A Survey of Distributed Graph Algorithms on Massive Graphs

LINGKAI MENG, Shanghai Jiao Tong University, Shanghai, China
YU SHAO, East China Normal University, Shanghai, China
LONG YUAN*, Nanjing University of Science and Technology, Nanjing, China
LONGBIN LAI, Alibaba Group, Hangzhou, China
PENG CHENG, East China Normal University, Shanghai, China
XUE LI, Alibaba Group, Hangzhou, China
WENYUAN YU, Alibaba Group, Hangzhou, China
WENJIE ZHANG, University of New South Wales, Sydney, Australia
XUEMIN LIN, Shanghai Jiao Tong University, Shanghai, China
JINGREN ZHOU, Alibaba Group, Hangzhou, China

Distributed processing of large-scale graph data has many practical applications and has been widely studied. In recent years, a lot of distributed graph processing frameworks and algorithms have been proposed. While many efforts have been devoted to analyzing these, with most analyzing them based on programming models, less research focuses on understanding their challenges in distributed environments. Applying graph tasks to distributed environments is not easy, often facing numerous challenges through our analysis, including parallelism, load balancing, communication overhead, and bandwidth. In this paper, we provide an extensive overview of the current state-of-the-art in this field by outlining the challenges and solutions of distributed graph algorithms. We first conduct a systematic analysis of the inherent challenges in distributed graph processing, followed by presenting an overview of existing general solutions. Subsequently, we survey the challenges highlighted in recent distributed graph processing papers and the strategies adopted to address them. Finally, we discuss the current research trends and identify potential future opportunities.

CCS Concepts: • **Computing methodologies → Distributed algorithms**; • **Theory of computation → Distributed algorithms**; **Graph algorithms analysis**.

Additional Key Words and Phrases: Distributed Processing, Graph Algorithms, Big Data

*Corresponding author.

Authors' Contact Information: Lingkai Meng, Shanghai Jiao Tong University, Shanghai, Shanghai, China; e-mail: mlk123@sjtu.edu.cn; Yu Shao, East China Normal University, Shanghai, Shanghai, China; e-mail: yushao@stu.ecnu.edu.cn; Long Yuan, Nanjing University of Science and Technology, Nanjing, China; e-mail: longyuan@njust.edu.cn; Longbin Lai, Alibaba Group, Hangzhou, Zhejiang, China; e-mail: longbin.lailb@alibaba-inc.com; Peng Cheng, East China Normal University, Shanghai, Shanghai, China; e-mail: pcheng@sei.ecnu.edu.cn; Xue Li, Alibaba Group, Hangzhou, Zhejiang, China; e-mail: youli.lx@alibaba-inc.com; Wenyuan Yu, Alibaba Group, Hangzhou, Zhejiang, China; e-mail: wenyuan.ywy@alibaba-inc.com; Wenjie Zhang, University of New South Wales, Sydney, New South Wales, Australia; e-mail: wenjie.zhang@unsw.edu.au; Xuemin Lin, Shanghai Jiao Tong University, Shanghai, China; e-mail: xuemin.lin@gmail.com; Jingren Zhou, Alibaba Group, Hangzhou, Zhejiang, China; e-mail: jingren.zhou@alibaba-inc.com.

ACM 1557-7341/2024/9-ART
https://doi.org/10.1145/3694966

1 Introduction

Graph is a high-dimensional structure that models point-to-point relationships among entities. Due to its strong representation ability, the graph is widely applied in social network analysis [27], road network routing [75], and biological structure forecasting [23]. With the development of information science and big data applications [1, 54] in recent years, the scales of graph data sets become too large for single machines due to their limited storage and computing power. To support the queries and analyses on massive graphs, researchers proposed many distributed graph algorithms and systems to store a large-scale graph in multiple machines separately and compute collaboratively, such as Pregel [118], Giraph [11], GraphX [77], and GraphScope [62].

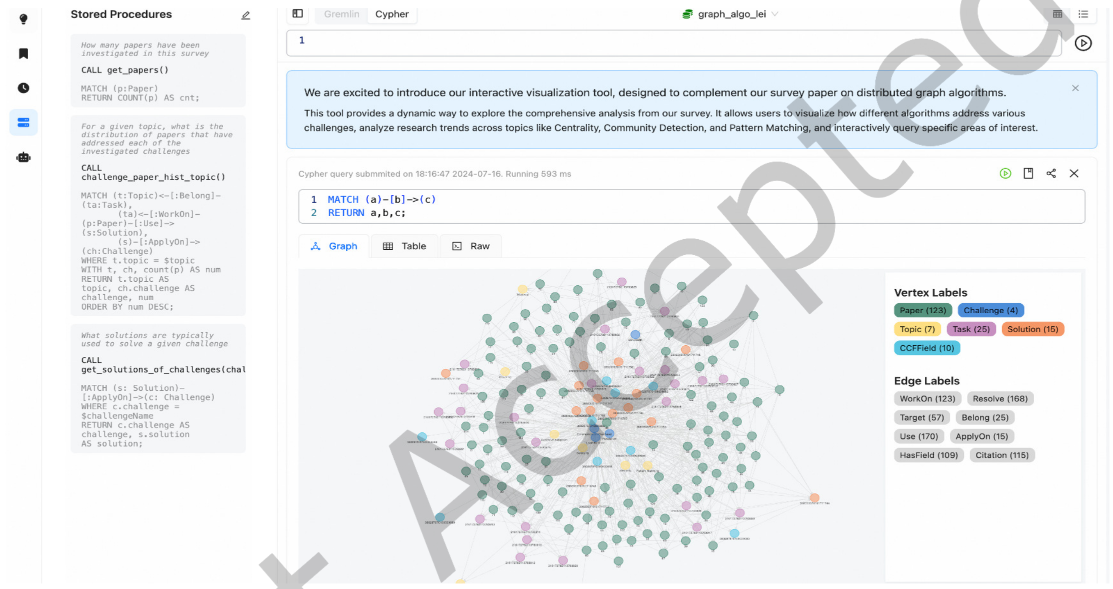
Recent years have seen a surge in research on distributed graph algorithms, with a focus on developing specific algorithms like PageRank, label propagation, and triangle counting, or addressing challenges such as workload scheduling and machine-to-machine communication. However, comprehensive surveys providing a comprehensive view of the field remain limited. This paper aims to bridge that gap by consolidating research on distributed graph algorithms for massive graphs over the past decade, as featured in prestigious conferences and journals like SIGMOD, VLDB, PPoPP, SC, TPDS, and TC. We distill four primary and recurrently addressed challenges among these papers:

- **Parallelism** is a major objective that requires processing multiple operations at the same time and reducing iteration rounds. Achieving efficient parallelism is challenging due to the inherent sequential dependencies in graph analysis tasks. These dependencies require certain computations to be completed before others can begin, limiting the ability to execute multiple operations simultaneously.
- **Load balance** aims to distribute work evenly across vertices and improve the utilization of computing resources. This helps prevent some machines from becoming overloaded while others remain idle. Graphs often have skewed degree distributions, where a small number of vertices have many connections while the majority have few. This imbalance can lead to uneven distribution of workload across machines.
- **Communication** refers to the exchange of messages between vertices, which is an expensive operation compared to random memory access. Distributed graph processing involves frequent exchanges of data between machines. These communications can become a significant overhead, especially in large-scale graphs, due to the complex and dense interconnections between vertices. Optimizing communication overhead can improve execution efficiency in practice.
- **Bandwidth** limits the size of messages transmitted between vertices. Some algorithms require a large amount of bandwidth, which may not be feasible in certain frameworks. The primary reasons for the bandwidth limitations challenge in distributed graph algorithms are twofold: first, high-degree vertices necessitate substantial bandwidth for communication; second, the frequent transmission of small messages can result in inefficient bandwidth utilization.

To address the challenges, a lot of open-source distributed graph processing frameworks (e.g., Pregel [118] and GPS [137]) have been proposed. Generic solutions (e.g., parallel looping, message receiving and sending, and broadcasting) are abstracted in these frameworks. Users have the flexibility to develop graph algorithms following high-level functionalities, effectively abstracting away the complexities of underlying implementation details. However, these solutions are highly diverse and tailored for specific algorithms due to the irregularity of graph algorithms, and there is no one unified pattern that can fit all graph algorithms.

In addition, various graph tasks are addressed by distributed graph algorithms in the existing studies. To clearly introduce them, we classify the widely studied graph tasks, including PageRank, Louvain, SimRank, max flow, subgraph matching, and vertex cover, into seven topics: *Centrality*, *Community Detection*, *Similarity*, *Cohesive Subgraph*, *Traversal*, *Pattern Matching*, and *Covering*.

In this paper, we first introduce the generic solutions for the four challenges and then dissect the proportion of research papers that address the challenges across different algorithmic topics. Moreover, we delve into the reasons behind the varying degrees of attention that certain challenges receive within specific topics. For example, 70% of the papers related to *Similarity* topic (Figure 7c) have focused on reducing communication overhead. Through the analysis, we show some deep views of the research in distributed graph algorithms and also give insights into the potentially promising directions for future studies. A unique contribution of this work is the construction of a comprehensive graph that encapsulates the surveyed material, as shown in Figure 1. This graph maps out the intricate connections among papers, topics, algorithms, solutions, and challenges, etc., providing a visual narrative of the landscape. Readers can play with the graph using an online interactive tool (http://gsp.blue/querying?name=graph_algo), which includes several examples to guide readers on how to use the tool.



```
How many papers have been
investigated in this survey

CALL get_papers()

MATCH (p:Paper)
RETURN COUNT(p) AS cnt;
```

```
For a given topic, what is the
distribution of papers that have
addressed each of the
investigated challenges

CALL
challenge_paper_hist_topic()

MATCH (t:Topic)<-[:Belong]-
(ta:Task),
       (ta)<-[:WorkOn]-
(p:Paper)-[:Use]->
(s:Solution),
       (s)-[:ApplyOn]->
(ch:Challenge)
WHERE t.topic = $topic
WITH t, ch, count(p) AS num
RETURN t.topic AS
topic, ch.challenge AS
challenge, num
ORDER BY num DESC;
```

```
What solutions are typically
used to solve a given challenge

CALL
get_solutions_of_challenges(chal

MATCH (s: Solution)-
[:ApplyOn]->(c: Challenge)
WHERE c.challenge =
$challengeName
RETURN c.challenge AS
challenge, s.solution
AS solution;
```

We are excited to introduce our interactive visualization tool, designed to complement our survey paper on distributed graph algorithms.

This tool provides a dynamic way to explore the comprehensive analysis from our survey. It allows users to visualize how different algorithms address various challenges, analyze research trends across topics like Centrality, Community Detection, and Pattern Matching, and interactively query specific areas of interest.

Cypher query submitted on 18:16:47 2024-07-16. Running 593 ms

```
MATCH (a)-[b]->(c)
RETURN a,b,c;
```

Fig. 1. A comprehensive graph that encapsulates the surveyed material.

__**Contribution**__. Existing surveys focus on either specific distributed challenges (e.g., load balance [92]) or particular distributed algorithms (e.g., pattern matching [24]). However, our survey targets challenges faced by different distributed graph algorithms in processing, with irregular computation taken into consideration. Specifically, we make the main contributions as follows:

- We provide an overview of the main challenges in distributed graph algorithms and the different solutions proposed to tackle them. This provides a comprehensive understanding of distributed graph processing.
- We survey various distributed graph algorithms and categorize them into seven topics based on the challenges they address.
- For each topic of distributed graph algorithms, we conduct a thorough analysis of existing efforts. We also summarize the main challenges they address and provide unique insights into the underlying reasons.

The outline of our survey is shown in Figure 2 and the rest of this paper is organized as follows. Section 2 provides a review of existing distributed graph systems and computing processing. Section 3 summarizes some challenges and solutions, which are not common for single-machine algorithms. Section 4 describes popular distributed graph algorithms in detail, highlighting differences from their single-machine versions. Section 5 presents various application scenarios for different graph tasks. Section 6 discusses prevalent research trends and potential research opportunities. Section 7 concludes this survey.

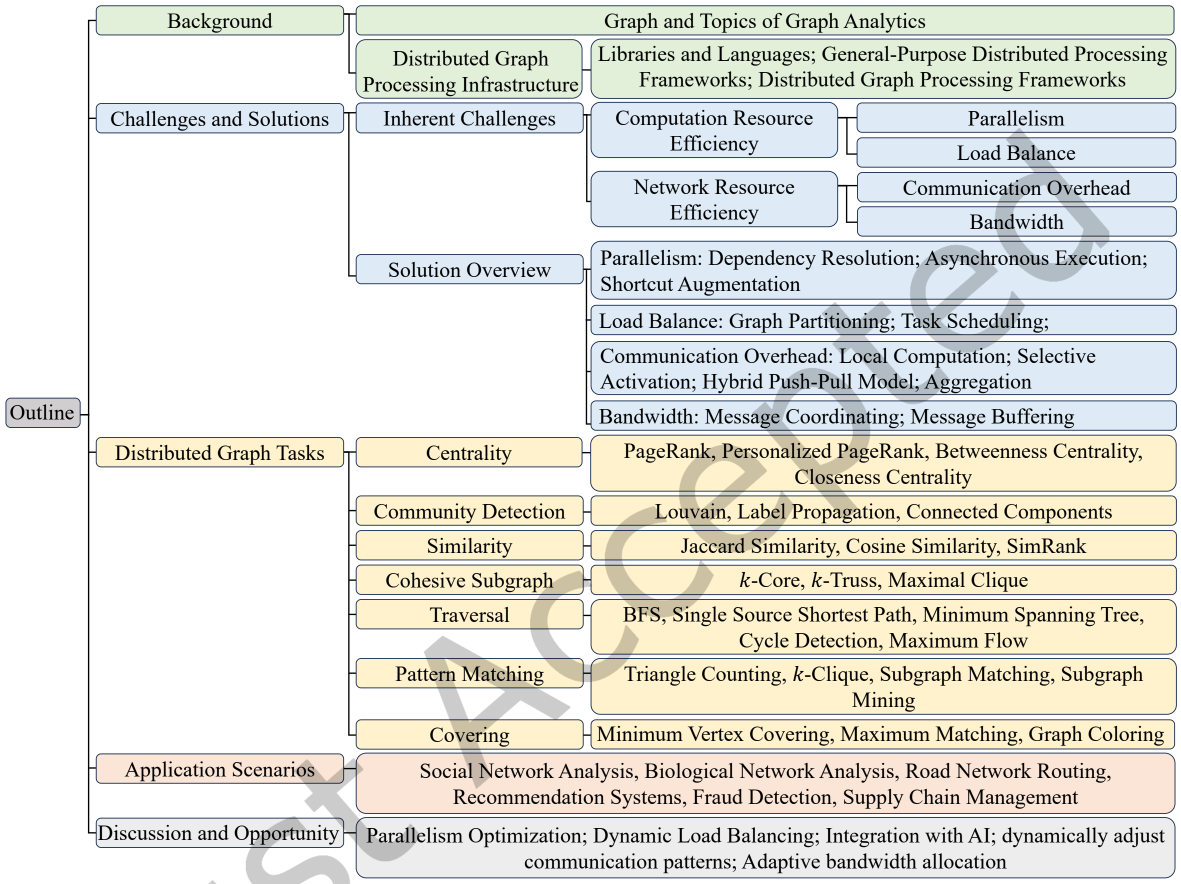

Outline
Background
Graph and Topics of Graph Analytics
Distributed Graph Processing Infrastructure
Libraries and Languages; General-Purpose Distributed Processing Frameworks; Distributed Graph Processing Frameworks
Challenges and Solutions
Inherent Challenges
Computation Resource Efficiency
Parallelism
Load Balance
Network Resource Efficiency
Communication Overhead
Bandwidth
Solution Overview
Parallelism: Dependency Resolution; Asynchronous Execution; Shortcut Augmentation
Load Balance: Graph Partitioning; Task Scheduling;
Communication Overhead: Local Computation; Selective Activation; Hybrid Push-Pull Model; Aggregation
Bandwidth: Message Coordinating; Message Buffering
Distributed Graph Tasks
Centrality
PageRank, Personalized PageRank, Betweenness Centrality, Closeness Centrality
Community Detection
Louvain, Label Propagation, Connected Components
Similarity
Jaccard Similarity, Cosine Similarity, SimRank
Cohesive Subgraph
k-Core, k-Truss, Maximal Clique
Traversal
BFS, Single Source Shortest Path, Minimum Spanning Tree, Cycle Detection, Maximum Flow
Pattern Matching
Triangle Counting, k-Clique, Subgraph Matching, Subgraph Mining
Covering
Minimum Vertex Covering, Maximum Matching, Graph Coloring
Application Scenarios
Social Network Analysis, Biological Network Analysis, Road Network Routing, Recommendation Systems, Fraud Detection, Supply Chain Management
Discussion and Opportunity
Parallelism Optimization; Dynamic Load Balancing; Integration with AI; dynamically adjust communication patterns; Adaptive bandwidth allocation

Fig. 2. Outline of the Survey

2 Background

2.1 Graph and Topics of Graph Analytics

We first define the graph and other relevant concepts that are used throughout this paper.

Definition 2.1 (Graph). A graph G comprises a set $V(G)$ of n vertices and a set $E(G)$ of m edges. Vertices represent entities, and edges signify the relationships between entities. This paper primarily focuses on simple graphs, namely having at most one edge between any pair of vertices. Graphs are categorized into directed and undirected graphs. In an undirected graph, an edge (u, v) denotes an unordered pair of vertices, implying that $(u, v) = (v, u)$. Conversely, in a directed graph, an edge (u, v) represents an ordered pair of vertices, where

$(u, v) \neq (v, u)$. Additionally, when edges are assigned a float number based on edge attributes, the graph is termed a weighted graph.

