstly, an examination is conducted to determine whether $S(P_k)$ is empty. If it is, it means that the partition has not yet started, so a vertex will be randomly chosen as the initial vertex. In the event that $S(P_k)$ is not empty, the vertex with the highest *Cluster_Coefficient* is selected, added to the current partition, and considered as the next exploration vertex. In this context, *Cluster_Coefficient* refers to the closeness between the vertex and the partition itself. A higher *Cluster_Coefficient* implies that the vertex is closer to the partition, making its selection beneficial to further expanding of the partition.

Algorithm 1 Selecting exploration vertices

```
Input: core set $C(P_k)$, boundary set $S(P_k)$
Output: $v_{best}$
1: if $S(P_k) = \emptyset$ then
2:     $v_{best} \leftarrow RandomSelect(O(P_k))$
3: else
4:     $v_{best} \leftarrow \arg\max_{v \in S(P_k) \setminus C(P_k)} Cluster\_Coefficient(v)$
5: end if
```

Each expansion process in parallel selects a new vertex from the boundary vertex set $S(P_k)$ to expand the edge set $E(P_k)$. The specific execution process is detailed in Algorithm 2. Each partition undergoes iterative computation, including the selection of the best exploration vertex (lines 5), the request allocation process (line 6), and the synchronization and updating of the boundary vertex set and edge set (lines 7-8). In the exploration vertex selection phase, if the partition capacity could not exceed the upper load limit and the boundary vertex set $S(P_k)$ is not empty, the best exploration vertex v_{best} is selected from $S(P_k)$ by Eq. (7). Otherwise, v_{best} is randomly selected from the external vertex set $O(P_k)$ of unallocated edges. Subsequently, during the request allocation phase, each expansion process propagates v_{best} to the corresponding allocation process and awaits the allocation process for edge distribution. Once the edge partition is completed, each expansion process receives new edges and adds E_{new} to $E(P_k)$. Finally, the expansion process verifies the termination condition.

Algorithm 2 PECC algorithm

```
**Input:** input Graph $G$, partition set $P$, load threshold $\delta$    $\triangleright\ \delta = \alpha\,|E|\,/\,|P|$
**Output:** final partition $P_{new}$
1:  $E(P_k) \leftarrow \emptyset$, $S(P_k) \leftarrow \emptyset$
2:  for $P_k$ in $P$ do
3:      $v_{best} \leftarrow \emptyset$
4:      while $|E(P_k)| \le \delta$ do
5:          $SelectExplorationVertex()$
6:          $AllocateEdges(v_{best})$
7:          $E_{new} \leftarrow ReceiveNewAlloEdges()$
8:          $E(P_{new}) \leftarrow E(P_k) \cup E_{new}$
9:          $|E|_{sum} \leftarrow GatherSum(E(P_k))$
10:         if $|E|_{sum} = |E|$ then
11:             $break$
12:         end if
13:     end while
14: end for
```

5.3 Allocation process

The allocation process uses a distributed approach to manage the input graph and is responsible for the assignment of edges. Algorithm 3 provides a detailed description of the execution steps of the allocation process. In the initial state, the unallocated edge set E_{new} is initialized as an empty set. At the same time, the union of the core vertex set $C(P_k)$ of partition P_k and the best exploration vertex v_{best} is set as a new variable C_{new}, while the union of the boundary vertex set $S(P_k)$ of partition P_k and the best exploration vertex v_{best} is set as a new variable S_{new}. Upon receiving the best exploration vertex v_{best}, the allocation process considers v_{best} as a core vertex and its adjacent vertices as boundary vertices. Then, by calculating the total number of edges between core vertices, between boundary vertices, and between core vertices and boundary vertices, denoted by $|E_{C_{new}-C_{new}}| + |E_{C_{new}-S_{new}}| + |E_{S_{new}-S_{new}}|$, the allocation process evaluates whether it exceeds the load threshold. If the condition is met, these edges are considered assignable and added to the E_{new}. Otherwise, if the condition is not satisfied, the allocation process selects the vertex with the highest *Clustering_Coefficient* from the boundary vertex set and adds the edges between the core vertices and that selected vertex to E_{new}. Assigning multiple unallocated edges at once effectively improves the iterative efficiency of the algorithm. At the end of each allocation process, the process sends the edge set E_{new} to the corresponding expansion process, ensuring cooperation among the processes.