Definition 2.2 (Density). The density of a graph is defined as the ratio of the edge number m and the maximum possible edge number (n^2 in directed graphs and $n(n-1)/2$ in undirected graphs). Generally, a graph is sparse if $m \ll O(n^2)$ and dense otherwise.

Definition 2.3 (Neighbors and Degree). In a directed graph G, for any given vertex $u \in V(G)$, we define the in-neighbor and out-neighbor set of u as $N_{in}(u) = \{v|(v,u) \in E(G)\}$ and $N_{out}(u) = \{v|(u,v) \in E(G)\}$. The in-degree and out-degree of vertex u are the sizes of $N_{in}(u)$ and $N_{out}(u)$, respectively. In the context of an undirected graph G, the neighbor set for a vertex $u \in V(G)$ is defined as $N(u) = \{v|(u,v) \in E(G)\}$, and its degree is the size of the neighbor set.

Definition 2.4 (Path). A path in a graph is a sequence of vertices where each pair of consecutive vertices is connected by an edge. Specifically, a path P can be defined as $P = (v_1, v_2, \ldots, v_k)$, where each pair $(v_i, v_{i+1}) \in E(G)$ for all $1 \leq i < k$. The length of the path is determined by the number of vertices it contains, which in this case is k. In a weighted graph, where edges carry weights, the distance is defined as the minimum sum of the weights of all edges (v_i, v_{i+1}) along all paths. Consequently, the shortest path length and distance from vertex u to vertex v are the minimum length and the minimum sum of weights, respectively, among all possible paths connecting u and v.

Definition 2.5 (Diameter). The diameter of a graph is defined as the longest shortest path *length* between any pair of vertices. In other words, it represents the largest length that must be traversed to connect any two vertices in the graph using the shortest path between them.

Definition 2.6 (Cycle). A cycle is a special case of a path that starts and ends at the same vertex, i.e., $C = (v_1, v_2, \ldots, v_k)$ with $v_1 = v_k$. The length of a cycle is the number of vertices it contains.

Definition 2.7 (Tree). A tree, commonly defined within the scope of an undirected graph, is a connected graph that is free of cycles. A forest is a *disjoint* set of trees.

Definition 2.8 (Subgraph). A subgraph H of a graph G is another graph where $V(H) \subset V(G)$, $E(H) \subset E(G)$, and for each $(u,v) \in E(H)$, $u, v \in V(H)$.

Due to the diverse application scenarios of graph analytics, a broad range of topics within this field are explored in the literature. In this paper, we concentrate on specific categories of graph analytics that are frequently examined in the context of distributed graph algorithms. A similar taxonomy can be found in other works [58, 83, 103]. These categories are: (1) **Centrality**: Focus on determining the relative significance of each vertex within a complex graph. (2) **Community Detection**: Aim at segregating the vertices of a graph into distinct communities, characterized by denser connections within the same community compared to those with vertices outside the community. (3) **Similarity**: Concern with assessing the likeness between two vertices in a graph, based on defined criteria. (4) **Cohesive Subgraph**: Involve identifying subgraphs that meet specific cohesion criteria. (5) **Traversal**: Entail visiting vertices in a graph following a certain procedure, to review or modify the vertices' attributes. (6) **Pattern Matching**: Search for specific input patterns within a graph. (7) **Covering**: Address combinatorial optimization problems in graphs, often linked to minimum vertex-covering challenges. These categories represent key areas in graph analytics, each with distinct methodologies and applications in distributed graph algorithms, as will be discussed in detail in Section 4.

2.2 Distributed Graph Processing Infrastructure

Processing a large volume of data on a cluster of machines has been studied for decades in computer science. Many tools and systems have been developed to facilitate distributed computing, which can be utilized for distributed graph processing as well. We divide them into three categories based on their closeness to distributed graph processing as follows:

2.2.1 *Distributed Computing Libraries and Languages.*

To support efficient distributed algorithm and system development, distributed computing libraries based on traditional sequential languages are designed, such as MPI [65], Java Remote Method Invocation [126], PyCOMPSs [15], and OpenSHMEM [40]. Among them, MPI is one of the earliest libraries designed for distributed computing. Strictly, MPI is a library standard specification for message passing. It defines a model of distributed computing where each process has its own local memory, and data is explicitly shared by passing messages between processes. MPI includes point-to-point and collective communication routines, as well as support for process groups, communication contexts, and application topologies [155]. MPI has now emerged as the de facto standard for message passing on computer clusters [78].

Besides the distributed computing libraries, programming languages specifically for implementing distributed applications are also created [16]. Unlike sequential programming languages, distributed programming languages aim to address problems such as concurrency, communication, and synchronization in distributed computing from the language level to simplify distributed programming. They generally provide built-in primitives for task distribution, communication, and fault-tolerance. Representative languages included X10 [41], Haskell [159], JoCaml [66], and Erlang [8].

2.2.2 *General-Purpose Distributed Processing Frameworks.*

Although distributed computing libraries and languages enable efficient distributed computing, the applications still put lots of effort into the details of running a distributed program, such as the issues of parallelizing the computation, distributing the data, and handling failures. To address this complexity, general-purpose distributed processing frameworks are designed.

MapReduce [56] is a simple and powerful framework introduced by Google for scalable distributed computing, simplifying the development of applications that process vast amounts of data. It takes away the complexity of distributed programming by exposing two processing steps that programmers need to implement: Map and Reduce. In the Map step, data is broken down into key-value pairs and then distributed to different computers for processing, and in the Reduce step, all computed results are combined and summarized based on key. MapReduce and its variants [166] reduce the complexity of developing distributed applications. However, MapReduce reads and writes intermediate data to disks, which incurs a significant performance penalty. To address the deficiency of MapReduce, Spark [176] shares a similar programming model to MapReduce but extends it with a data-sharing abstraction called resilient distributed dataset (RDD) [175], which can be explicitly cached in memory and reused in multiple MapReduce-like parallel operations. Due to the introduction of RDD, Spark achieves significant performance improvement compared to MapReduce [177]. In addition to MapReduce and Spark, other general-purpose distributed processing frameworks are also designed, such as Storm [158], Flink [32], and Ray [124]. [53, 136, 146] provide comprehensive surveys on general-purpose distributed processing frameworks.

2.2.3 *Distributed Graph Processing Frameworks.*

General-purpose distributed processing frameworks facilitate efficient distributed computing by isolating the applications from the lower-level details. However, the general-purpose distributed processing frameworks are inefficient in processing big graph data due to the skewed degree distributions of graph data and poor locality during the computation [83]. As a consequence, distributed graph processing systems started to emerge with Google's Pregel [118] for better performance and scalability.

Pregel [118] is based on the Vertex-Centric programming model in which the graph computation is divided into a series of synchronized iterations, each of which is called a superstep. During a single superstep, each vertex executes a user-defined function in parallel. This function typically processes data based on the vertex's current state and the messages it received from other vertices [94]. Besides Pregel, representative Vertex-Centric graph processing framework includes GraphLab [109], PowerGraph [76], GPS [137], Giraph [81], and Pregel+ [171].

Beyond the Vertex-Centric based framework, the Edge-Centric and Subgraph-Centric based distributed graph processing frameworks have also been developed. The Edge-Centric model, pioneered by X-Stream [134] and WolfGraph [181], focuses on edges rather than vertices, offering better performance for non-uniform graphs by streamlining edge interactions and avoiding random memory access [83]. Despite its programming complexity, the Edge-Centric model boosts performance for algorithms that require pre-sorted edge lists [134, 181]. The Subgraph-Centric model concentrates on entire subgraphs. This model minimizes communication overhead, making it ideal for tasks where frequent vertex interaction is crucial, such as in social network analysis. Frameworks like Giraph++ [156], GoFFish [147], and Blogel [170] adapt this model. Despite these frameworks' advancements, distributed graph algorithms still face challenges like parallelism, load balance, communication, and bandwidth due to the complexity and irregularity of graph data, as discussed in Section 1.

3 Distributed Graph Processing: Challenges and Solution Overview

Distributed graph processing is able to handle graphs of very large scale via interconnected computers. However, the shift from single-machine computing to distributed computing introduces challenges arising from the inherent characteristics of distributed systems and graphs, making them essential considerations in the design of distributed graph algorithms. In this section, we conduct a systematic analysis of the inherent challenges in distributed graph processing (Section 3.1) and provide an overview of existing solutions (Section 3.2).

3.1 Inherent Challenges in Distributed Graph Processing

In a distributed system, comprising multiple interconnected machines, each serves as an independent computational unit, often dispersed across different locations. This setup, as depicted in Figure 3, harnesses collective computing power for efficient data processing. However, it also introduces significant challenges in computation and network resource utilization, aspects that are particularly critical in the context of distributed graph processing.

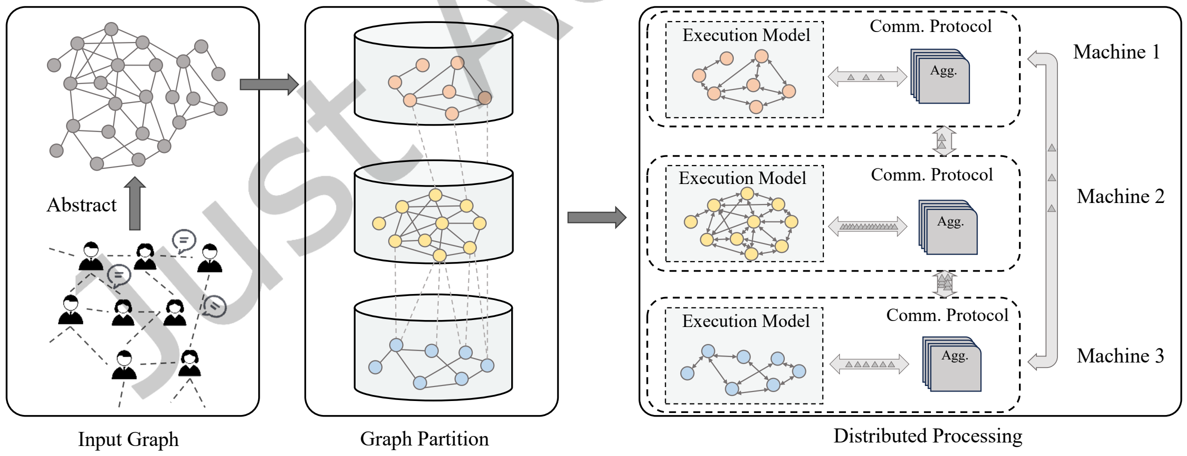


Fig. 3. The framework of the distributed graph processing. The graph is divided into several partitions that are assigned to different machines. Vertices in different partitions require a global communication to send messages while vertices in the same partition can exchange messages by local communication. The volume of one communication has a limitation called bandwidth, indicating the largest size of a single message.

Computation Resource Efficiency: Distributed systems are characterized by their vast and scalable computation resources, which enable the system to process massive volumes of graph data and execute complex graph computations. Therefore, it is of great importance to fully exploit the computation resource in the system when designing distributed graph algorithms. Different from the centralized graph algorithms in which all the instructions are executed in a single machine, distributed graph algorithms need the collaboration and cooperation of multiple machines to finish a task, which brings the challenges of *parallelism* and *load balance*.

- *Parallelism*: Parallelism in distributed graph processing involves executing multiple computations concurrently across various machines. This approach entails segmenting a larger graph analysis task into smaller, manageable subtasks. These subtasks are then distributed among different machines, enabling them to be executed simultaneously. Such a strategy not only facilitates efficient resource utilization but also significantly reduces the overall computation time, thereby enhancing the performance of graph processing tasks. However, graph analysis tasks often exhibit inherent sequential dependencies [3, 88, 179], making the realization of parallelism in distributed graph algorithms a complex endeavor. A profound understanding of the fundamental nature of these tasks is essential to discern independent subtasks that can be parallelized effectively. This necessitates a careful analysis to strike a balance between preserving the integrity of the sequential dependencies and optimizing for parallel execution.
- *Load Balance*: Load balance in distributed graph processing ensures that the computational workload is evenly distributed across all machines. An imbalance in load can lead to inefficiencies: some machines may complete their tasks quickly and remain underutilized, while others (often known as stragglers) struggle with ongoing computations, ultimately delaying the overall process. This imbalance is particularly problematic in distributed graph processing, where the irregular nature of computations arises from non-uniform [50] degree distribution and topological asymmetry. Albeit crucial, effectively resolving load imbalance is complex. It demands not only precise initial workload quantification but also ongoing adjustments during runtime to address any imbalances.

Network Resource Efficiency: In distributed systems, where machines communicate via networks, efficiently using network resources becomes vital, particularly in graph processing. The inherent complexity of graph data, marked by intricate structures and irregular vertex connections, often requires operations on a single vertex to interact with multiple others. This scenario leads to frequent and extensive network data exchanges especially when interconnected vertices are spread in different machines. Consequently, two main challenges arise in terms of network resource efficiency.

- *Communication Overhead*: Communication overhead in distributed systems, defined by the network resource usage from message exchanges, is largely dependent on data transmission volumes. In distributed graph processing, the necessity to communicate across machines for accessing vertices or edges on different machines increases network communication. Inefficient management of these data exchanges can lead to significant network congestion, turning network communication into a critical bottleneck for overall computing performance. Thus, managing communication overhead is crucial for optimizing the efficiency and effectiveness of distributed graph processing.
- *Bandwidth*: Bandwidth in distributed systems represents the maximum data transfer capacity between machines in each round of message passing. Limited by hardware and network infrastructure, bandwidth is not infinitely scalable. In distributed graph processing, vertices with high degrees demand substantial bandwidth due to intensive communication with neighbors [34] and frequent visits from multiple vertices, common in random walk algorithms [111]. Additionally, low bandwidth utilization is also a challenge. For tasks, like triangle counting, BFS, and connected components, numerous small messages are exchanged between low-degree vertices, containing only neighbor information. On the other hand, each message exchange using the message-passing interface, like MPI, introduces additional overhead in the form of header information and handshake

protocol messages resulting in a diminished ratio of actual payload data and thereby leading to an inefficient utilization of bandwidth resources [149]. For these reasons, effectively and efficiently optimizing bandwidth utilization is challenging in distributed graph processing.

Remark. Besides the above challenges, there are still other challenges like fault tolerance [80, 180] that need to be considered in distributed graph processing. However, distributed graph processing often addresses them following the established solutions for general distributed algorithms in the literature. Therefore, we focus on the above four challenges in this survey and omit other challenges (Interested readers can refer to [47, 120, 154, 161] for the solutions to these challenges).

3.2 Solution Overview

Following our analysis of the inherent challenges in distributed graph processing in Section 3.1, this section summarizes various solutions developed to address these challenges, particularly in the field of distributed graph processing, and provides an overview of the common techniques in the detailed algorithms presented in Section 4.

3.2.1 Computation Resource Efficiency Optimization. This section focuses on the solutions to optimize the computation resource efficiency including parallelism and load balance.

Parallelism: Breaking down the graph analysis task into smaller subtasks that can be processed independently reduces the overall execution time by allowing multiple computations to proceed concurrently in the distributed system. To achieve an effective parallelism, various techniques, including *dependency resolution*, *asynchronous execution*, and *shortcut augmentation*, have been proposed.

- *Dependency Resolution*: Graph analysis tasks generally have an intrinsic dependency order on the computation. For example, the order to visit the vertices in the graph traversal issues is specific, and any algorithms on the graph traversal have to follow this order to ensure correctness. The implied dependency order significantly limits the potential parallelism of the distributed graph algorithms. Dependency resolution focuses on identifying and exploiting the inherent dependencies among subtasks in the original graph analysis task to ensure correct and efficient parallel execution. Given the dependency order is unique to each specific graph analysis task, the dependency resolution methods also vary accordingly. The specific details on these dependency resolution methods are presented in Section 4 [37, 48, 52, 86, 88, 105, 107, 120, 123].
- *Asynchronous Execution*: Synchronization (Figure 4 (a)) is a coordination mechanism used to ensure that multiple concurrent processes in distributed graph processing do not cause data inconsistency and are able to successfully complete the task without conflict. In distributed graph processing, synchronous execution is the easiest synchronization mechanism in which all computations are divided into discrete global iterations or super-steps. During each super-step, all machines perform their computations simultaneously locally, and at the end of each super-step, there is a barrier synchronization point where all machines must wait until every other machine has finished its computations and communications [9, 122]. This can ensures consistency but often lead to poor computation performance due to waiting times caused by slower machines or stragglers.
  Asynchronous execution (Figure 4 (b)), instead, allows machines to operate independently without waiting for each other at barrier synchronization points. Once a machine finishes its computation, it immediately sends its updated information to its neighbors and proceeds with the next round of computation. This approach can significantly speed up the computation process as there's no idle waiting time, and it can also converge faster in certain scenarios. Therefore, many distributed graph algorithms [82, 121, 133, 179] exploit the asynchronous execution to enhance parallelism.
- *Shortcut Augmentation*: Shortcut augmentation aims to reduce the number of communication rounds required by adding additional edges, known as shortcuts, to the original graph. These shortcuts serve as direct links between distant vertices and enable more efficient information exchange among them. By introducing shortcuts

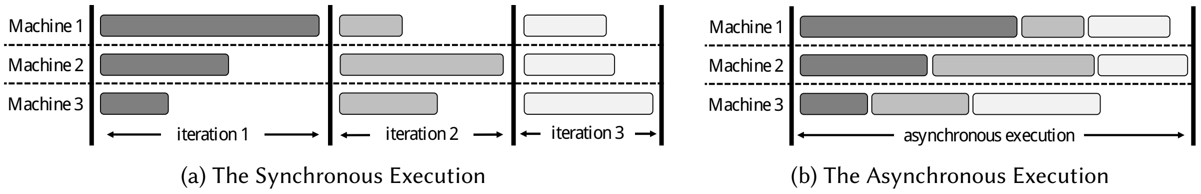


Fig. 4. An example of synchronous and asynchronous execution: In (a), vertical lines represent synchronous signals, where each machine must wait until all other machines have completed their workloads. In (b), asynchronous execution involves no waiting time, allowing each machine to compute independently, thereby increasing overall parallelism.

between vertices that are far apart, the algorithm can effectively reduce the number of rounds required to disseminate information or perform computations on the graph. This reduction in communication rounds leads to significant improvements in both time and resource utilization. Shortcuts augmentation optimization is widely used in path-related algorithms, in which shortcuts can significantly reduce the length of a path [31, 113, 140], and thus achieve faster convergence of the algorithm.