Algorithm 3 Allocate edges

```
Input: best exploration vertex $v_{best}$
1: $E_{new} \leftarrow \emptyset$
2: $C_{new} \leftarrow C(P_k) \cup v_{best}$
3: $S_{new} \leftarrow S(P_k) \cup N(v_{best})$
4: if $|E_{C_{new}-C_{new}}| + |E_{C_{new}-C_{new}}| + |E_{C_{new}-C_{new}}| < \delta$ then
5:     $E_{new} \leftarrow E_{C_{new}-C_{new}} \cup E_{C_{new}-S_{new}} \cup E_{S_{new}-S_{new}}$
6: else
7:     $x \leftarrow \arg\max_{v \in S_{new} \setminus C_{new}} Cluster\_Coefficient(v)$
8:     $E_{new} \leftarrow E_{C_{new}-x}$
9: end if
10: $SynchronizeVertex()$
11: $SendNewAllocatedEdges(E_{new})$
```

Distributed lock. During the edge allocation process, there is a synchronization of global information between partitions. The synchronization process plays a crucial role in ensuring the consistency of allocated edges in a distributed environment. However, unlike in a single-machine environment where edges can be directly allocated, edge allocation in a distributed environment leads to concurrency management problems. When multiple partitions attempt to allocate the same edge simultaneously, conflicts arise and must be solved to determine which partition successfully allocates the corresponding edge.

Since the allocated edges in the edge partitioning model are unique to each partition, we only need to synchronize the globally shared information of the replicated vertices. Therefore, we propose the use of a distributed lock engine to effectively synchronize vertex information between partitions, ensuring the consistency and integrity of data in each process.

Distributed locks play a crucial role in coordinating access to shared resources among multiple processes or threads in a distributed system. Specifically, when an allocation process requires access to and modification of a vertex's information, it must first acquire the distributed lock associated with that vertex. Only the process that successfully acquires the lock can perform the corresponding operation, while other processes waiting for the lock will be blocked until the lock is released.

The execution process of distributed lock is illustrated as Fig. 4. Taking four partitions P_1, P_2, P_3, and P_4 as an example, in the allocation process, processes A_1 and A_3 simultaneously attempt to expand vertex v. Without distributed locks, A_1 and A_3 may modify the information of vertex v at the same time, leading to data conflicts. However, by introducing distributed locks, this situation can be avoided. When A_1 starts processing v, it first tries to acquire the shard lock associated with vertex v. After successfully obtaining the lock, A_1 attempts to add the edge (x, v)) to partition

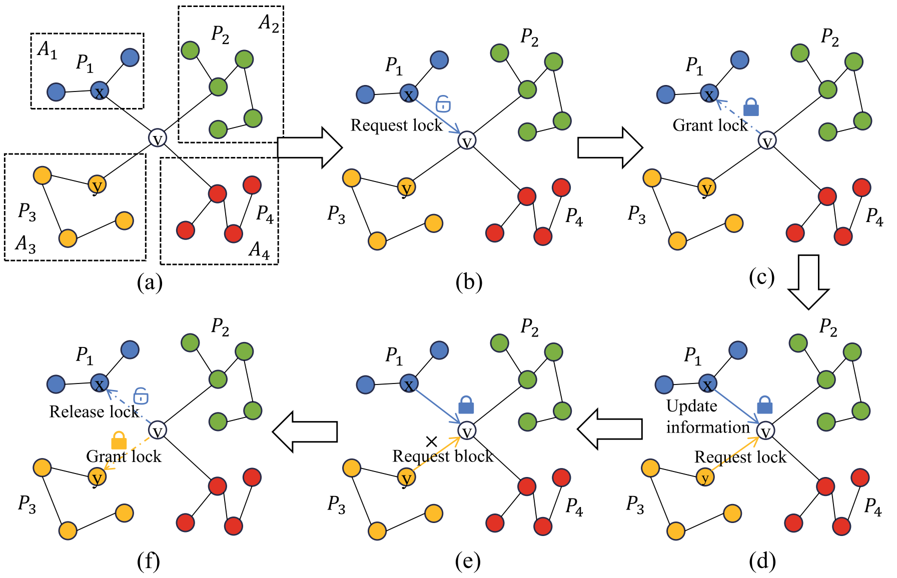

Fig. 4 The workflow of distributed locks. Each allocation flow (A_1, A_2, A_3, A_4) handles one partition, and each color (blue, green, yellow, red) represents a partition, and each line (solid, dashed) represents a request (Color figure online)