Load balance: Uneven workload distribution is a critical challenge encountered in distributed graph computing, as it often leads to load imbalance and hampers optimal performance. In the literature, *graph partitioning* and *task scheduling* have emerged as the primary approaches to address the load balance challenge.

- *Graph Partitioning*: Partitioning-based methods aim to distribute the graph data evenly by estimating the computational requirements of individual vertices and edges and partitioning them in advance. By carefully assigning vertices and edges to different machines, these methods strive to achieve load balance and improve overall system efficiency. Partition-based methods (Figure 5) can be further divided into three categories: (1) Vertex partitioning which divides the graph into subgraphs by assigning vertices to the different partition sets while minimizing edge cuts concerning load balance constraint. (2) Edge partitioning divides the graph into subgraphs by assigning edges to the different partition sets while considering a maximum load balance and minimum vertex cut. (3) Hybrid partitioning exploits the interior structure, such as vertex in-degree, the vertex out-degree, the degeneracy number, and the core number, of the graph to perform partitioning for a better-balanced workload [21, 35, 36, 72, 73, 79, 106, 117, 127, 142, 150, 168, 174, 179]. Moreover, recent algorithms also adopt the dynamic re-partitioning strategy in which the graph is dynamically redistributed across machines to adapt to changes in the graph analysis workload or resource availability to further improve the load balance [4, 47, 89, 162].

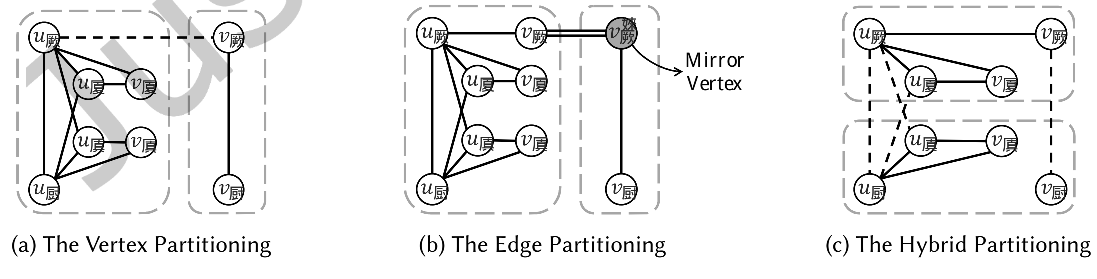
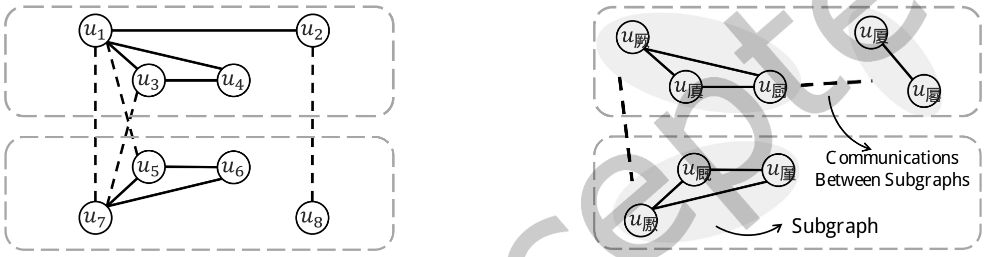

Mirror Vertex
(a) The Vertex Partitioning
(b) The Edge Partitioning
(c) The Hybrid Partitioning

Fig. 5. An example of partitioning methods: (a) vertex partitioning with 1 edge-cut, (b) edge partitioning with 1 vertex-cut (requiring mirror vertices for synchronization). Minimum-cut can cause unbalanced partitions. The hybrid method (c) with 3 edge-cuts considers both cut numbers and vertex distribution for better balance.

- *Task Scheduling*: Scheduling-based methods focus on dynamically scheduling the computation resources (e.g., processors). By monitoring the workload and resource availability in real-time, these methods enable the system to dynamically distribute workload and adjust resource allocation across machines as the computation progresses, aiming to maintain load balance and optimize the utilization of computation resources as a whole. A typical task scheduling implementation generally encompasses a task queue, responsible for holding pending tasks available for execution, and a task scheduler that selects the subsequent task for execution using diverse policies [44, 172].

3.2.2 Optimizing Network Resource Efficiency. This section focuses on the solutions to address communication overhead and bandwidth challenges regarding network resource efficiency.

Communication Overhead: In distributed graph processing, vertices in different machines frequently exchange messages, resulting in substantial communication overhead. To address this challenge, researchers have devoted considerable efforts to developing effective solutions to optimize communication overhead, which are categorized as follows:



(a) Vertex-Centric Model (b) Subgraph-Centric Model

Fig. 6. An example of the Subgraph-Centric model is shown in (b). Compared to the Vertex-Centric model in (a), where communication occurs between vertices, in (b), local computations are carried out within subgraphs until convergence, and then communication only occurs between subgraphs.

- *Local Computation*: By exploiting the inherent locality of the graph analysis task, local computation aims to perform the computation primarily based on locally available information at each machine to avoid message exchange to reduce communication overhead [51, 141, 142]. To achieve this goal, a widely adopted solution is to aggregate information from neighbors in advance, and thus accessing the information about these neighbors only requires local communication or local shared memory [30]. Moreover, Subgraph-Centric computation (Figure 6) [14, 100, 105, 117, 123, 144] is another popular local computation variant to reduce communication overhead. Subgraph-centric computation decomposes the graph into smaller subgraphs. Each machine in a distributed system is tasked with processing a specific subgraph independently, leveraging local information and neighboring subgraphs' data as necessary. This localized approach minimizes the need for frequent expensive inter-machine communication.
- *Selective Activation*: Selective activation solution selects a subset of vertices to be activated in each iteration, thereby mitigating the need for extensive communication across all edges during the distributed graph processing. Two prevalent methods for selective activation have been proposed in the literature. The randomized method activates vertices according to certain probabilistic criteria following the intuition that deleting one edge from the graph rarely changes the required proprieties such as shortest path distances [139] and cycle-freeness [67], which effectively prevents unnecessary communication on inactive vertices [55]. The deterministic method often adopts an activating threshold, whereby a vertex only sends messages when it

satisfies a specific condition based on the threshold [37, 110, 123, 165]. By leveraging these selective activation solutions, distributed graph algorithms can effectively reduce the communication overhead.

- *Hybrid Push-Pull Model*: The *push* and *pull* are two fundamental communication paradigms in distributed graph processing. In the *push* model, each vertex sends its updating messages or partial computation results to its neighboring vertices, while in the *pull* model, vertices retrieve information from their neighbors as needed to complete their computations. Generally, these two paradigms each have their own merits regarding communication efficiency in distributed graph processing. For instance, the *push* model might be a better choice for the initial stage with fewer active vertices, while the *pull* model might perform better on few inactive vertices [30, 133]. To combine the strengths of both models to reduce the communication overhead, hybrid solutions that automatically switch between the *push* model and the *pull* model are proposed [36, 108].
- *Aggregation*: Aggregation aims to reduce the amount of data transmitted between machines by merging multiple data elements and eliminating duplicates or redundant elements. A prominent method is to aggregate messages locally before sending them to other machines, wherein only essential information is retained. This method not only reduces the overall communication volume but also effectively diminishes the rounds of communications required [63, 138]. In addition, some algorithms [35, 101] focus on simplifying the graph structure by aggregating the information of vertices/edges within a certain area where communication primarily occur between these vertices/edges. This simplification facilitates the communication between these aggregated vertices/edges, thereby significantly alleviating the overall communication overhead.

Bandwidth: Frequent data exchange between machines also poses significant strain on the available network bandwidth. To mitigate the requirement of bandwidth in distributed graph processing, various solutions are designed in the literature, including *message coordination* and *dense representation*.

- *Message Coordinating*: In distributed graph processing, a vertex may receive a substantial number of messages within a single iteration and the available network bandwidth cannot accommodate the simultaneous transmission of these messages to the neighbors, message coordinating methods prioritize the messages based on certain criteria such that a message with potentially better result or a farther destination is assigned higher priority while less promising messages may be pruned by later updates, and transmit the messages based on their priorities accordingly [86, 88, 111, 113]. Message coordinating allows for more efficient utilization of available network resources and helps mitigate the impact of bandwidth limitations on distributed graph processing systems.
- *Message Buffering*: Batch processing is commonly used to tackle the issue of low bandwidth utilization in distributed graph processing. It utilizes a message buffer queue within each machine, wherein messages intended for transmission are temporarily stored. Once the buffer reaches its capacity, all accumulated messages are combined into a single batch and subsequently transmitted to other machines. Although this approach may introduce a delay in communication, it yields significant benefits in terms of optimizing bandwidth resource utilization and reducing the processing time associated with sending and receiving messages [72, 73, 138, 149, 161].

4 Distributed Graph Tasks

In Section 3, we have explored the inherent challenges in implementing distributed graph algorithms, alongside a discussion of their respective solutions. This section builds upon that foundation, delving deeper into the distinct challenges and solutions pertinent to each algorithmic topic, as introduced in Section 2. Utilizing the graph (http://gsp.blue/querying?name=graph_algo) constructed for this survey, we can analyze how challenges are resolved within a given topic, e.g. "`Centrality`", using the query: `CALL get_challenge_papers_hist_of_topic("Centrality")`.

The aggregate results of this analysis are depicted in Figure 7. Our objective is to systematically examine how these challenges differ among the various topics and to critically assess the particular methodologies adopted in existing literature to tackle these challenges.

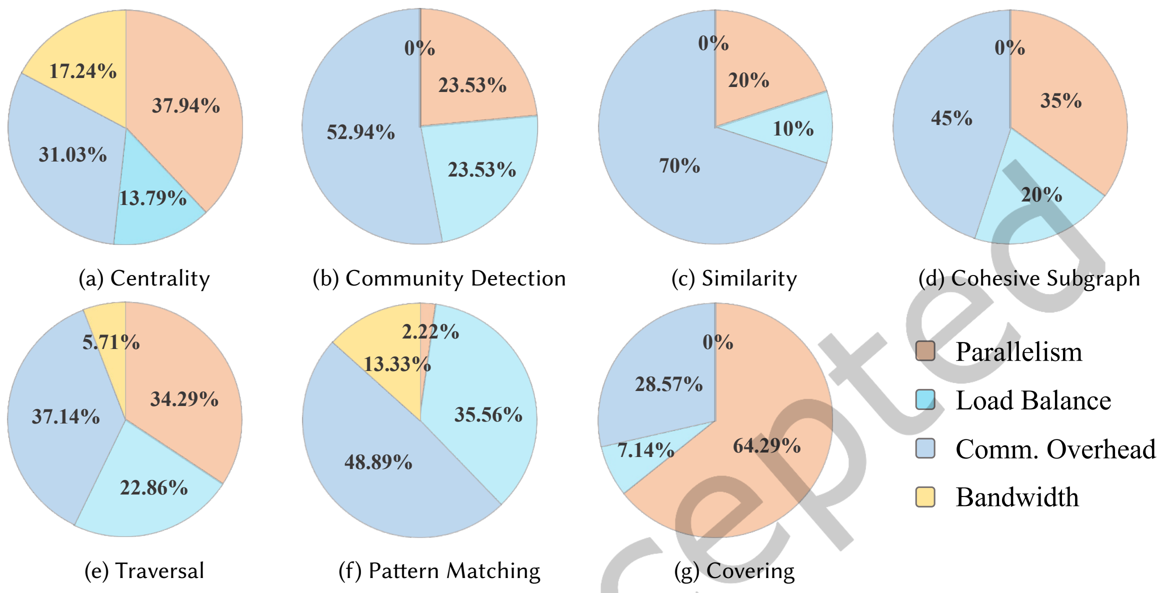


Fig. 7. The distribution of challenges resolved across various topics

4.1 Centrality

Centrality measures in graph theory are instrumental for quantifying the relative importance of vertices within a complex network. PageRank and personalized PageRank are relatively easy to parallelize as they only require neighbor information for computation. In contrast, Betweenness centrality and closeness centrality assess vertex influence based on shortest paths, requiring global graph information. This complexity poses challenges for designing distributed algorithms. Due to these characteristics, the latter is more suitable for identifying critical nodes that control the flow of communication or resources in networks compared to the former.

As Figure 7(a) illustrates, parallelism in centrality tasks is a popular research focus (37.93%). This is primarily due to the inherent independence of centrality computations for each vertex, which lends itself well to parallel processing approaches. Surprisingly, load balancing—a fundamental challenge in distributed environments—has garnered limited attention in this domain, accounting for only 13.79% of the research. This could be attributed to two potential factors. Firstly, the computational requirements for centrality calculations on each vertex are relatively straightforward, such as the summation process in PageRank algorithms. Secondly, in cases where the centrality task is complex, such as in calculating betweenness centrality, the research community may prioritize developing efficient parallelization techniques over addressing load balancing issues. A significant portion of the centrality research, precisely 31.03%, is devoted to minimizing communication overhead, while 17.24% of the studies aim to address bandwidth constraints.

PageRank, initially conceptualized by in [26], is designed to assess the importance of webpages, represented as vertices in a web graph, interconnected by hyperlinks symbolizing the edges. The PageRank score, symbolized by π, is derived from the stationary probabilities of an α-decay random walk. This process starts from a vertex selected uniformly from all n vertices at random. Subsequently, at each step, there exists a probability α of initiating a new random walk from a random vertex, and a complementary probability $1-\alpha$ of transitioning to an adjacent vertex. The PageRank score of a vertex u, denoted as $\pi(u)$, quantifies the relative importance of vertex u within the web graph, which can be formulated as $\pi(u) = (1-\alpha) \cdot \frac{1}{n} + \alpha \cdot \sum_{v \in N_{in}(u)} \frac{\pi(v)}{N_{out}(v)}$. Typically, algorithms for computing PageRank can be categorized into three distinct types:

(1) Iterative algorithms: Examples include PCPM [100] and SubgraphRank [14]. These algorithms iteratively update PageRank values until converge.

(2) Random-walk-based algorithms: This category comprises algorithms including IPRA [140], RP [113], MRP [111, 113], and FP [112], which generate and simulate multiple random walks and estimate PageRank values through the destination vertices of these walks.

(3) Eigenvector-based algorithms: The PageRank values as fixed points π for all vertices represent an eigenvector. Rather than directly using the original equation, the Stochastic Approximation (SA) algorithm [82] iteratively approximates the stable state π by constructing a differential equation, enabling gradual convergence towards the solution.

Optimizing parallelism is crucial, especially in random-walk-based algorithms. These algorithms generate multiple independent random walks to approximate PageRankvalues, benefiting significantly from parallel processing. Specifically, IPRA introduces a method using short walks as efficient indices in the graph, reducing the number of processing rounds needed. Similarly, MRP starts with short walks that are then combined to form longer walks, further enhancing efficiency. In contrast, the eigenvector-based SA algorithm employs asynchronous computation. It updates PageRankvalues by selecting individual vertices in each iteration, allowing subsequent iterations to proceed asynchronously as long as they do not update the same vertices simultaneously.

As a prevalent challenge in distributed computing, load balancing can impact the efficiency of PageRank algorithms. The computation of a vertex's PageRank is influenced by its connectivity, resulting in vertices with higher degrees engaging in more intensive operations of PageRank values. To mitigate this imbalance, iterative algorithms like PCPM and SubgraphRank employ the Subgraph-Centric computation model, which divides the graph into multiple partitions, each managed by a dedicated thread. However, balancing the workload, which includes both internal computation and global communication, remains complex. They use tools like OpenMP for dynamic scheduling of vertex computation, rather than adhering to an ideal static partitioning.

PageRank algorithms face significant challenges in minimizing communication overhead. A major advantage of this model is its ability to replace inter-partition vertex communication with local memory access. Additionally, it allows multiple communications from one vertex to all destination vertices within the same partition to be consolidated into a single data transfer, further reducing communication overhead. The SA algorithm optimizes communication costs by using the *pull* mode. In each iteration, a selected vertex updates its PageRankvalue by "pulling" information from its neighbors. This eliminates the need for the vertex to "push" its updated PageRankvalue to others, thereby halving communication overhead.

Bandwidth constraints can pose challenges, particularly in random walk-based algorithms, where a proliferation of random walks may converge on a single vertex, creating a bottleneck. To mitigate this, both RP and MRP algorithms employ a coordinator mechanism to evenly distribute walks among all neighboring vertices, thus preventing concentration on a single vertex. In contrast, the FP algorithm adopts a more bandwidth-efficient approach by transmitting only the expected number of paths, rather than sending individual path messages.

Personalized PageRank modifies the standard PageRank algorithm by enabling users to specify a start vertex, denoted as s. Unlike traditional PageRank , where the random walk commences from and restarts at a randomly selected vertex with probability α, personalized PageRank consistently initializes and restarts the walk at the specified source vertex s.

Given the similarities between personalized PageRank and the standard PageRank algorithm, most techniques for addressing distributed computing challenges are comparable. For instance, the Subgraph-Centric computation model [79] and shared-memory approaches [148] effectively reduce internal communication within partitions. Additionally, employing a coordinator between two partitions can significantly minimize redundant global communications, as demonstrated in [79].

We highlight techniques employed in personalized PageRank to achieve optimal load distribution. The study [79] proposes a theoretically perfect partition utilizing hierarchical methods. Moreover, load imbalance can also arise during sampling in weighted graphs, where the next hop in random walks is uneven, necessitating weighted sampling. The Alias method, capable of constant-time sampling for weighted elements, and its extension, the Alias Tree, are effective solutions for handling extremely large numbers of neighbors in memory-limited scenarios [106].

Betweenness Centrality measures the extent to which a specific vertex lies on the shortest paths between other vertices within a graph. The underlying premise is that important vertices are more likely to be on many of these paths, influencing communication flow across the graph. Given $C_b(u)$ as the normalized betweenness centrality value for a vertex u, most algorithms [52, 86, 88, 90] follow the Brandes' algorithm [25] to compute $C_b(u)$ as: $C_b(u) = \frac{1}{(n-1)(n-2)} \sum_{s \in V \setminus \{u\}} \delta_{s\bullet}(u)$, and $\delta_{s\bullet}(u) = \sum_{t \in V \setminus \{s,u\}} \frac{\sigma_{su} \cdot \sigma_{ut}}{\sigma_{st}}$, where σ_{st} is the number of shortest paths from a vertex s to another vertex t. The value of $\delta_{s\bullet}(u)$ can be recursively computed as: $\delta_{s\bullet}(u) = \sum_{u \in P_s(w)} \frac{\sigma_{su}}{\sigma_{sw}} \cdot (1 + \delta_{s\bullet}(w))$, where $P_s(w)$ is the set of predecessors of w in the SSSP Directed Acyclic Graph (SSSP DAG).

To compute betweenness centrality values of all vertices, Brandes' algorithm comprises two primary steps: (1) SSSP Step involves constructing a Directed Acyclic Graph (DAG) from each vertex in the graph; (2) Brandes' Step recursively computes the dependency score $\delta_{s\bullet}(u)$ and systematically accumulate these scores to determine the value of u.

In adapting Brandes' algorithm for distributed computing, a key consideration is parallelizing the computation across n SSSP DAGs to enhance parallelism. This involves executing the SSSP and Brandes' steps concurrently from all vertices. In unweighted graphs, BFS is used for this parallel execution, computing σ_{su} and $\delta_{s\bullet}(u)$ in the BFS order [88, 90]. For weighted graphs, alternatives like the Bellman-Ford [29] and Lenzen-Peleg [102] algorithms are employed [52, 86].

The communication demands of the SSSP step, especially on weighted graphs, are high due to frequent message passing. To mitigate this, the algorithm developed by Crescenzi et al. [52], which is based on Bellman-Ford, updates σ_{su} and $\delta_{s\bullet}(u)$ with differential values instead of full recomputation. Alternatively, the Lenzen-Peleg-based algorithm in [86] adopts an early termination strategy called Finalizer to stop the SSSP step as long as all vertices have received correct σ_{su}, which is shown to be effective when graph diameters are small.

Addressing bandwidth limitations involves coordinating the order of message sending. In the SSSP step, messages concerning shorter distances are prioritized [86], as longer distances are more likely to change. In the Brandes' step, priority is reverted to longer distances due to their greater number of hops to the source, as discussed in [88]. Given the inherent difference between the two steps, the algorithm in [86] attempts to track the round numbers at which updates occur in the SSSP step and then apply these in reverse order during the Brandes' step. Additionally, Pebble BFS [87], particularly effective in parallel SSSP for unweighted graphs, adopts a staggered BFS approach. The algorithm [88] utilizes the idea of visiting one vertex (instead of multiple vertices simultaneously) per round, thereby avoiding message congestion to some large-degree vertices.

Closeness Centrality quantifies the average shortest path length from a given vertex to all other vertices in the network, suggesting that a vertex of higher importance typically exhibits shorter average distances to other vertices. The normalized closeness centrality value $C_c(u)$ for a vertex u is calculated as $C_c(u) = \frac{n-1}{\sum_{v \in V} d(u,v)}$, where $d(u, v)$ is the shortest distance from u to v.

The calculation of closeness centrality values, much like the betweenness centrality, confronts considerable communication overhead. This is particularly severe in incremental closeness centrality, where the introduction of new edges necessitates the reevaluation of closeness centrality values. The STREAMER algorithm [139] effectively addresses this by utilizing a level structure to filter out edges that do not impact shortest path distances, which is a common scenario in real-world graphs. To further refine the process, the DACCER algorithm [164] employs an approximation of the "relative rank". This is based on the observation that vertices with higher closeness centrality tend to have larger neighborhoods with higher-degree vertices. By aggregating degree information from neighbors, each vertex can estimate its closeness centrality rank with reduced communication.

4.2 Community Detection

Community detection aims to partition vertices in a graph into different communities where vertices within a community are more closely linked to each other than to external vertices. Louvain optimizes modularity through hierarchical clustering, offering effectiveness at the cost of computational efficiency. In contrast, label propagation iteratively assigns labels based on neighbor majority voting, providing efficiency but potentially leading to unstable community detection, suitable for real-time scenarios. The connected components algorithm identifies subgraphs where all vertices are connected directly or indirectly, helping to understand the graph's basic structure.

As Figure 7 (b) illustrates, a notable proportion of research (23.53%) focuses on enhancing parallelism in community detection algorithms. The computation of a vertex's community state primarily relies on information that is local to the vertex, making the process inherently amenable to parallel execution across multiple processors. Load balancing is another critical aspect, with many studies (23.53%) aiming to resolve the skewed workload associated with high-degree vertices. The primary challenge in community detection, tackled by over half of the research (52.9%), is minimizing communication overhead. Despite the frequent exchanges between vertices in community detection algorithms, where each piece of communicated data is generally small, our survey reveals no studies have specifically tackled bandwidth limitations in community detection.

Louvain is widely regarded as one of the most popular methods in this topic [20]. Central to this algorithm is the concept of modularity, denoted by Q, which serves as a measure to assess the quality of the identified communities within a graph. The modularity is given by the equation: $Q = \frac{1}{2m}\sum_{i,j}\left[A_{ij} - \frac{k_i k_j}{2m}\right]\delta(c_i, c_j)$, where A_{ij} represents the weight of the edge connecting vertices i and j, $k_i = \sum_j A_{ij}$ is the sum of the weights of all edges attached to vertex i, c_i denotes the community to which vertex i belongs, $\delta(u, v)$ outputs 1 if $u = v$ and 0 otherwise, and $m = \frac{1}{2}\sum_{ij} A_{ij}$ is the sum of weights of all edges.

At the beginning of the algorithm, each vertex is treated as a separate community. It then proceeds iteratively, examining each vertex i and evaluating the gain in modularity that would result from moving i to a neighboring community j. The vertex i is placed in the community that yields the highest increase in modularity. The process of moving vertices between communities is repeated until the modularity no longer increases.

Load balancing plays a critical role in the efficiency of Louvain algorithms in distributed environments, since the workload of a single machine is typically proportional to the degree of the vertices it processes, thus a few high-degree vertices can lead to significant skew in the workload distribution. Jianping Zeng, et al. [178] extend the graph partitioning method based on vertex delegates, initially proposed by Pearce et al. [128]. The method described involves duplicating high-degree vertices across all machines in a distributed system. The edges connected to these high-degree vertices are then reassigned to ensure a balanced distribution of edges, and consequently, computational work, across the different machines in the system. Another distributed algorithm, called DPLAL [142], employs the well-known graph partitioning tool METIS [95], which is designed to minimize the number of cut edges or the volume of communication needed between different parts of the graph.

Reducing communication overhead is also a prevalent challenge in Louvain algorithms. Some recent studies [74, 178] reduce this overhead by utilizing "ghost" vertices, which are a set of replicas of the vertices that are connected to local vertices but are processed by other machines. These ghost vertices help to maintain the local integrity of the graph, thereby reducing inter-machine communication. The paper [74] also presents two heuristics for optimizing the distributed Louvain algorithm, one is setting varying thresholds during different stages of iterations to decide termination, effectively reducing the total number of iterations during the algorithm; another is setting vertices with minor changes in modularity during iterations as "inactive", to save computational and communication resources. Moreover, some papers [141, 142] adopt the Subgraph-Centric model. Each machine collects information about adjacent vertices, then updates local community states independently, without inter-machine communication.

Label Propagation algorithm (LPA) [131] is an effective approach to detecting communities by propagating labels throughout the network. The algorithm starts by assigning a unique label to each vertex, which represents the community to which the vertex currently belongs. In each subsequent iteration, every vertex considers the labels of its immediate neighbors and adopts the label that is most frequent among them. The algorithm continues to iterate through the vertices and update their labels until convergence, i.e., when each vertex has the majority label of its neighbors or the maximum number of iterations defined by the user is reached.

Implementing distributed label propagation algorithms involves balancing parallelism with the quality of identified communities. Misordered computations can result in overly large, impractical communities. The EILPA algorithm [13] identifies influential vertices, termed "leaders", using a personalized PageRank model. Label propagation starts from these leader vertices, ensuring a structured, influence-based community detection. The PSPLAP algorithm [116] also considers vertex influence by assigning weights via k-shell decomposition and calculating propagation probabilities and vertex similarities based on these weights. Vertices update their labels during each iteration using these probabilities and similarities, aiming for more coherent community detection by reducing label updating randomness typical in label propagation algorithms.

Communication overhead is another prevalent challenge when implementing the label propagation algorithm in distributed environments. To tackle this challenge, Xu Liu, et al. [108] propose a dynamic mechanism that alternates between *push* and *pull* strategies to reduce communication costs. Initially, *push* is used for rapid community formation, and then the algorithm switches to *pull* in later iterations to consider neighbors' label quality.

Connected Components refer to subgraphs where every vertex can be reached from any other vertex. Computing connected components typically involves traversing all vertices and edges in a graph based on Breadth-First-Search (BFS) or Depth-First-Search (DFS), resulting in a linear time complexity. However, in distributed environments, the irregularity of the graph leads to centralized workload and communication in high-degree vertices, which are the primary focus for optimizing connected component computation.

For improving parallelism, the paper [132] leverages the max-consensus-like protocol where each vertex only considers the maximum value among its in-neighbors to update its own value. The algorithm can terminate in $D + 2$ iterations where D is the diameter of the graph. Benefiting from this protocol, each vertex only needs to receive information from its in-neighbors, which also helps to improve computational and communication efficiency.

Balancing workload is also important for improving the efficiency of connected component computation. In the paper of Sebastian Lamm, et al. [101], high-degree vertices are replicated to different machines to alleviate the load imbalance issues caused by processing such vertices on a single machine. Chirag Jain, et al. [89] adopt the strategy of dynamically redistributing vertices. Due to the vertex pruning strategy used, vertices may be removed from machines after each iteration, leading to load imbalance issues. In their work, after each iteration, some vertices will be redistributed to achieve load balancing.

To minimize communication overhead, selective activation strategies are adopted by a number of papers [89, 110]. CRACKER [110] identifies seed vertices to construct trees for each connected component. Vertices are iteratively removed from the graph and added to the tree, reducing communication volume by trimming vertices during this process. Another work [89] removes completed connected components in each iteration to decrease the number of active vertices in subsequent iterations. The paper [101] proposes a graph contraction pre-processing strategy. It conducts local BFS on each machine and merges vertices within the same connected component into a single vertex, avoiding multiple messages to common neighbors. In addition, Xing Feng, et al. [63] adopts the local computation strategy, where they first run BFS locally to label vertices in the same connected component with the same color. Then, inter-machine communication is carried out to merge subgraphs by merging different colors received by the same vertex. To further reduce the amount of communication, messages are locally aggregated before being sent to other machines.

4.3 Similarity

This topic refers to evaluating the similarity of the two vertices in a graph. Vertices with a high similarity score indicate that they have something in common in structure or attribute. Jaccard similarity and cosine similarity are basic measurement methods using set operations, requiring only neighboring information for computation, thus enabling high parallelization. In contrast, SimRank calculates similarity by considering the entire graph's information. Despite its lower efficiency, it can uncover latent relationships and similarities hidden within the graph structure.

According to Figure 7c, efforts are divided, with 20% focusing on enhancing parallelism and 10% on improving load balancing. This distribution underscores the challenge in similarity computation across all vertex pairs, necessitating extensive inter-vertex communication. Consequently, the predominant emphasis of research, at 70%, is on reducing communication overhead. It's noteworthy, however, that bandwidth constraints have not received significant attention in this topic.

Jaccard Similarity and Cosine Similarity are commonly used to measure the similarity of vertices. Jaccard similarity measures the similarity between two sets based on their shared elements. Specifically, it is defined as the size of the intersection of two sets divided by the size of their union, which is: $Jaccard(A, B) = \frac{|A \cap B|}{|A \cup B|}$. The value of Jaccard similarity ranges from 0 to 1, and if either A or B is an empty set, the value is 0. In the context of graph data, sets A and B are typically defined as the sets containing all the neighbors of a given vertex. cosine similarity is another measure of similarity between two sets, which is defined as: $Cosine(A, B) = |A \cap B| / \sqrt{|A| \times |B|}$.

Communication overhead poses a significant challenge, especially when computing similarity values between all vertex pairs, as each vertex is required to exchange information with nearly all others. WHIMP [145] calculates the incidence vector of vertices in the graph to identify all pairs whose cosine similarity values exceed a specific threshold. SimilarityAtScale [19] also adopts the method of compressing vectors to reduce the volume of communication between machines during the computation of Jaccard similarity values. It first divides the vector into smaller batches, filters out zero rows using distributed sparse vectors, and then converts segments of these rows into more efficient bit vectors for processing. Finally, a study [51] adopts a local computation strategy, decomposing the computational tasks and allocating them to different machines. In this process, there will be communication of the intermediate results for the local solutions.

SimRank is an algorithm based on the observation that "two objects are considered similar if they are referenced by similar objects" [91], i.e., the similarity of two vertices is determined by its neighbors' similarities. The formal definition is: $s(a, b) = \frac{C}{|I(a)| \cdot |I(b)|} \sum_{i=1}^{|I(a)|} \sum_{j=1}^{|I(b)|} s(I_i(a), I_j(b))$, where C is a constant between 0 and 1, and the set of in-neighbors of a/b is denoted as $I(a)/I(b)$, where $I_i(a)/I_j(b)$ represent the i-th/j-th in-neighbor of a/b. Note that $s(a, b) = 0$ when $I(a) = \emptyset$ or $I(b) = \emptyset$ and $s(a, b) = 1$ if $a = b$. Computing SimRank typically involves two primary algorithms:

(1) Iterative algorithm: The algorithm iteratively computes the SimRank scores according to its definition until convergence or a fixed number of iterations.

(2) Monte Carlo algorithm: It is a Random-Walk-based algorithm, where the SimRank score between two vertices is based on the expected first meeting time of random walkers starting from each vertex and traversing the graph in reverse.

Optimizing parallelism is significant, particularly for those algorithms that involve random walks. Each walk simulation is independent and suitable for parallel processing. However, the sequence-dependent nature of individual random walks, where the next vertex selected depends on the previous one, presents parallel processing challenges. DISK [162] adopts the approach of constructing forests instead of random walks, thus each vertex can independently build its next layer of child vertices, enhancing parallelism. In CloudWalker [104], the recursive computation of SimRank is transformed into solving a linear system by estimating a diagonal matrix, which represents attributes of each vertex. Subsequently, this linear system is solved using the Jacobi method, which eliminates the recursive dependencies when computing SimRank values.

To balance workload, DISK [162] declares two load thresholds W_{max} and W_{min}, and after each iteration, some vertices from high-load machines (larger than W_{max}) will be moved to low-load machines (smaller than W_{min}) to achieve dynamic load balancing.

Many papers [104, 115] adopt the method of data sharing for reducing communication overhead. CloudWalker [104] caches the intermediate results during the computation process, specifically computed by counting the landing positions of all walkers at some vertices, for saving the network communication cost. Besides, UniWalk [115] adopts a path-sharing method. Specifically, when computing the SimRank scores for multiple vertices, it allows the reuse of certain paths, thereby reducing the amount of data that needs to be communicated or processed. Another solution, pruning, is also used to reduce communication overhead. Siqiang Luo, et al. [114] adopt the method of truncating the length of each random walk and limiting the number of random walks to be sampled for accelerating computation and reducing the volume of communication. Another tree-based algorithm, DISK [162], builds the forest from leaves to roots, and the SimRank values are computed based on the overlapping leaf vertices. To optimize both computational and communication overhead, each vertex samples its in-neighbors with a certain probability during the forest construction.

4.4 Cohesive Subgraph

In graph theory, cohesive subgraphs are defined as parts of the original graph where vertices are densely connected with each other. In this survey, we focus on three popular cohesive subgraphs, k-core, k-truss, and maximal clique. The level of density they identify differs due to their different definitions, with k-Core < k-Truss < maximal clique. Correspondingly, their computational efficiency decreases in the same order, that is, k-Core > k-Truss > maximal clique. Therefore, each has its own applications depending on the required density and efficiency.

Computing these subgraphs, varying in density requirements, necessitates traversing all vertices or edges, posing significant challenges. As Figure 7 (d) shows, parallelism (35%) presents a challenge, especially in tasks like k-core and k-truss that require iterative updates. Load balancing, crucial in computationally demanding tasks like maximal clique, is the focus of 20% of studies, as imbalanced workloads can severely impact overall performance. Moreover, 45% of research focuses on reducing communication overhead due to frequent vertex-data communications. The simplicity of exchanged data results in bandwidth limitations being a minor issue.

k-Core of a graph, as defined in [119], is the maximal subgraph in which each vertex has at least a degree of k. The core value (i.e. coreness) of a vertex is the highest k for which it is part of the k-core. The k-core decomposition algorithm is usually used to compute k-core, which involves iteratively computing these core values for all vertices, starting with a specified k (typically the largest degree of the graph) and then decrementing k in subsequent iterations. The algorithm functions by "deleting" vertices (and the associated edges) with degrees less than the current k, while simultaneously updating the degrees of the remaining vertices. This process continues until all vertices are assigned a core value, marking the completion of the computation.