P_1 while updating the vertex-partition relationship (v, P_1) on vertex v, making vertex v a new boundary vertex. Meanwhile, when A_3 tries to acquire the same lock during this time, it is blocked until A_1 completes its operations on vertex v and releases the lock. This way, A_1 can only start processing vertex v after A_1 completes its operations, ensuring the serialization of operations on vertex v are serialized and avoiding data conflicts. During the processing of A_3, the vertex-partition relationship (v, P_3) on vertex v is updated. Once all operations are completed, the vertex-partition relationships of the duplicated vertex v are synchronized across each process. Following the synchronization process, the newly allocated edge set E_{new} is sent back to the expansion process. The expansion process then calls |ReceiveNewAlloEdges()| (Line 7 of Algorithm 2) to add E_{new} to the partition.

6 Evaluation

6.1 Experiment setting

Graph datasets. We evaluate the performance of our partitioning algorithm through a lot of real-world graphs obtained from the Stanford Network Analysis Platform (SNAP) website [40]. Table 1 presents the fundamental characteristics of the datasets used in our experiments. We choose graphs of different sizes to comprehensively evaluate the partitioning performance of algorithms. Different structure and

Table 1 Graph datasets used in this evaluation

Dataset	Vertices	Edges	Type
Wiki-Talk(WT)	2,394,385	5,021,410	Communication network
soc-LiveJ(LJ)	4,847,571	68,993,773	Social network
com-Orkut(OK)	3,072,441	117,185,083	Social network
com-Friendster(CF)	65,608,366	1,806,067,135	Social network

volumes of data lead to subtle variations in partitioning behavior and performance. These datasets are randomly ordered.

Benchmark partitioning algorithms. First, we compare with state-of-the-art offline and streaming algorithms on large-scale real-world graphs. NE [12] is the offline algorithm, whereas DBH [41], HDRF [14], SNE [12] and 2PS [42] are the streaming algorithms. DBH is fast streaming partitioner based on hashing for power-law graphs. HDRF achieves a lower replication factor with moderate runtime and memory overhead. NE is currently the edge partitioning algorithm with the best partitioning quality. SNE extends NE to a streaming partitioning algorithm, which takes into account both the partition quality and memory consumption. 2PS adopts a two-phase out-of-core model to achieve a trade-off between partitioning quality and running time. Second, we compare PECC with some distributed partitioning algorithms, including XtraPuLP [26] and Sheep [28]. XtraPuLP is the state-of-the-art distributed-memory graph partitioner. Sheep is a distributed edge partitioning algorithm that produces high partitioning quality. It is based on the construction of the distributed elimination tree. Finally, we compare our PECC integrated on GraphX with the built-in native algorithms, including RandomVertexCut (RVC), EdgePartition1D (EP1D), EdgePartition2D (EP2D) and CanonicalRandomVertex-Cut (CRVC). RVC achieves a randomized vertex-cut distribution by hashing the ID of source and destination vertices. EP1D allocates edges to corresponding partitions based only on the source ID. EP2D uses a two-dimensional partitioning strategy of the sparse edge adjacency matrix for edge allocation. CRVC allocates duplicate edges connecting any two specific vertices to the same partition using a standardized random vertex-cut method.

Evaluation scenario. We implement all methods in C++. All graph partitioning algorithms are evaluated in the distributed environment. We assume that it is most efficient for each partition to be assigned to one machine. Each machine in the cluster has the following specifications: Ubuntu 20.04 LTS operating system, Intel Ice Lake(2.7GHz/3.3GHz) with 8 cores Cpu, 32GB memory. To ensure reliable results, each evaluation result is an average of 10 runs and the differences from the mean are below 5%. For the sake of clarity, these minor deviations have been omitted from the presented results. At the same time, for HDRF, λ is also set to 1.1 [14].

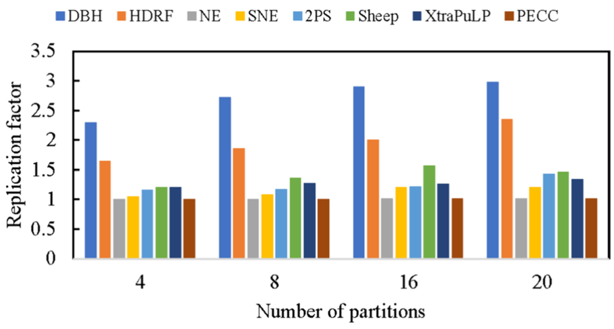

Fig. 5 Replication factor of WT on different partitions

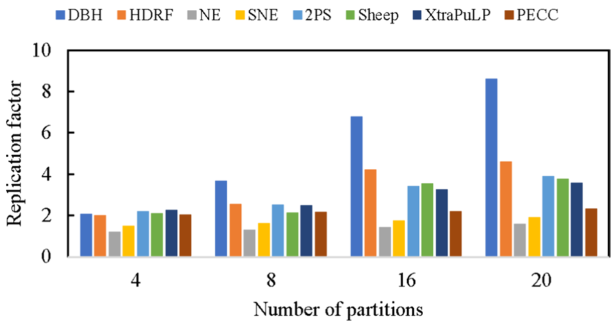

Fig. 6 Replication factor of LJ on different partitions

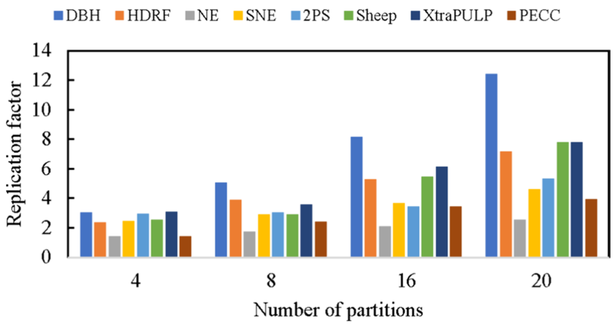

Fig. 7 Replication factor of OK on different partitions

6.2 Performance evaluation

Replication Factor (RF). All graph partitioning algorithms are tested on 4 different graph datasets with the number of different partitions (e.g. 4, 8, 16, 20). We first conducted a performance analysis of PECC and some classical offline and streaming

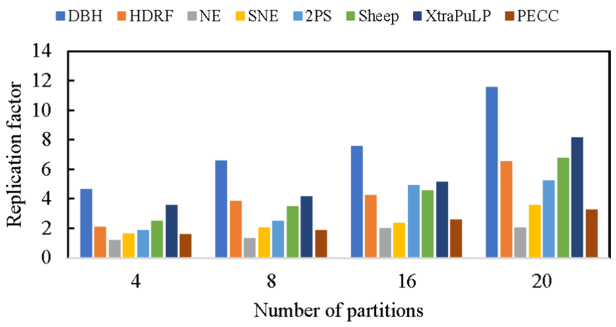

Fig. 8 Replication factor of CF on different partitions

partitioning algorithms. From the experimental results shown in Figs. 5, 6, 7 and 8, it can be demonstrated that the RF of DBH and HDRF perform poorly on all four datasets. DBH and HDRF are graph partitioning algorithms specifically designed for power-law graphs, with the main idea of replicating high-degree vertices as much as possible to reduce the replication factor generated by low-degree vertices. Both of two algorithms, however, are classified as streaming partitioning algorithms because they partition the edges only once during partitioning process. However, they fail to take into account the subsequent impact of incoming vertices on the partitions, resulting in relatively higher replication factors. 2PS yields a lower replication factor than DBH and HDRF, but it is far from the quality of PECC. On the other hand, NE consistently produces relatively lower replication factors, and SNE, as the streaming partitioning version of NE, also demonstrates good performance. According to literature [12], NE is currently regard as the best graph partitioning algorithm, while PECC achieves competitive partitioning performance on the four datasets.