Parallelism poses a significant challenge in k-core decomposition algorithms, particularly within the "vertex-deletion" approach. In this method, the deletion of a vertex necessitates the sequential updating of its neighbors' degrees, which complicates the efforts of parallelization. To tackle this issue, a series of studies [37, 105, 120, 123, 165] have proposed a distributed k-core decomposition algorithm that decouples the "deleting" and "updating" phases. Initially, the deletion phase is parallelized, ignoring potential update conflicts. Then, an updating phase adjusts the degrees of vertices affected in the deletion phase. It is important to note, however, that this distributed version of the k-Core algorithm may require more iterations (than the sequential counterpart) to achieve convergence.

To minimize communication overhead, The Subgraph-Centric approach is widely adopted, as highlighted in [105, 123], as a natural fit for the distributed k-core decomposition algorithm as previously mentioned. Alongside the Subgraph-Centric model, the works in [123, 165] focus on reducing communication overhead using the Vertex-Centric model. In this approach, each vertex does not broadcast messages to all neighbors but selectively to those whose coreness values are anticipated to change. Moreover, T.-H. Hubert Chan, et al. [37] devised an approximate k-core decomposition algorithm that imposes an iteration limit. They demonstrated that, for any $\gamma > 2$, $\lceil \log n / \log(\frac{\gamma}{2}) \rceil$ iterations are sufficient to achieve an approximate core value for all vertices, at most $\gamma\times$ the actual value. Differing from the traditional "vertex-deleting" approach, this algorithm considers multiple thresholds in each iteration to expedite the process and removes vertices whose degree falls below the highest threshold. Furthermore, each vertex communicates only with a specific subset Λ of vertices, which effectively reduces the communication volume to $\log_2 |\Lambda|$ bits each iteration. Upon receipt of a message, a subroutine is invoked to adjust Λ if necessary.

***k*-Truss** is defined as a subgraph where each edge in the subgraph is part of at least $(k-2)$ triangles. Similar to the computation of a k-core, the process of determining a k-truss involves iteratively deleting edges that are part of fewer than $(k-1)$ triangles until convergence is reached. The time complexity of calculating a k-Truss is primarily constrained by the process of listing triangles, which, in the worst-case scenario, is $O(m^{1.5})$.

Similar to the computation of k-core, parallelizing the k-truss algorithm is challenging due to its inherently sequential nature. To address this, Pei-Ling Chen et al. [48] proposed a distributed k-truss algorithm that splits the computation into two phases: "deleting" and "updating", aiding in parallel execution management. Additionally, Yingxia Shao et al. [144] introduced PETA, which constructs a Triangle-Complete (TC) subgraph for each vertex to enhance the k-truss algorithm's efficiency within its partition, minimizing communication during triangle counting.

Research also dives into load balancing. Following the initial placement of all vertices and their neighbor lists, KTMiner [4] involves strategically reorganizing and redistributing the vertices. This approach ensures even workload distribution among all machines involved in the computation.

Communication overhead remains a significant challenge in k-truss computations. Pei-Ling Chen et al. [48] proposed an algorithm centered on the Edge-Centric model, which proves more apt for k-truss computations than the Vertex-Centric model. Their approach starts by transforming the original graph into a line graph, where each vertex represents an edge from the original graph. In this line graph, vertices are connected if their corresponding edges in the original graph share a common vertex. To further minimize communication overhead, they introduced a pruning strategy: in the line graph, two vertices are adjacent only if their corresponding edges in the original graph form a triangle with another edge. Moreover, KTMiner [4] enhances efficiency by pre-computing neighbor lists for each vertex and its neighbors, allowing triangles to be calculated locally. This method effectively eliminates the need for cross-machine communication between vertices.

Maximal Clique in a graph refers to a clique—a subgraph where each vertex is connected to all others—that cannot be expanded by adding another vertex. Enumerating maximal cliques is a local task, allowing each vertex to independently participate without requiring intermediate results from others. This feature makes it suitable for parallel processing. However, its computational complexity requires $O(3^{n/3})$ in the worst case [157], where n represents the number of vertices. Due to non-uniform subproblem sizes, and the unbalanced lengths of search paths in different subproblems, balancing the workload is a significant challenge many researchers focus on.

For workload balancing, Xu et al. [168, 169] initiate the process by sorting vertices using three distinct methods: degeneracy ordering, degree ordering, and core number ordering. Following this, the algorithm distributes the neighbor lists for each vertex. Specifically, a vertex v and its corresponding neighbor list are allocated to the machine responsible for processing v. For each neighbor u of v, if u appears after v in the ordering, the information about v is also transmitted to the machine that processes u. This method ensures that the data is shared efficiently between machines, enhancing the overall execution of the task. Another algorithm, PECO, proposed in [152], also adopts the strategy of sorting the vertices before distributing them to different machines. For sorting methods, degree ordering, triangle ordering, lexicographic ordering and random ordering are used and the experiment shows that the distribution of work is better with the degree ordering.

4.5 Traversal

Traversal, a fundamental method in graph analysis, involves visiting each vertex or edge in a graph to identify paths, trees, or cycles. This process is crucial for uncovering the structural information of a graph. Each traversal algorithm, such as single-source shortest path (SSSP) for finding the shortest path and maximum flow for capacity planning in flow networks, serves specific applications. However, these algorithms all require global updates and synchronization during computation, posing significant challenges in distributed environments.

As indicated in Figure 7e, the studies on parallelism focus on identifying sub-structures that have strong dependencies among vertices during traversal, which account for 34.29% of the research efforts in traversal. Additionally, about 22.86% of works consider load balancing, especially when the Subgraph-Centric model is adopted. A significant number of papers (37.14%) have contributed to the reduction of communication overhead in traversal: the inherent redundancy in visiting vertices and the need for transferring messages make it a communication-intensive task. Finally, There is little (5.71%) research specifically targeting bandwidth optimization.

Breadth-First-Search (BFS) is utilized to systematically visit all vertices starting from a specified source vertex, facing challenges due to irregular vertex access patterns despite its simplicity.

The nature of the workload in BFS processes, where a vertex becomes active only upon visitation, poses a challenge in maintaining load balance. To address this, BFS-4K [28] employs a dynamic thread scheduling strategy for child kernels to effectively balance the workload. Additionally, a delegate-based approach [30] involves assigning delegates to high-degree vertices in different machines. This method ensures that access to these vertices primarily involves local communication, thereby reducing the reliance on more costly global communication.

To address the challenge of communication overhead in BFS, particularly due to redundant communications with already visited vertices, a direct solution is to "mask" these vertices, omitting them from future iterations as detailed in [18]. Additionally, optimizing BFS based on its stages further reduces this overhead. Initially, when few vertices have been visited, a top-down (*push* model) is more effective. As the number of visited vertices grows, a bottom-up (*pull* model) approach becomes preferable to limit redundant communications. Identifying the ideal point to switch between these models is key, as explored in [30, 133].

Single Source Shortest Path (SSSP) aims to find the shortest path from a source vertex to all others. The discussions of SSSP are predominantly on weighted graphs. While the sequential Bellman-Ford and Dijkstra algorithms are hard to parallelize, there are four major methodologies for implementing SSSP in distributed environments:

(1) Subgraph SSSP: Applying Dijkstra's algorithm on subgraphs within a distributed framework and exchanging distance messages between partitions [117].

(2) Δ-stepping: The Δ-stepping method operates by selecting vertices within a specific distance range, denoted by $[k\Delta, (k+1)\Delta)$. This approach prioritizes the relaxation of shorter edges that fall below a threshold Δ until reaching convergence [36, 161].

(3) Skeleton: These algorithms [31, 43, 60, 64, 84], which offer faster processing with approximation, construct a "skeleton" graph (a subgraph) and compute "hopsets" to expedite finding the shortest paths. Skeleton-based algorithms compute distances from skeleton vertices to other graph vertices, adding as shortcuts into the original graph. These shortcuts can significantly decrease the number of hops (vertices) of a shortest path.

(4) Transshipment: A broader approach that generalizes the SSSP task, often modeled as a linear programming problem [17] or linear oblivious routing [135, 182] for approximating distances.

Skeleton-based algorithms focus on achieving higher parallelism. In these algorithms, parallelism largely depends on the quality of the "hopset". Utilizing an approximate "hopset" with a shorter hop distance (the number of hops required between vertices) has proven more practical [84]. CFR algorithms introduce a tradeoff parameter to balance hop distance, "hopset" size, and computation time [31]. Moreover, a two-level skeleton approach, which constructs an additional skeleton atop the original, achieves even greater efficiency, closely approaching theoretical limits [43].

In the context of optimizing load imbalance, Delta-stepping algorithms skillfully integrate aspects of Bellman-Ford and Dijkstra for distributed environments. Nevertheless, they are susceptible to load unbalancing, particularly when high-degree vertices are processed simultaneously, leading to excessive communications. A practical solution [36] involves distributing these vertices across different machines. For updating long edges, instead of transmitting sparse pairs of delayed distance messages, a more efficient approach [161] is to aggregate these as a dense distance vector. This uniform communication strategy not only streamlines the process but also aids in balancing the communication load across the system.

High communication overhead, a prevalent challenge in SSSP algorithms, results from extensive distance relaxing and updating processes. One approach to address this is dynamically switching between *push* and *pull* models at different stages, as proposed in [36]. Another optimization involves using a hierarchical sliding window instead of a fixed Δ-bucket to enhance communication efficiency [161]. The Subgraph SSSP algorithm [117] reduces communication costs by computing partial shortest path distances within individual partitions. In transshipment algorithms, gradient descent techniques are employed to minimize communication by optimizing update directions [17].

Minimum Spanning Tree (MST) is a connected subgraph that encompasses all n vertices and $n - 1$ edges, forming a tree. It is characterized by the minimum aggregated weight of its edges, representing the least costly way to connect all vertices in a graph. In distributed environments, many works are based on the GHS algorithm [69], which involves three steps: (1) each vertex starts as a single-vertex tree (fragment); (2) each fragment selects the lightest edge connecting it to another fragment, merging them; (3) reducing the number of fragments by at least half each iteration, achieving a single fragment representing the MST after at most $O(\log n)$ rounds.

Distributed MST algorithms, if not optimized, may incur significant communication overhead, particularly when a fragment's diameter greatly exceeds the graph's diameter. in a linear graph with n vertices where the first vertex connects to all others (resulting in a diameter of 1), choosing a line as a fragment increases its diameter to $n - 1$. If a line is chosen as a fragment, its diameter becomes $n - 1$. This scenario requires the GHS algorithm to perform $O(n)$ rounds for merging. The Controlled-GHS algorithm [70] addresses this by controlling fragment diameter growth. It merges fragments using specific methods when a fragment is small or few fragments remain, or by merging close fragments to minimize communication. Besides, some approaches focus on creating a low-diameter spanning tree first, then applying Controlled-GHS to maintain low-diameter growth [121].

Cycle Detection entails identifying cycles of a specified length in a graph. For smaller cycles, there are numerous effective algorithms. The task of finding a 3-cycle, commonly known as a triangle, typically involves checking for common neighbors between pairs of vertices on each edge. Similarly, detecting a 4-cycle, or square, usually requires examining common neighbors between pairs of vertices on all wedges (a wedge being a path of two edges). However, efficiently detecting larger cycles, namely k-cycle for arbitrary k, aiming to reduce communication costs and iteration rounds, continues to be a challenging area of research.

Cycle detection, often based on a BFS approach where two traversals run from a vertex until they meet, hinges on minimizing communication overhead. A notable strategy, as described in [33], involves sparsification—running BFS on a subgraph with edges randomly removed to reduce communication and computation costs. However, this can cause false negatives if edges in a cycle are removed. To counter this, the algorithm is executed multiple times on varied sparsified subgraphs, with the results aggregated to identify cycles. An enhanced method in [67] improves accuracy by selectively removing edges less likely to form cycles, balancing efficiency with the likelihood of detecting cycles.

Maximum Flow in graph theory focuses on computing the maximum flow achievable from a source vertex s to a sink vertex t in a graph. Each edge in this graph is assigned a certain capacity that limits the flow through it. The core methodology for determining maximum flow involves repeatedly identifying paths through the graph—known as augmenting paths—that can still carry more flow. This augmentation process is conducted iteratively: after each traversal, the flow is augmented along the identified path, and the residual capacities are updated. This process continues until no further augmenting paths can be identified, indicating that the maximum flow has been achieved.

In tackling parallelism challenges within maximum flow algorithms, optimizing the efficiency of graph traversals is crucial, particularly since each augmentation process involves a traversal that can be computationally intensive. The primary optimization strategy in this context aims to maximize the number of paths augmented in a single iteration, thereby enhancing overall efficiency. In line with this strategy, it has been proven more practical and efficient for identifying augmenting paths on a sparse subgraph. For example, the algorithm in [130] employs a method that extracts a spanning tree from the graph. It then concurrently finds multiple augmenting paths within this sub-structure. This approach not only boosts parallelism but also significantly reduces the number of iterative rounds required. Furthermore, as discussed in the context of SSSP tasks, flow problems can be reformulated as linear optimization problems and tackled using methods like gradient descent. This perspective allows for alternative solutions, as proposed in [71], which computes an approximate solution with a capacity penalty. Such an approach, focusing on approximation rather than exact solutions, enables the algorithm to achieve satisfactory results in fewer rounds.

4.6 Pattern Matching

In graph analysis, the task of pattern matching involves finding all subgraphs within a large data graph G that are isomorphic to a given small pattern graph p. Triangle counting helps understand the graph's fundamental structure, while k-Clique identifies larger, more complex specific patterns. Subgraph matching and mining offer broader applications by matching general patterns. These tasks are computationally intensive but only require local structural information, making them well-suited for distributed computing with large datasets. A survey presented in [24] offers an in-depth overview of distributed subgraph matching algorithms, focusing on programming models. Building on this, our survey highlights the primary challenges addressed by these studies and includes the latest research not covered in the previous survey.

Numerous studies have transformed essential operations in pattern matching into binary join operations [98], which are also inherently amenable to parallel processing. As illustrated in Figure 7, this has resulted in only a small portion of research (2.22%) focusing on the challenges of parallelizing pattern matching. In contrast, the issues of load balancing (35.56%) as well as communication overhead (48.89%) have garnered significantly more attention. Interestingly, fewer papers focus on the issue of bandwidth constraints (13.33%) in pattern matching. This oversight could be critical, considering the vast amounts of intermediate data that pattern matching processes can generate.

Triangle Counting, as implied by its name, is a specialized variant of pattern matching where the specific pattern p under consideration is a triangle. It is important to note that while cycle detection and k-clique could also be considered specific forms of pattern matching, they have been previously discussed in sections Section 4.5 and Section 4.4, respectively. Algorithms for computing triangles generally fall into two main categories: (1) List-based algorithms: These algorithms [38, 39, 73, 85, 138, 149, 150] primarily focus on traversing adjacency lists of two vertices to identify their common neighbors. (2) Map-based algorithms: In contrast, these algorithms [7, 72, 127, 173] employ an auxiliary data structure, such as a hash map, to maintain each vertex's adjacency list, which involves a vertex checking whether its neighbors appear in the auxiliary structures of other neighbors.

Load balancing poses a significant challenge in both list-based and map-based algorithms, especially when dealing with power-law graphs characterized by uneven vertex degree distribution. The key of achieving workload balance lies in effective graph partitioning strategies. In list-based algorithms, 1D partitioning [72, 73, 150] is commonly employed, focusing on partitioning either vertices or edges. Conversely, 2D partitioning [7, 127, 173] is more prevalent in map-based algorithms, wherein the data is divided across the graph's two-dimensional adjacency matrix. Optimization strategies vary based on the partitioning method chosen. The study in [7] begins with generating an adjacency matrix from sorted vertices to distribute tasks evenly across machines. It utilizes cyclic distribution to balance light and heavy tasks. Similarly, [173]adapts this approach in heterogeneous distributed environments. TRUST [127] adopts a distinct 2D distribution strategy, partitioning vertices first based on their one-hop and then using these to form two-hop neighbor partitions, to ensure balanced load distribution. In [150], a cyclic distribution method akin to [7] is implemented for distributing sorted vertices in a 1D partitioning scheme. Meanwhile, other 1D partitioning algorithms like [72, 73] strive for a more equitable distribution of edges to improve overall load balance.

Optimizing communication overhead is critical in triangle counting, given its communication-intensive nature [149]. The efficiency in distributed environments is significantly impacted by the frequency and volume of communications. One effective strategy is message aggregation, where small messages are consolidated into larger ones for collective transmission, reducing communication frequency and potentially optimizing bandwidth [72, 73, 138, 149]. Expander decomposition is another approach used in several studies [38, 39, 72]. It segments graphs into partitions of varying densities, assigning computational tasks based on these densities. High-density partitions are processed within single machines to minimize communication overhead. In [150],caching RMA accesses are studied to reduce remote data requests and communication demands. DistTC [85] introduces a fully asynchronous triangle counting algorithm, using boundary vertices and their connecting edges as proxies replicated across machines to enable independent triangle counting without inter-machine communication. Finally, Ancy Sarah Tom et al. [7] implement a pruning technique. Specifically, during the intersection of two adjacency lists of vertices u_k and u_j, only the triangle-closing vertices, u_k, satisfying $k > j$ are considered, reducing unnecessary computations and communications.

k-Clique is defined as a subgraph that contains k vertices, with each pair of vertices being interconnected by an edge. As a local task, the computation of k-clique is well-suited for parallel processing. However, the high computational complexity of k-clique computations, typically requiring $O(k^2n^k)$ time in single-machine environments as noted by Shahrivari and Jalili [143], poses a significant challenge. During the computation, each vertex is involved in sending/receiving substantial amounts of information to/from its neighbors. Consequently, key focuses in optimizing k-clique computations include achieving load balance, reducing communication, and managing bandwidth constraints.