Furthermore, we measured the performance of Sheep and XtraPuLP partitioning algorithms in a distributed environment, assuming that each partition is assigned to a separate machine for optimal efficiency. By comparing these algorithms, PECC consistently produces the lowest replication factors. In contrast, XtraPuLP produces poorer replication factors. This is because XtraPuLP prioritizes optimizing the partition balance during the iterative process and then considers minimizing the edge cutting ratio. Additionally, it only stops the algorithm execution based on the maximum number of iterations, which can result in poorer replication factors. However, on the OK dataset, PECC produces larger replication factors, reaching a value of 3.9 on 32 partitions. This is mainly because it is more challenging to generate high-quality partitions with the increase of graph size. Additionally, the OK dataset follows a power-law distribution, where the degree distribution of vertices is highly uneven and a few vertices have extremely high degrees. In summary, PECC demonstrates relatively good partition quality on real-world graphs with different scales.

Running Time (RC). As shown in fig.9, we compare the RC of PECC with the aforementioned algorithms on 16 partition. It is evident that PECC is significantly faster than offline and streaming algorithms. Compared with distributed graph partitioning algorithms, PECC outperform Sheep in terms of efficiency. These results

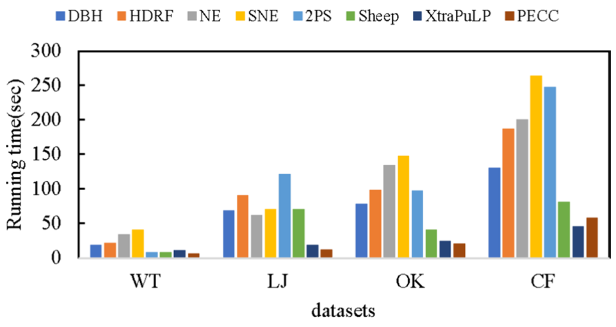

Fig. 9 Running time of different partitioning algorithms on 16 partitions

Table 2 Partitioing quality on GraphX

	Wiki-Talk			Soc-LiveJ			Com-Orkut			Com-Friendster		
	RF	EB	VB	RF	EB	VB	RF	EB	VB	RF	EB	VB
RVC	1.4	1	1	3	1	1	3.8	1	1	7.6	1	1
CRVC	1.3	1	1	2.8	1	1	3.8	1	1	7.6	1	1
EP1D	1.3	1.1	1.1	2.8	1	1	3.7	1	1	7.6	1	1
EP2D	1.2	1.1	1.1	2.4	1	1.3	2.8	1	1.3	5.3	1.1	1.1
PECC	1	1.3	1.3	1.8	1.3	1.3	2.3	1.3	1.3	2.5	1	1

The number of partitions is 4

highlight the efficiency of the PECC algorithm in diverse real-world graph scenarios. it is important to note that PECC consistently demonstrates competitive performance with other state-of-the-art algorithms, such as XtraPuLP. On LJ and CF datasets, XtraPuLP takes less time than PECC because XtraPuLP speeds up algorithm convergence at the expense of sacrificing partition quality. Understanding these variations in algorithm performance across different datasets and scenarios is essential for making informed decisions when selecting a distributed graph partitioning algorithm.

6.3 Experiments on GraphX

We compares our PECC algorithm against four native partitioners, including RVC, CRVC, EP1D and EP2D, in GraphX, in terms of replication factor (RF), edge balance (EB), and vertex balance (VB) on a cluster of 20 machines. Each result is an average of ten runs to ensure that their relative standard error is less than 5%. As shown in Table 2, 3, 4 and 5, RVC and CRVC, based on hash functions, randomly select and cut vertices, which to some extent limits their randomness due to the graph's structural characteristics. While they achieve load balance, the randomness leads to instability in partitioning results. In term of RF,

Table 3 Partitioing quality on GraphX

	Wiki-Talk			Soc-LiveJ			Com-Orkut			Com-Friendster		
	RF	EB	VB	RF	EB	VB	RF	EB	VB	RF	EB	VB
RVC	1.5	1	1	5	1	1	7.2	1	1	11.6	1	1
CRVC	1.5	1	1	4.3	1	1	7.2	1	1	11.6	1	1
EP1D	1.5	1.2	1.2	4.3	1	1	6.8	1	1	11.3	1	1
EP2D	1.3	1	1	3.4	1	1	4.3	1	1.1	6.4	1.2	1.2
PECC	1.1	1	1	2.5	1	1	2.7	1	1	2.8	1	1