For workload balancing, the study by Censor-Hillel et al. [35] introduces the generalized partition trees, which are constructed by a partial-pass streaming algorithm: it adds vertices one by one to the current partition, and either continues if the number of edges is within a threshold, or creates a new partition if otherwise. This method is designed to ensure equitable workload distribution, with each machine processing a similar number of edges.

Many research efforts in the field of k-clique computation are directed toward reducing communication overhead. A notable example presented in [35] introduces a two-tiered communication granularity, comprising auxiliary tokens and main tokens. The core strategy of this approach is to prioritize the transmission of main tokens between vertices, resorting to the transmission of auxiliary tokens only when absolutely necessary. By adopting this method, the algorithm effectively reduces the overall volume of communication. Additionally, the algorithm in [34] contributes to reducing communication overhead by eliminating vertices with lower degrees and their connected edges in each iteration. A similar pruning strategy is employed in KCminer [143]. In this approach, vertices with fewer than k neighbors are discarded before computation begins.

Bandwidth constraints pose significant challenges in computing k-clique. Matthias Bonne et al. [22] address this by sending only digests of neighbors to avoid network congestion. Another approach [34] defers processing edges likely to require substantial computations early on, reducing initial information transmission that can lead to network bottlenecks. Later in computation, removing vertices and edges gradually decreases communicated data, easing bandwidth limitations.

Subgraph Matching is another variant of pattern matching. In this variant, the pattern graph p can be any arbitrary graph, though it is usually small in size. This is the most general form of pattern matching, which is the core operation while querying modern graph databases [6].

The super-linear nature of subgraph matching often leads to significant load skew, making load balancing a crucial aspect of the task. Static heuristics are typically leveraged for this issue. In [163], task splitting is implemented for vertices whose degree exceeds a predefined threshold. This method divides large tasks into smaller subtasks, promoting more balanced workload distribution. Lizhi Xiang et al. [167] address load imbalance caused by processing vertices in the order of their degrees. They introduce a strategy that randomizes task execution paths, thereby preventing skew due to sequential processing. Besides static heuristics, dynamic scheduling presents an effective solution for balancing workloads during runtime. One such approach permits workers to "steal" unprocessed tasks from their peers after completing their assigned tasks. Implemented in [172], this method significantly enhances efficiency by dynamically redistributing workloads among machines.

The task of subgraph matching tends to become communication-intensive [98] in the distributed context, and thus reducing communication overhead is a critical challenge. One effective approach is to utilize an optimizer to devise an execution plan that minimizes this estimated communication cost [98]. Adopting this strategy, the study in [97] proposes estimating the cost on a random graph. Based on this estimation, they demonstrate that decomposing the pattern into wedges (intersections of two edges) and subsequently joining the matches of these wedges is an instance-optimal method in terms of communication cost. The approach in [5] employs a subgraph matching approach based on the worst-case optimal join algorithm [125]. The GLogS system [99] refines cost estimation by computing smaller pattern sizes on a sparsified graph, improving the accuracy of communication cost estimates. Caching accessed vertex data locally is another effective solution, as it reduces redundant communication that occurs when the same vertex data is fetched repeatedly [163, 172]. This method ensures that frequently accessed data is readily available, thereby minimizing unnecessary data transfers. Yang et al. [172] dynamically select between *push* and *pull* communication strategies to further reduce communication costs based on the context. Furthermore, the study in [167] introduces a new data structure that facilitates the reuse of prefix paths. This structure significantly reduces data redundancy and lessens the volume of data transmitted between machines, contributing to more efficient communication in distributed subgraph matching.

Subgraph Mining goes a step further than merely identifying a single pattern; it involves the computation of a sequence of interconnected patterns. Essentially, the task of subgraph mining can be viewed as a series of subgraph matching processes. Consequently, the challenges encountered in subgraph mining closely mirror those in subgraph matching.

Achieving workload balance is critical in subgraph mining, prompting the development of various static and dynamic strategies. Statically, the study in [44] introduces "BDG Partitioning" as a method for maintaining graph locality within subgraphs. This technique use multi-source distributed BFS with limited depth to color vertices and form subgraphs with balanced computational loads. Dynamically, Arabesque, as presented in [154], adopts a round-robin strategy for workload distribution. This approach sequentially and cyclically allocates tasks to machines, inherently balancing the load due to the randomness in task assignment. Another dynamic approach is detailed in [47], where workloads are dynamically dispatched. This method involves maintaining a workload queue, into which intermediate results from completed tasks are subdivided into smaller tasks. These smaller tasks are then dynamically distributed to available working threads as needed. G-Miner, also proposed in [44], supports dynamic load balancing as well. It utilizes a strategy that allows workers to "steal" unprocessed tasks from others. This method ensures that all workers are actively engaged and effectively redistributes tasks to balance the overall workload across the system.

Subgraph mining necessitates strategies to minimize communication overhead. A widely adopted approach, similar to that of subgraph matching, involves caching frequently accessed vertex data [44, 47, 49]. This caching mechanism ensures that critical data is readily available, thereby reducing repetitive data requests across the network. In addition to caching, the research in [44] introduces a task priority queue to organize tasks with common remote candidates. This grouping allows related data to be transferred in a single operation, reducing the frequency and volume of remote data fetches. Furthermore, the study in [153] explores the use of a personalized broadcast operation. This method involves selectively assigning data to relevant machines based on their specific needs or the tasks they are performing, aiming at minimizing unnecessary communication.

4.7 Covering

Covering tasks comprise a category of optimization problems centered around identifying sets of vertices or edges that adhere to specific constraints. Minimum vertex covering seeks the smallest set of vertices covering all edges, maximum matching aims to find the largest set of non-overlapping edges, and graph coloring assigns colors to vertices such that no adjacent vertices share the same color, with each problem focusing on different aspects of graph structure and optimization. These problems are NP-hard, making the algorithms computationally intensive and difficult to parallelize.

Consequently, over half of the reviewed papers focus on enhancing parallelism (64.29%). Additionally, the complexities of computation pose issues with load balancing, and 7.14% of the surveyed papers target it. Moreover, these algorithms often necessitate decisions based on neighboring vertices' data, making the reduction of communication overhead (28.57%) another critical concern.

Minimum Vertex Covering is challenging due to its NP-hard nature, which is further compounded in distributed environments. Recent works [45] try to find a strict Nash Equilibrium state as an approximate, yet near-optimal, solution. In this context, vertices act as strategic agents in a snowdrift game with adjacent vertices, representing a local optimization process. The game's strict Nash Equilibrium corresponds to a vertex cover, aligning the strategy with the overall objective. This game-based algorithm is well-suited to the message-passing model, allowing each vertex to select a beneficial neighbor based on both past and present decisions through rational adjustments.

The level of parallelism achieved can significantly influence the rate of convergence. With limited local storage, where a vertex can only retain information on one prior selection, the process may fail to reach a strict Nash

Equilibrium and get stuck in a deadlock. To overcome these limitations and expedite convergence, some algorithms employ randomization strategies—permitting vertices to select neighbors randomly instead of rationally, thereby introducing a chance to *jump out* from the dead loop [45, 46]. Further enhancements in parallelism can be realized by expanding the local memory capacity, allowing for the storage of additional previous choices [151].

Maximum Matching seeks the largest collection of edges in a graph where no two edges share a common vertex. A traditional algorithm to find a maximum matching is the Hungary algorithm [96], which iteratively searches for an augmenting path from each unmatched vertex. An augmenting path, consisting of alternating matched and unmatched edges, is leveraged to increment the size of the matching. The Hopcroft-Karp algorithm [68], optimized for parallelism, can generate multiple augmenting paths within a single iteration. While these algorithms are fundamentally designed for bipartite graphs, they can be adapted for use with general graphs.

The augmenting process is the bottleneck of the parallelism. An efficient augmenting path algorithm can reduce iteration rounds [2]. Another algorithm is to locally simulate multiple augmenting paths, since these paths are not too long [61]. Recently, Sparse Matrix-Vector Multiplication (SpMV) has been well studied and the augmenting process can be rewritten by matrix operations [12]. The fast matrix calculation can significantly increase parallelism with humongous computing resources.

Another challenge is the high communication cost, stemming from the need for each augmenting path search to traverse the entire graph. However, since augmenting paths are typically short, operating on a strategically chosen subgraph with fewer vertices can be more efficient without compromising the quality of the result [55].

Graph Coloring algorithms iteratively select an independent set (IS) for coloring. An IS is a set of vertices in which no two are adjacent, allowing for simultaneous conflict-free coloring. Each vertex in the IS greedily selects the lowest available color not used by its neighbors. The IS can be obtained by some sampling methods (e.g., Maximal Independent Set) or priority methods.

To enhance parallelism, one strategy is to identify a larger IS, coloring more vertices concurrently. A vertex-cut-based algorithm partitions the graph into components by removing vertex-cuts, allowing IS algorithms to be efficiently applied to these smaller, more manageable components [129]. Another approach optimizes the coloring priority scheme so that lower-priority vertices get colored earlier, reducing wait times for higher-priority decisions [3].

Graph coloring algorithms often incur excessive communication overhead towards the end of the process due to increased dependency on a dwindling number of uncolored vertices. A straightforward solution is to precompute frequently occurring structures, thereby coloring them in a single instance [129], or to defer their coloring to a later stage [42]. Building on this concept, a subgraph-based algorithm decomposes the main graph into several subgraphs, prioritizes the coloring of internal vertices, and subsequently resolves color conflicts that arise among the boundary vertices [21].

5 Application Scenarios of Various Graph Tasks

Distributed graph tasks have numerous real-world applications, thanks to the expressive power of graphs and their ability to handle large-scale data efficiently in distributed environments. This section provides examples (Figure 8), highlighting how distributed graph tasks are applied across various scenarios, as derived from applications mentioned in academic papers. Additionally, we provide descriptions of the performance differences between various graph tasks on certain topics and explain how these differences result in different application scenarios.

Centrality. In social network analysis, PageRank and personalized PageRank identify influential individuals [82, 112]. Closeness and betweenness centrality, calculated based on the shortest path, have lower efficiency but are more suitable for identifying key vertices in information transmission. For example, in biological network analysis, betweenness centrality identifies crucial genes or proteins that act as key intermediaries [59]. In supply

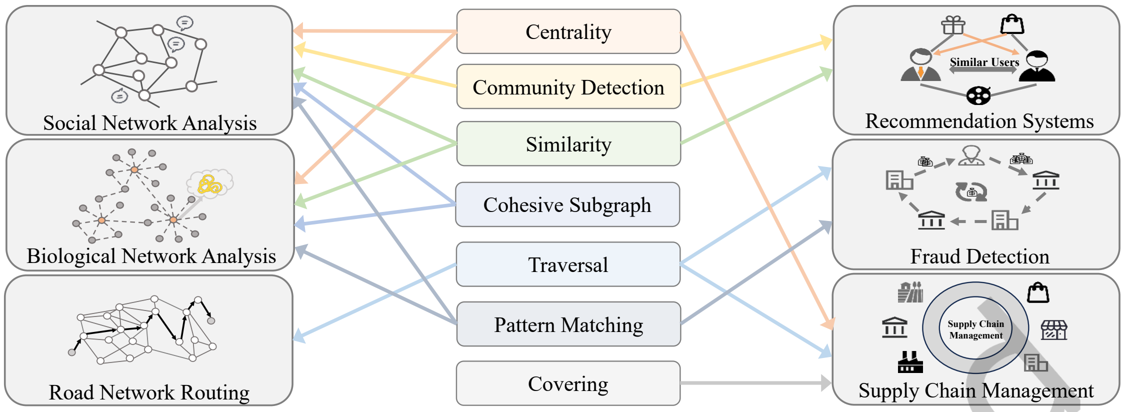


Fig. 8. Application scenarios of graph tasks. Each arrow indicates that academic papers mention the application of graph tasks within the corresponding topic to the target scenarios.

chain management, betweenness centrality finds important intermediaries in logistics pathways, improving efficiency and reducing costs [160]. Closeness centrality helps optimize logistics routes by identifying key vertices.

Community Detection. In social network analysis, these algorithms are used to identify densely connected communities [13, 116, 178]. Users within the same community often share common interests, making community detection algorithms highly useful for recommendation systems. Louvain is an iterative method that optimizes modularity by progressively merging vertices and communities, ultimately forming a stable community structure. Due to its efficiency and scalability, Louvain is well-suited for handling large networks [142, 178]. LPA is fast but may produce unstable results in large networks, ideal for real-time data stream analysis [10, 108, 116]. The connected components algorithm is low-cost and suitable for basic network analysis [89, 110].

Similarity. In social network analysis, similarity algorithms identify users with similar interests, helpful for making friends. In biological networks, it compares genes or proteins to find those with similar functions, aiding in research and drug development. In recommendation systems, it can be used to recommend other products similar to those a user has browsed or purchased, enhancing the user experience and increasing sales [114, 115].

Cohesive Subgraph. Cohesive subgraphs have significant applications in various networks. In social networks, they help identify tightly connected user groups, like interest-based communities [37, 105, 120, 144]. In biological networks, they find gene groups with similar expressions, indicating potential functions [123, 152]. In e-commerce networks, they identify user groups with similar behavior, improving recommendation accuracy. Besides, different algorithms are also suited to some specific scenarios based on their trade-off between efficiency and subgraph density. For example, the maximum clique algorithm suits scenarios needing very tight connections but less time sensitivity, while k-Core works well for large networks due to its efficiency and scalability [105, 168].

Traversal. Traversal algorithms have diverse applications due to their different definitions. In road networks, BFS is used for simple tasks like finding the shortest path in an unweighted map [57], while SSSP handles complex networks, considering direction and traffic [84]. In supply chain management, the minimum spanning tree reduces total transportation costs, and maximum flow optimizes goods flow, increasing logistics throughput [93]. Cycle detection identifies money laundering in financial networks and is used in operating systems to detect deadlocks [67].

Pattern Matching. In social network analysis, triangle counting evaluates community cohesion and aids clustering [38, 85, 173], while k-Clique detection identifies tightly connected user groups [143]. In biological networks, *k*-Clique identifies protein interaction clusters, and subgraph matching locates specific specific gene clusters [44, 153, 154]. In financial networks, subgraph matching and subgraph mining detect abnormal transaction patterns and potential fraudulent activities [44].

Covering. The algorithms are commonly used for optimization of resource allocation and supply chain management [2]. Minimum vertex cover identifies key vertices to monitor, optimizing resource allocation [46]. For example, in sensor networks, minimum vertex cover helps in placing the least number of sensors to monitor all critical points. Maximum matching pairs vertices to maximize the number of edges, useful in optimizing resource distribution [12]. Graph coloring assigns colors to vertices to minimize conflicts, essential in scheduling and resource allocation tasks [21].

6 Discussion And Opportunities

This section aims to discuss prevalent research trends addressing various challenges in this domain, and offer perspectives on potential future research opportunities.

6.1 Computation Resource Efficiency

In the realm of distributed graph processing, the complete utilization of computation resources is identified as a primary objective. Numerous studies spanning a variety of topics have been conducted with the aim of optimizing resource utilization. Nevertheless, the effects of parallelism and load balancing have been found to vary significantly across these diverse topics.

- *Parallelism*: Paralleling graph tasks is the first step in distributed computing. It has been observed that not all topics focus on the challenges posed by parallelism. Tasks that possess sequential dependencies, such as centrality, cohesive subgraph (like *k*-core and *k*-truss), and traversal, as well as those requiring global information like graph coloring, tend to allocate special attention to these challenges. In such tasks, the state of each vertex in each step depends on the results of previous steps, or there is a necessity to consider the structure of the entire graph. This makes it particularly challenging to circumvent conflicts and ensure the accuracy of algorithms under parallel computation. Conversely, for tasks that are local in nature, such as similarity and pattern matching, parallelism is not the primary concern.

  Opportunity. Current solutions aim to enhance the parallelism of algorithms but also introduce new challenges. In tasks such as *k*-core/*k*-truss, parallel computation of coreness/trussness might increase the number of iterations required for convergence. In traversal and graph coloring tasks, additional computational costs are incurred to ensure algorithmic correctness when applied in distributed environments. Furthermore, tasks adapting random walk-based algorithms can readily lead to local network congestion. It continues to be a significant challenge to parallelize algorithms while minimizing the negative impacts caused by such parallelization.
- *Load Balance*: Load balancing is a prevalent challenge, with substantial research across all topics focusing on strategies to effectively balance workloads. Computation-intensive tasks, such as pattern matching and cohesive subgraph (like maximal clique), especially emphasize this challenge. Real-world graphs often exhibit a power-law degree distribution where the workload is typically proportional to the vertex degree. This results in a skewed workload distribution that is particularly problematic in computation-intensive tasks.

  Opportunity. The majority of existing work employs static partitioning strategies, which involve initially allocating a similar number of edges or vertices to each machine. However, a single static partitioning strategy may not be suitable for every type of graph data and algorithm, and it is challenging to ensure balanced workloads throughout the computation process. Developing dynamic load balance strategies is a valuable future direction.

6.2 Network Resource Efficiency

Our survey indicates that network resource efficiency, which encompasses communication overhead and bandwidth, follows a comparable pattern across different topics. There is a significant emphasis on minimizing communication overhead, yet bandwidth concerns receive comparatively less focus.

- *Communication Overhead*: Communication overhead is a challenge that nearly all topics address, with more than half of the studies primarily dedicated to reducing it. Graph algorithms, particularly iterative ones, tend to produce substantial intermediate data throughout their execution. When vertices are distributed across separate machines, they require network communication to exchange data, making communication overhead a widespread issue in distributed computing.
  Opportunity. Minimizing communication overhead remains a critical challenge for many tasks within distributed settings. Existing work typically employs one or two strategies to optimize communication overhead, such as the Subgraph-Centric model or pruning strategies. These methods are broadly applicable and generally suited for almost all tasks. Using multiple optimization techniques could further optimize communication overhead. Moreover, the integration of deep learning techniques to predict and dynamically adjust communication patterns based on the current network state and workload distribution is also gaining increased attention.
- *Bandwidth*: Relatively few studies concentrate on bandwidth issues; however, some computation-intensive tasks such as pattern matching, as well as algorithms utilizing random walks, have proposed methods to mitigate this challenge. Bandwidth commonly is not a limiting factor for the performance of tasks that are not communication-intensive or when processing sparse graph data. Furthermore, strategies aimed at minimizing communication overhead and achieving load balance often help to alleviate bandwidth constraints.
  Opportunity. Nevertheless, as graph data scales continue to grow, the importance of bandwidth management is expected to escalate. Addressing this issue is inherently more complex than other challenges because it involves preventing network congestion while also maximizing bandwidth efficiency to avoid resource wastage. Future research will likely place increased emphasis on effectively tackling bandwidth challenges, potentially through the development of advanced network management techniques, like adaptive bandwidth allocation based on live traffic analysis.

7 Conclusions

Graphs can well represent relationships among entities. Analyzing and processing large-scale graph data has been applied in many applications, such as social network analysis, recommendation systems, and road network routing. Distributed graph processing provides a solution for efficiently handling large-scale graph data in the real world. To understand the state-of-the-art research of graph tasks in distributed environments and facilitate its development, in this paper, we conduct an extensive survey on distributed graph tasks.

We first overview the existing distributed graph processing infrastructure. These tools facilitate the design of distributed algorithms, but it is still difficult to overcome the challenges arising from the inherent characteristics of distributed systems and graphs. We then analyze and summarize the main challenges and solutions for graph tasks in distributed environments. Next, we provide a taxonomy of primary graph tasks and conduct a detailed analysis of their existing efforts on distributed environments, including challenges they focus on and unique insights for solving those challenges. Finally, we discuss the research focus and the existing research gaps in the field of distributed graph processing and identify potential future research opportunities.

Acknowledgments

Long Yuan is supported by NSFC 62472225. Xuemin Lin is supported by NSFC U2241211 and U20B2046.

References

[1] James Abello, Frank van Ham, and et al. 2006. ASK-GraphView: A Large Scale Graph Visualization System. *TVCG* 12, 5 (2006), 669–676.

[2] Mohamad Ahmadi, Fabian Kuhn, and et al. 2018. Distributed Approximate Maximum Matching in the CONGEST Model. In *DISC 2018 (LIPIcs, Vol. 121)*. 6:1–6:17.

[3] Ghadeer Alabandi, Evan Powers, and et al. 2020. Increasing the parallelism of graph coloring via shortcutting. In *PPoPP 2020*. 262–275.

[4] Mehdi Alemi and Hassan Haghighi. 2019. KTMiner: Distributed k-truss detection in big graphs. *Inf. Syst.* 83 (2019), 195–216.

[5] Khaled Ammar, Frank McSherry, and et al. 2018. Distributed Evaluation of Subgraph Queries Using Worst-case Optimal and Low-Memory Dataflows. *Proc. VLDB Endow.* 11, 6 (2018), 691–704.

[6] Renzo Angles, Marcelo Arenas, and et al. 2017. Foundations of Modern Query Languages for Graph Databases. *ACM Comput. Surv.* 50, 5 (2017), 68:1–68:40.

[7] Shaikh Arifuzzaman, Maleq Khan, and et al. 2017. Distributed-Memory Parallel Algorithms for Counting and Listing Triangles in Big Graphs. *CoRR* abs/1706.05151 (2017). arXiv:1706.05151

[8] Joe Armstrong. 2010. erlang. *Commun. ACM* 53, 9 (2010), 68–75.

[9] Arthur U. Asuncion, Padhraic Smyth, and et al. 2008. Asynchronous Distributed Learning of Topic Models. In *NIPS 2008*. Curran Associates, Inc., 81–88.

[10] Jean-Philippe Attal, Maria Malek, and et al. 2019. Parallel and distributed core label propagation with graph coloring. *Concurr. Comput. Pract. Exp.* 31, 2 (2019).

[11] Ching Avery. 2011. Giraph: Large-scale graph processing infrastructure on hadoop. *Proceedings of the Hadoop Summit. Santa Clara* 11, 3 (2011), 5–9.

[12] Ariful Azad and Aydin Buluç. 2015. Distributed-Memory Algorithms for Maximal Cardinality Matching Using Matrix Algebra. In *CLUSTER 2015*. 398–407.

[13] Mehdi Azaouzi and Lotfi Ben Romdhane. 2017. An evidential influence-based label propagation algorithm for distributed community detection in social networks. *Procedia computer science* 112 (2017), 407–416.

[14] Nitin Chandra Badam and Yogesh Simmhan. 2014. Subgraph Rank: PageRank for Subgraph-Centric Distributed Graph Processing. In *COMAD 2014*. 38–49.

[15] Rosa M. Badia, Javier Conejero, and et al. 2022. PyCOMPSs as an Instrument for Translational Computer Science. *Comput. Sci. Eng.* 24, 2 (2022), 79–84.

[16] Henri E. Bal, Jennifer G. Steiner, and et al. 1989. Programming Languages for Distributed Computing Systems. *ACM Comput. Surv.* 21, 3 (1989), 261–322.

[17] Ruben Becker, Sebastian Forster, and et al. 2021. Near-Optimal Approximate Shortest Paths and Transshipment in Distributed and Streaming Models. *SIAM J. Comput.* 50, 3 (2021), 815–856.

[18] Massimo Bernaschi, Giancarlo Carbone, and et al. 2015. Solutions to the st-connectivity problem using a GPU-based distributed BFS. *JPDC* 76 (2015), 145–153.

[19] Maciej Besta, Raghavendra Kanakagiri, and et al. 2020. Communication-Efficient Jaccard similarity for High-Performance Distributed Genome Comparisons. In *IPDPS 2020*. 1122–1132.

[20] Vincent D Blondel, Jean-Loup Guillaume, and et al. 2008. Fast unfolding of communities in large networks. *Journal of statistical mechanics: theory and experiment* 2008, 10 (2008), P10008.

[21] Ian Bogle, Erik G. Boman, and et al. 2020. Distributed Memory Graph Coloring Algorithms for Multiple GPUs. In *IA3 2020*. 54–62.

[22] Matthias Bonne and Keren Censor-Hillel. 2019. Distributed Detection of Cliques in Dynamic Networks. In *ICALP 2019 (LIPIcs, Vol. 132)*. 132:1–132:15.

[23] Karsten M. Borgwardt, Cheng Soon Ong, and et al. 2005. Protein function prediction via graph kernels. In *ISMB 2005*. 47–56.

[24] Sarra Bouhenni, Saïd Yahiaoui, and et al. 2022. A Survey on Distributed Graph Pattern Matching in Massive Graphs. *ACM Comput. Surv.* 54, 2 (2022), 36:1–36:35.

[25] Ulrik Brandes. 2001. A Faster Algorithm for Betweenness Centrality, In Journal of Mathematical Sociology. *Journal of Mathematical Sociology* 25, 163–177.

[26] Sergey Brin and Lawrence Page. 1998. The Anatomy of a Large-Scale Hypertextual Web Search Engine. *Comput. Networks* 30 (1998), 107–117.

[27] Andrei Z. Broder, Ravi Kumar, and et al. 2000. Graph structure in the Web. *Comput. Networks* 33, 1-6 (2000), 309–320.

[28] Federico Busato and Nicola Bombieri. 2015. BFS-4K: An Efficient Implementation of BFS for Kepler GPU Architectures. *IEEE Trans. Parallel Distributed Syst.* 26, 7 (2015), 1826–1838.

[29] Federico Busato and Nicola Bombieri. 2016. An Efficient Implementation of the Bellman-Ford Algorithm for Kepler GPU Architectures. *IEEE Trans. Parallel Distributed Syst.* 27, 8 (2016), 2222–2233.

[30] Huanqi Cao, Yuanwei Wang, and et al. 2022. Scaling graph traversal to 281 trillion edges with 40 million cores. In *PPoPP 2022*. 234–245.

[31] Nairen Cao, Jeremy T. Fineman, and et al. 2021. Brief Announcement: An Improved Distributed Approximate Single Source Shortest Paths Algorithm. In *PODC 2021*. 493–496.

[32] Paris Carbone, Asterios Katsifodimos, and et al. 2015. Apache Flink™: Stream and Batch Processing in a Single Engine. *IEEE Data Eng. Bull.* 38, 4 (2015), 28–38.

[33] Keren Censor-Hillel, Eldar Fischer, and et al. 2019. Fast distributed algorithms for testing graph properties. *Distributed Comput.* 32, 1 (2019), 41–57.

[34] Keren Censor-Hillel, François Le Gall, and et al. 2020. On Distributed Listing of Cliques. In *PODC 2020*. 474–482.

[35] Keren Censor-Hillel, Dean Leitersdorf, and et al. 2022. Deterministic Near-Optimal Distributed Listing of Cliques. In *PODC 2022*. 271–280.

[36] Venkatesan T. Chakaravarthy, Fabio Checconi, and et al. 2017. Scalable Single Source Shortest Path Algorithms for Massively Parallel Systems. *IEEE Trans. Parallel Distributed Syst.* (2017), 2031–2045.

[37] T.-H. Hubert Chan, Mauro Sozio, and et al. 2021. Distributed approximate k-core decomposition and min-max edge orientation: Breaking the diameter barrier. *J. Parallel Distributed Comput.* 147 (2021), 87–99.

[38] Yi-Jun Chang, Seth Pettie, and et al. 2021. Near-optimal Distributed Triangle Enumeration via Expander Decompositions. *J. ACM* 68, 3 (2021), 21:1–21:36.

[39] Yi-Jun Chang and Thatchaphol Saranurak. 2019. Improved Distributed Expander Decomposition and Nearly Optimal Triangle Enumeration. In *PODC 2019*. 66–73.

[40] Barbara M. Chapman, Tony Curtis, and et al. 2010. Introducing OpenSHMEM: SHMEM for the PGAS community. In *PGAS 2010*. ACM, 2.

[41] Philippe Charles, Christian Grothoff, and et al. 2005. X10: an object-oriented approach to non-uniform cluster computing. In *OOPSLA 2005*. ACM, 519–538.

[42] Shiri Chechik and Doron Mukhtar. 2019. Optimal Distributed Coloring Algorithms for Planar Graphs in the LOCAL model. In *SODA 2019*. 787–804.

[43] Shiri Chechik and Doron Mukhtar. 2022. Single-source shortest paths in the CONGEST model with improved bounds. *Distributed Comput.* 35, 4 (2022), 357–374.

[44] Hongzhi Chen, Miao Liu, and et al. 2018. G-Miner: an efficient task-oriented graph mining system. In *EuroSys 2018*. 32:1–32:12.

[45] Jie Chen and Xiang Li. 2021. A Minimal Memory Game-Based Distributed Algorithm to Vertex Cover of Networks. In *ISCAS 2021*. 1–5.

[46] Jie Chen and Xiang Li. 2023. Toward the minimum vertex cover of complex networks using distributed potential games. *Sci. China Inf. Sci.* 66, 1 (2023).

[47] Jingji Chen and Xuehai Qian. 2023. Khuzdul: Efficient and Scalable Distributed Graph Pattern Mining Engine. In *ASPLOS 2023*. 413–426.

[48] Pei-Ling Chen, Chung-Kuang Chou, and et al. 2014. Distributed algorithms for k-truss decomposition. In *IEEE BigData 2014*. 471–480.

[49] Xuhao Chen, Tianhao Huang, and et al. 2021. FlexMiner: A Pattern-Aware Accelerator for Graph Pattern Mining. In *ISCA 2021*. 581–594.

[50] Aaron Clauset, Cosma Rohilla Shalizi, and et al. 2009. Power-Law Distributions in Empirical Data. *SIAM Rev.* 51, 4 (2009), 661–703.

[51] Mirel Cosulschi, Mihai Gabroveanu, and et al. 2015. Scaling Up a Distributed Computing Of Similarity Coefficient with Mapreduce. *Int. J. Comput. Sci. Appl.* 12, 2 (2015), 81–98.

[52] Pierluigi Crescenzi, Pierre Fraigniaud, and et al. 2020. Simple and Fast Distributed Computation of Betweenness Centrality. In *INFOCOM 2020*. 337–346.

[53] Pawel Czarnul, Jerzy Proficz, and et al. 2020. Survey of Methodologies, Approaches, and Challenges in Parallel Programming Using High-Performance Computing Systems. *Sci. Program.* 2020 (2020), 4176794:1–4176794:19.

[54] Guohao Dai, Tianhao Huang, and et al. 2019. GraphH: A Processing-in-Memory Architecture for Large-Scale Graph Processing. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* 38, 4 (2019), 640–653.

[55] Varsha Dani, Aayush Gupta, and et al. 2023. Wake up and join me! An energy-efficient algorithm for maximal matching in radio networks. *Distributed Comput.* 36, 3 (2023), 373–384.

[56] Jeffrey Dean and Sanjay Ghemawat. 2004. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI 2004*. 137–150.

[57] Stéphane Devismes and Colette Johnen. 2016. Silent self-stabilizing BFS tree algorithms revisited. *J. Parallel Distributed Comput.* 97 (2016), 11–23.

[58] David Dominguez-Sal, Norbert Martínez-Bazan, and et al. 2010. A Discussion on the Design of Graph Database Benchmarks. In *TPCTC 2010 (Lecture Notes in Computer Science, Vol. 6417)*. Springer, 25–40.

[59] Kevin Durant and Stephan Wagner. 2017. On the distribution of betweenness centrality in random trees. *Theor. Comput. Sci.* 699 (2017), 33–52.

[60] Michael Elkin. 2020. Distributed Exact Shortest Paths in Sublinear Time. *J. ACM* 67, 3 (2020), 15:1–15:36.

[61] Guy Even, Moti Medina, and et al. 2015. Distributed Maximum Matching in Bounded Degree Graphs. In *Proceedings of the 2015 International Conference on Distributed Computing and Networking, ICDCN 2015*. 18:1–18:10.

[62] Wenfei Fan, Tao He, and et al. 2021. GraphScope: A Unified Engine For Big Graph Processing. *Proc. VLDB Endow.* 14, 12 (2021), 2879–2892.

[63] Xing Feng, Lijun Chang, and et al. 2018. Distributed computing connected components with linear communication cost. *Distributed Parallel Databases* 36, 3 (2018), 555–592.

[64] Sebastian Forster and Danupon Nanongkai. 2018. A Faster Distributed Single-Source Shortest Paths Algorithm. In *FOCS 2018*. 686–697.

[65] Message P Forum. 1994. MPI: A message-passing interface standard.

[66] Cédric Fournet, Fabrice Le Fessant, and et al. 2002. JoCaml: A language for concurrent distributed and mobile programming. In *International School on Advanced Functional Programming*. 129–158.

[67] Pierre Fraigniaud and Dennis Olivetti. 2017. Distributed Detection of Cycles. In *Proceedings of the 29th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA 2017*. 153–162.

[68] Harold N. Gabow. 2017. The Weighted Matching Approach to Maximum Cardinality Matching. *Fundam. Informaticae* 154, 1-4 (2017), 109–130.

[69] Robert G. Gallager, Pierre A. Humblet, and et al. 1983. A Distributed Algorithm for Minimum-Weight Spanning Trees. *ACM Trans. Program. Lang. Syst.* 5, 1 (1983), 66–77.

[70] Juan A. Garay, Shay Kutten, and et al. 1998. A Sublinear Time Distributed Algorithm for Minimum-Weight Spanning Trees. *SIAM J. Comput.* 27, 1 (1998), 302–316.

[71] Mohsen Ghaffari, Andreas Karrenbauer, and et al. 2018. Near-Optimal Distributed Maximum Flow. *SIAM J. Comput.* 47, 6 (2018), 2078–2117.

[72] Sayan Ghosh. 2022. Improved Distributed-memory Triangle Counting by Exploiting the Graph Structure. In *HPEC 2022*. 1–6.

[73] Sayan Ghosh and Mahantesh Halappanavar. 2020. TriC: Distributed-memory Triangle Counting by Exploiting the Graph Structure. In *HPEC 2020*. 1–6.

[74] Sayan Ghosh, Mahantesh Halappanavar, and et al. 2018. Distributed Louvain Algorithm for Graph Community Detection. In *2018 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2018*. 885–895.

[75] Andrew V. Goldberg and Chris Harrelson. 2005. Computing the shortest path: *A* search meets graph theory. In *SODA 2005*. 156–165.

[76] Joseph E. Gonzalez, Yucheng Low, and et al. 2012. PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs. In *OSDI 2012*. 17–30.

[77] Joseph E. Gonzalez, Reynold S. Xin, and et al. 2014. GraphX: Graph Processing in a Distributed Dataflow Framework. In *OSDI 2014*. 599–613.

[78] William Gropp. 2001. Learning from the Success of MPI. In *HiPC 2001 (Lecture Notes in Computer Science, Vol. 2228)*. Springer, 81–94.

[79] Tao Guo, Xin Cao, and et al. 2017. Distributed Algorithms on Exact Personalized PageRank. In *SIGMOD 2017*. 479–494.