The number of partitions is 8

Table 4 Partitioing quality on GraphX

	Wiki-Talk			Soc-LiveJ			Com-Orkut			Com-Friendster		
	RF	EB	VB	RF	EB	VB	RF	EB	VB	RF	EB	VB
RVC	1.7	1	1	7.7	1	1	13.1	1	1	15.1	1	1
CRVC	1.6	1	1	6.4	1	1	13.1	1	1	15.1	1	1
EP1D	1.6	1.3	1.3	6.2	1	1	11.4	1.1	1.1	14.7	1	1
EP2D	1.4	1.1	1.1	4.5	1	1.1	6.1	1	1.1	7.5	1.1	1.1
PECC	1.2	1	1	2.7	1	1	3.1	1	1	3.2	1.1	1.1

The number of partitions is 16

Table 5 Partitioing quality on GraphX

	Wiki-Talk			Soc-LiveJ			Com-Orkut			Com-Friendster		
	RF	EB	VB	RF	EB	VB	RF	EB	VB	RF	EB	VB
RVC	1.7	1	1	8.8	1	1	15.6	1	1	17.8	1	1
CRVC	1.7	1	1	7.1	1	1	15.6	1	1	17.8	1	1
EP1D	1.6	1.3	1.3	6.8	1	1	13.2	1	1	16.4	1	1
EP2D	1.4	1.1	1.1	4.9	1	1	6.9	1	1	8.6	1	1
PECC	1.2	1.1	1.1	3.2	1.3	1.3	3.2	1	1	3.5	1.3	1.1

The number of partitions is 20

EP1D slightly improves partitioning quality by placing edges of source vertices in the same partition, effectively reducing the number of cut vertices. EP2D further optimizes partitioning quality by dividing edges in a two-dimensional space, but EP1D and EP2D lack effective heuristic strategies, resulting in lower partition quality. Compared with the above algorithms, PECC achieves better partitioning quality. However, in the LJ dataset with 20 partitions, VB is relatively poor. This is because PECC uses a heuristic strategy during iterations, where each partition selects the best exploration vertex based on its current state. This helps control the growth rate of vertices in each partition, ensuring high-quality partitions are achieved. Consequently, PECC consistently outperforms the native graph partitioning algorithms of GraphX in terms of partitioning quality and load balance.

7 Conclusion

In this paper, we firstly conducted a comprehensive study of NE and analysed its shortcomings during the partitioning process. On the one hand, the internal structural changes during the expansion of vertex neighborhood causes partition disturbance. To address this problem, we proposed a heuristic partitioning strategy based on clustering coefficient to improve the quality of algorithm. On the other hand, in a distributed environment, multiple processes partitioning the same graph element simultaneously leads to concurrency management problems. To address this problem, we introduced a distributed lock engine to ensure the consistency of global synchronization information. Based on the above challenges, we presented the parallel extension version of NE, called PECC. Finally, experimental results on four datasets demonstrated that PECC can achieve a balance between partition quality and computational efficiency.

Nevertheless, the scope of this study is limited to homogeneous computing environment. In future work, we aim to reconstruct and optimize the proposed algorithm in heterogeneous computing environment.

Acknowledgements This work was supported by the National Natural Science Foundation of China under Grants 62272201 and 61872166.

Author contributions Zhenping Xie contributed to the conception of the study, reviewed and edited the manuscript. Chengcheng Shi performed the experiment and the data analyses, and wrote the manuscript.

Data availability No datasets were generated or analysed during the current study.

Declarations

Conflict of interest None of the authors have any competing interests in the manuscript.

References

1. Gao, C., Zheng, Y., Li, N., Li, Y., Qin, Y., Piao, J., Quan, Y., Chang, J., Jin, D., He, X.: A survey of graph neural networks for recommender systems: challenges, methods, and directions. ACM Trans. Recomm. Syst. **1**(1), 1–51 (2023)
2. 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, pp. 135–146 (2010)
3. Gonzalez, J.E., Low, Y., Gu, H., Bickson, D., Guestrin, C.: {PowerGraph}: distributed {graph-parallel} computation on natural graphs. In: 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12), pp. 17–30 (2012)

4. Xu, J., Bai, Z., Fan, W., Lai, L., Li, X., Li, Z., Qian, Z., Wang, L., Wang, Y., Yu, W.: Graphscope: a one-stop large graph processing system. Proc. VLDB Endow. **14**(12), 2703–2706 (2021)
5. Fan, W., He, T., Lai, L., Li, X., Li, Y., Li, Z., Qian, Z., Tian, C., Wang, L., Xu, J.: Graphscope: a unified engine for big graph processing. Proc. VLDB Endow. **14**(12), 2879–2892 (2021)
6. Yang, K., Zhang, M., Chen, K., Ma, X., Bai, Y., Jiang, Y.: Knightking: a fast distributed graph random walk engine. In: Proceedings of the 27th ACM Symposium on Operating Systems Principles, pp. 524–537 (2019)
7. Zhu, X., Chen, W., Zheng, W., Ma, X.: Gemini: a {Computation-Centric} distributed graph processing system. In: 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16), pp. 301–316 (2016)
8. Ayall, T., Duan, H., Liu, C., Gereme, F., Abegaz, M., Deleli, M.: Taking heuristic based graph edge partitioning one step ahead via offstream partitioning approach. In: 2021 IEEE 37th International Conference on Data Engineering (ICDE), pp. 2081–2086. IEEE (2021)
9. Faraj, M.F., Schulz, C.: Buffered streaming graph partitioning. ACM J. Exp. Algorithmics **27**, 1–26 (2022)
10. Zwolak, M., Abbas, Z., Horchidan, S., Carbone, P., Kalavri, V.: Gcnsplit: bounding the state of streaming graph partitioning. In: Proceedings of the Fifth International Workshop on Exploiting Artificial Intelligence Techniques for Data Management, pp. 1–12 (2022)
11. Devi, N.M., et al.: Stream-based vertex cut partitioning with buffer support for power-law graphs (svbp). Turk. J. Comput. Math. Educ. (TURCOMAT) **12**(6), 5335–5350 (2021)
12. Zhang, C., Wei, F., Liu, Q., Tang, Z.G., Li, Z.: Graph edge partitioning via neighborhood heuristic. In: Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 605–614 (2017)
13. Mayer, R., Jacobsen, H.-A.: Hybrid edge partitioner: partitioning large power-law graphs under memory constraints. In: Proceedings of the 2021 International Conference on Management of Data, pp. 1289–1302 (2021)
14. 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, pp. 243–252 (2015)
15. Gonzalez, J.E., Xin, R.S., Dave, A., Crankshaw, D., Franklin, M.J., Stoica, I.: {GraphX}: graph processing in a distributed dataflow framework. In: 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14), pp. 599–613 (2014)
16. Jain, N., Liao, G., Willke, T.L.: Graphbuilder: scalable graph etl framework. In: First International Workshop on Graph Data Management Experiences and Systems, pp. 1–6 (2013)
17. Chen, R., Shi, J., Chen, Y., Zang, B., Guan, H., Chen, H.: Powerlyra: differentiated graph computation and partitioning on skewed graphs. ACM Trans. Parallel Comput. (TOPC) **5**(3), 1–39 (2019)
18. Valiant, L.G.: A bridging model for parallel computation. Commun. ACM **33**(8), 103–111 (1990)
19. Zaharia, M., Xin, R.S., Wendell, P., Das, T., Armbrust, M., Dave, A., Meng, X., Rosen, J., Venkataraman, S., Franklin, M.J., et al.: Apache spark: a unified engine for big data processing. Commun. ACM **59**(11), 56–65 (2016)
20. Zhuo, Y., Chen, J., Luo, Q., Wang, Y., Yang, H., Qian, D., Qian, X.: Symplegraph: distributed graph processing with precise loop-carried dependency guarantee. In: Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 592–607 (2020)
21. Li, X., Zhang, M., Chen, K., Wu, Y.: Regraph: a graph processing framework that alternately shrinks and repartitions the graph. In: Proceedings of the 2018 International Conference on Supercomputing, pp. 172–183 (2018)
22. Karypis, G., Kumar, V.: Multilevelk-way partitioning scheme for irregular graphs. J. Parallel Distrib. Comput. **48**(1), 96–129 (1998)
23. Pellegrini, F., Roman, J.: Scotch: a software package for static mapping by dual recursive bipartitioning of process and architecture graphs. In: High-Performance Computing and Networking: International Conference and Exhibition HPCN EUROPE 1996 Brussels, Belgium, April 15–19, 1996 Proceedings, vol. 4, pp. 493–498. Springer (1996)
24. Hendrickson, B., Leland, R.W., et al.: A multi-level algorithm for partitioning graphs. SC **95**(28), 1–14 (1995)
25. Chevalier, C., Pellegrini, F.: Pt-scotch: a tool for efficient parallel graph ordering. Parallel Comput. **34**(6–8), 318–331 (2008)

26. Slota, G.M., Rajamanickam, S., Devine, K., Madduri, K.: Partitioning trillion-edge graphs in minutes. In: 2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS), pp. 646–655. IEEE (2017)
27. 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. IEEE (2014)
28. Martella, C., Logothetis, D., Loukas, A., Siganos, G.: Spinner: scalable graph partitioning in the cloud. In: 2017 IEEE 33rd International Conference on Data Engineering (ICDE), pp. 1083–1094. IEEE (2017)
29. Margo, D., Seltzer, M.: A scalable distributed graph partitioner. Proc. VLDB Endow. **8**(12), 1478–1489 (2015)
30. Dean, J., Ghemawat, S.: Mapreduce: simplified data processing on large clusters. Commun. ACM **51**(1), 107–113 (2008)
31. Zeng, L., Huang, H., Zheng, B., Yang, K., Shao, S., Zhou, J., Xie, J., Zhao, R., Chen, X.: Windgp: efficient graph partitioning on heterogenous machines (2024). arXiv preprint arXiv:2403.00331
32. 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. In: 2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS), pp. 685–695. IEEE (2018)
33. Kong, D., Xie, X., Zhang, Z.: Clustering-based partitioning for large web graphs. In: 2022 IEEE 38th International Conference on Data Engineering (ICDE), pp. 593–606. IEEE (2022)
34. Hajidehi, M.R., Sridhar, S., Seltzer, M.: Cuttana: scalable graph partitioning for faster distributed graph databases and analytics (2024). arXiv preprint arXiv:2312.08356
35. Tang, Z., He, M., Fu, Z., Yang, L.: Incgraph: an improved distributed incremental graph computing model and framework based on spark graphx. IEEE Trans. Knowl. Data Eng. **34**(6), 2783–2797 (2020)
36. Ammar, K., Özsu, M.T.: Experimental analysis of distributed graph systems. Proc. VLDB Endow. **11**(10), 1151–1164 (2018)
37. Shuang, Z., Yubin, B., Zhigang, W., Fangling, L., Ge, Y., Chao, D., Leitao, G.: Bhp: Bsp model oriented hash graph data partition with load balancing. J. Front. Comput. Sci. Technol. **8**(1), 40 (2014)
38. Sahu, S.: Gvel: fast graph loading in edgelist and compressed sparse row (csr) formats (2023). arXiv preprint arXiv:2311.14650
39. Lane, P.A., Booth, J.D.: Heterogeneous sparse matrix-vector multiplication via compressed sparse row format. Parallel Comput. **115**, 102997 (2023)
40. Leskovec, J., Sosič, R.: Snap: a general-purpose network analysis and graph-mining library. ACM Trans. Intell. Syst. Technol. (TIST) **8**(1), 1–20 (2016)
41. Xie, C., Yan, L., Li, W.-J., Zhang, Z.: Distributed power-law graph computing: theoretical and empirical analysis. In: Advances in Neural Information Processing Systems, vol. 27 (2014)
42. Mayer, R., Orujzade, K., Jacobsen, H.-A.: Out-of-core edge partitioning at linear run-time. In: 2022 IEEE 38th International Conference on Data Engineering (ICDE), pp. 2629–2642. IEEE (2022)