[80] Shubhankar Gupta and Suresh Sundaram. 2023. Moving-Landmark Assisted Distributed Learning Based Decentralized Cooperative Localization (DL-DCL) with Fault Tolerance. In *AAAI 2023*. 6175–6182.

[81] Minyang Han and Khuzaima Daudjee. 2015. Giraph Unchained: Barrierless Asynchronous Parallel Execution in Pregel-like Graph Processing Systems. *Proc. VLDB Endow.* 8, 9 (2015), 950–961.

[82] Yiran He and Hoi-To Wai. 2021. Provably Fast Asynchronous And Distributed Algorithms For Pagerank Centrality Computation. In *ICASSP 2021*. 5050–5054.

[83] Safiollah Heidari, Yogesh Simmhan, and et al. 2018. Scalable Graph Processing Frameworks: A Taxonomy and Open Challenges. *ACM Comput. Surv.* 51, 3 (2018), 60:1–60:53.

[84] Monika Henzinger, Sebastian Krinninger, and et al. 2021. A Deterministic Almost-Tight Distributed Algorithm for Approximating Single-Source Shortest Paths. *SIAM J. Comput.* 50, 3 (2021).

[85] Loc Hoang, Vishwesh Jatala, and et al. 2019. DistTC: High Performance Distributed Triangle Counting. In *HPEC 2019*. 1–7.

[86] Loc Hoang, Matteo Pontecorvi, and et al. 2019. A round-efficient distributed betweenness centrality algorithm. In *PPoPP 2019*. 272–286.

[87] Stephan Holzer and Roger Wattenhofer. 2012. Optimal distributed all pairs shortest paths and applications. In *PODC 2012*. 355–364.

[88] Qiang-Sheng Hua, Haoqiang Fan, and et al. 2016. Nearly Optimal Distributed Algorithm for Computing Betweenness Centrality. In *36th IEEE International Conference on Distributed Computing Systems, ICDCS 2016*. 271–280.

[89] Chirag Jain, Patrick Flick, and et al. 2017. An Adaptive Parallel Algorithm for Computing Connected Components. *IEEE Trans. Parallel Distributed Syst.* 28, 9 (2017), 2428–2439.

[90] Fuad T. Jamour, Spiros Skiadopoulos, and et al. 2018. Parallel Algorithm for Incremental Betweenness Centrality on Large Graphs. *IEEE Trans. Parallel Distributed Syst.* 29, 3 (2018), 659–672.

[91] Glen Jeh and Jennifer Widom. 2002. SimRank: a measure of structural-context similarity. In *KDD 2002*. 538–543.

[92] Yichuan Jiang. 2016. A Survey of Task Allocation and Load Balancing in Distributed Systems. *IEEE Trans. Parallel Distributed Syst.* 27, 2 (2016), 585–599.

[93] Changhee Joo, Xiaojun Lin, and et al. 2016. Distributed Greedy Approximation to Maximum Weighted Independent Set for Scheduling With Fading Channels. *IEEE/ACM Trans. Netw.* 24, 3 (2016), 1476–1488.

[94] Vasiliki Kalavri, Vladimir Vlassov, and et al. 2018. High-Level Programming Abstractions for Distributed Graph Processing. *IEEE Trans. Knowl. Data Eng.* 30, 2 (2018), 305–324.

[95] George Karypis and Vipin Kumar. 1997. METIS: A software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices. (1997).

[96] Harold W Kuhn. 1955. The Hungarian method for the assignment problem. *Nav. Res. Logist. Q.* 2, 1-2 (1955), 83–97.

[97] Longbin Lai, Lu Qin, and et al. 2015. Scalable Subgraph Enumeration in MapReduce. *Proc. VLDB Endow.* 8, 10 (2015), 974–985.

[98] Longbin Lai, Zhu Qing, and et al. 2019. Distributed Subgraph Matching on Timely Dataflow. *Proc. VLDB Endow.* 12, 10 (2019), 1099–1112.

[99] Longbin Lai, Yufan Yang, and et al. 2023. GLogS: Interactive Graph Pattern Matching Query At Large Scale. In *USENIX ATC 2023*. USENIX Association, 53–69.

[100] Kartik Lakhotia, Rajgopal Kannan, and et al. 2018. Accelerating PageRank using Partition-Centric Processing. In *USENIX ATC 2018*. 427–440.

[101] Sebastian Lamm and Peter Sanders. 2022. Communication-efficient Massively Distributed Connected Components. In *IPDPS 2022*. 302–312.

[102] Christoph Lenzen, Boaz Patt-Shamir, and et al. 2019. Distributed distance computation and routing with small messages. *Distributed Comput.* 32, 2 (2019), 133–157.

[103] Xue Li, Ke Meng, and et al. 2023. Flash: A Framework for Programming Distributed Graph Processing Algorithms. In *ICDE 2023*. IEEE, 232–244.

[104] Zhenguo Li, Yixiang Fang, and et al. 2015. Walking in the Cloud: Parallel SimRank at Scale. *Proc. VLDB Endow.* 9, 1 (2015), 24–35.

[105] Xuankun Liao, Qing Liu, and et al. 2022. Distributed D-core Decomposition over Large Directed Graphs. *Proc. VLDB Endow.* 15, 8 (2022), 1546–1558.

[106] Wenqing Lin. 2019. Distributed Algorithms for Fully Personalized PageRank on Large Graphs. In *WWW 2019*. 1084–1094.

[107] Qing Liu, Xuankun Liao, and et al. 2023. Distributed (α, β)-Core Decomposition over Bipartite Graphs. In *ICDE 2023*. 909–921.

[108] Xu Liu, Jesun Sahariar Firoz, and et al. 2019. Distributed Direction-Optimizing Label Propagation for Community Detection. In *HPEC 2019*. 1–6.

[109] Yucheng Low, Joseph Gonzalez, and et al. 2012. Distributed GraphLab: A Framework for Machine Learning in the Cloud. *Proc. VLDB Endow.* 5, 8 (2012), 716–727.

[110] Alessandro Lulli, Emanuele Carlini, and et al. 2017. Fast Connected Components Computation in Large Graphs by Vertex Pruning. *IEEE Trans. Parallel Distributed Syst.* 28, 3 (2017), 760–773.

[111] Siqiang Luo. 2019. Distributed PageRank Computation: An Improved Theoretical Study. In *AAAI 2019*. 4496–4503.

[112] Siqiang Luo. 2020. Improved Communication Cost in Distributed PageRank Computation - A Theoretical Study. In *ICML 2020 (Proceedings of Machine Learning Research, Vol. 119)*. 6459–6467.

[113] Siqiang Luo, Xiaowei Wu, and et al. 2022. Distributed PageRank computation with improved round complexities. *Inf. Sci.* 607 (2022), 109–125.

[114] Siqiang Luo and Zulun Zhu. 2023. Massively Parallel Single-Source SimRanks in o(log n) Rounds. *CoRR* abs/2304.04015 (2023). arXiv:2304.04015

[115] Xiongcai Luo, Jun Gao, and et al. 2017. UniWalk: Unidirectional Random Walk Based Scalable SimRank Computation over Large Graph. In *ICDE 2017*. 325–336.

[116] Tinghuai Ma, Mingliang Yue, and et al. 2018. PSPLPA: Probability and similarity based parallel label propagation algorithm on spark. *Physica A: Statistical Mechanics and its Applications* 503 (2018), 366–378.

[117] Saeed Maleki, Donald Nguyen, and et al. 2016. DSMR: a shared and distributed memory algorithm for single-source shortest path problem. In *PPoPP 2016*. 39:1–39:2.

[118] Grzegorz Malewicz, Matthew H. Austern, and et al. 2010. Pregel: a system for large-scale graph processing. In *SIGMOD 2010*. 135–146.

[119] Fragkiskos D. Malliaros, Christos Giatsidis, and et al. 2020. The core decomposition of networks: theory, algorithms and applications. *VLDB J.* 29, 1 (2020), 61–92.

[120] Aritra Mandal and Mohammad Al Hasan. 2017. A distributed k-core decomposition algorithm on spark. In *IEEE BigData 2017*. 976–981.

[121] Ali Mashreghi and Valerie King. 2021. Broadcast and minimum spanning tree with o(m) messages in the asynchronous CONGEST model. *Distributed Comput.* 34, 4 (2021), 283–299.

[122] Pragnesh Jay Modi, Wei-Min Shen, and et al. 2005. Adopt: asynchronous distributed constraint optimization with quality guarantees. *Artif. Intell.* 161, 1-2 (2005), 149–180.

[123] Alberto Montresor, Francesco De Pellegrini, and et al. 2013. Distributed k-Core Decomposition. *IEEE Trans. Parallel Distributed Syst.* 24, 2 (2013), 288–300.

[124] Philipp Moritz, Robert Nishihara, and et al. 2018. Ray: A Distributed Framework for Emerging AI Applications. In *OSDI 2018*. USENIX Association, 561–577.

[125] Hung Q. Ngo, Ely Porat, and et al. 2018. Worst-case Optimal Join Algorithms. *J. ACM* 65, 3 (2018), 16:1–16:40.

[126] Oracle. 2014. Remote Method Invocation Home. https://www.oracle.com

[127] Santosh Pandey, Zhibin Wang, and et al. 2021. Trust: Triangle Counting Reloaded on GPUs. *IEEE Trans. Parallel Distributed Syst.* 32, 11 (2021), 2646–2660.

[128] Roger A. Pearce, Maya B. Gokhale, and et al. 2014. Faster Parallel Traversal of Scale Free Graphs at Extreme Scale with Vertex Delegates. In *SC 2014,*. IEEE Computer Society, 549–559.

[129] Yun Peng, Byron Choi, and et al. 2016. VColor: A practical vertex-cut based approach for coloring large graphs. In *ICDE 2016*. 97–108.

[130] Yossi Peretz and Yigal Fischler. 2022. A fast parallel max-flow algorithm. *J. Parallel Distr. Com.* 169 (2022), 226–241.

[131] Usha Nandini Raghavan, Réka Albert, and et al. 2007. Near linear time algorithm to detect community structures in large-scale networks. *Physical review E* 76, 3 (2007), 036106.

[132] Emily A. Reed, Guilherme Ramos, and et al. 2023. A Scalable Distributed Dynamical Systems Approach to Learn the Strongly Connected Components and Diameter of Networks. *IEEE Trans. Autom. Control.* 68, 5 (2023), 3099–3106.

[133] Luis Remis, María Jesús Garzarán, and et al. 2018. Exploiting social network graph characteristics for efficient BFS on heterogeneous chips. *J. Parallel Distributed Comput.* 120 (2018), 282–294.

[134] Amitabha Roy, Ivo Mihailovic, and et al. 2013. X-Stream: edge-centric graph processing using streaming partitions. In *SOSP 2013*. 472–488.

[135] Václav Rozhon, Christoph Grunau, and et al. 2022. Undirected (1+ϵ)-shortest paths via minor-aggregates: near-optimal deterministic parallel and distributed algorithms. In *STOC 2022*. 478–487.

[136] Sherif Sakr, Anna Liu, and et al. 2013. The family of mapreduce and large-scale data processing systems. *ACM Comput. Surv.* 46, 1 (2013), 11:1–11:44.

[137] Semih Salihoglu and Jennifer Widom. 2013. GPS: a graph processing system. In *SSDBM 2013*. 22:1–22:12.

[138] Peter Sanders and Tim Niklas Uhl. 2023. Engineering a Distributed-Memory Triangle Counting Algorithm. In *IPDPS 2023*. 702–712.

[139] Ahmet Erdem Sariyüce, Erik Saule, and et al. 2013. STREAMER: A distributed framework for incremental closeness centrality computation. In *CLUSTER 2013*. 1–8.

[140] Atish Das Sarma, Anisur Rahaman Molla, and et al. 2015. Fast distributed PageRank computation. *Theor. Comput. Sci.* 561 (2015), 113–121.

[141] Naw Safrin Sattar and Shaikh Arifuzzaman. 2018. Parallelizing Louvain Algorithm: Distributed Memory Challenges. In *DASC/PiCom/DataCom/CyberSciTech 2018*. 695–701.

[142] Naw Safrin Sattar and Shaikh Arifuzzaman. 2022. Scalable distributed Louvain algorithm for community detection in large graphs. *J. Supercomput.* 78, 7 (2022), 10275–10309.

[143] Saeed Shahrivari and Saeed Jalili. 2021. Efficient Distributed k-Clique Mining for Large Networks Using MapReduce. *IEEE Trans. Knowl. Data Eng.* 33, 3 (2021), 964–974.

[144] Yingxia Shao, Lei Chen, and et al. 2014. Efficient cohesive subgraphs detection in parallel. In *SIGMOD 2014*. 613–624.

[145] Aneesh Sharma, C. Seshadhri, and et al. 2017. When Hashes Met Wedges: A Distributed Algorithm for Finding High Similarity Vectors. In *WWW 2017*. ACM, 431–440.

[146] Muhammad Shiraz, Abdullah Gani, and et al. 2013. A Review on Distributed Application Processing Frameworks in Smart Mobile Devices for Mobile Cloud Computing. *IEEE Commun. Surv. Tutorials* 15, 3 (2013), 1294–1313.

[147] Yogesh Simmhan, Alok Gautam Kumbhare, and et al. 2014. GoFFish: A Sub-graph Centric Framework for Large-Scale Graph Analytics. In *Euro-Par 2014 (Lecture Notes in Computer Science, Vol. 8632)*. 451–462.

[148] Nasrin Mazaheri Soudani, Afsaneh Fatemi, and et al. 2019. PPR-partitioning: a distributed graph partitioning algorithm based on the personalized PageRank vectors in vertex-centric systems. *Knowl. Inf. Syst.* 61, 2 (2019), 847–871.

[149] Trevor Steil, Tahsin Reza, and et al. 2021. TriPoll: computing surveys of triangles in massive-scale temporal graphs with metadata. In *SC 2021*. ACM, 67.

[150] András Strausz, Flavio Vella, and et al. 2022. Asynchronous Distributed-Memory Triangle Counting and LCC with RMA Caching. In *IPDPS 2022*. 291–301.

[151] Changhao Sun, Huaxin Qiu, and et al. 2022. Better Approximation for Distributed Weighted Vertex Cover via Game-Theoretic Learning. *IEEE Trans. Syst. Man Cybern. Syst.* 52, 8 (2022), 5308–5319.

[152] Michael Svendsen, Arko Provo Mukherjee, and et al. 2015. Mining maximal cliques from a large graph using MapReduce: Tackling highly uneven subproblem sizes. *J. Parallel Distributed Comput.* 79-80 (2015), 104–114.

[153] Nilothpal Talukder and Mohammed J. Zaki. 2016. A distributed approach for graph mining in massive networks. *Data Min. Knowl. Discov.* 30, 5 (2016), 1024–1052.

[154] Carlos H. C. Teixeira, Alexandre J. Fonseca, and et al. 2015. Arabesque: a system for distributed graph mining. In *SOSP 2015*. 425–440.

[155] CORPORATE The MPI Forum. 1993. MPI: a message passing interface. In *Proceedings of SC*. 878–883.

[156] Yuanyuan Tian, Andrey Balmin, and et al. 2013. From "Think Like a Vertex" to "Think Like a Graph". *Proc. VLDB Endow.* 7, 3 (2013), 193–204.

[157] Etsuji Tomita, Akira Tanaka, and et al. 2004. The Worst-Case Time Complexity for Generating All Maximal Cliques. In *COCOON 2004 (Lecture Notes in Computer Science, Vol. 3106)*. 161–170.

[158] Ankit Toshniwal, Siddarth Taneja, and et al. 2014. Storm@ twitter. In *Proceedings of SIGMOD*. 147–156.

[159] Philip W. Trinder, Hans-Wolfgang Loidl, and et al. 2002. Parallel and Distributed Haskells. *J. Funct. Program.* 12, 4&5 (2002), 469–510.

[160] Christian Wallmann and Markus Gerschberger. 2020. The association between network centrality measures and supply chain performance: The case of distribution networks. In *ISM 2020*, Vol. 180. Elsevier, 172–179.

[161] Yuanwei Wang, Huanqi Cao, and et al. 2022. Scaling Graph 500 SSSP to 140 Trillion Edges with over 40 Million Cores. In *SC 2022*. 19:1–19:15.

[162] Yue Wang, Ruiqi Xu, and et al. 2020. DISK: A Distributed Framework for Single-Source SimRank with Accuracy Guarantee. *Proc. VLDB Endow.* 14, 3 (2020), 351–363.

[163] Zhaokang Wang, Rong Gu, and et al. 2019. BENU: Distributed Subgraph Enumeration with Backtracking-Based Framework. In *ICDE 2019*. 136–147.

[164] Klaus Wehmuth and Artur Ziviani. 2013. DACCER: Distributed Assessment of the Closeness Centrality Ranking in complex networks. *Comput. Networks* 57, 13 (2013), 2536–2548.

[165] Tongfeng Weng, Xu Zhou, and et al. 2022. Efficient Distributed Approaches to Core Maintenance on Large Dynamic Graphs. *IEEE Trans. Parallel Distributed Syst.* 33, 1 (2022), 129–143.

[166] Tom White. 2012. *Hadoop: The definitive guide.* " O'Reilly Media, Inc.".

[167] Lizhi Xiang, Arif Khan, and et al. 2021. cuTS: scaling subgraph isomorphism on distributed multi-GPU systems using trie based data structure. In *SC 2021*. 69.

[168] Yanyan Xu, James Cheng, and et al. 2016. Distributed Maximal Clique Computation and Management. *IEEE Trans. Serv. Comput.* 9, 1 (2016), 110–122.

[169] Yanyan Xu, James Cheng, and et al. 2014. Distributed Maximal Clique Computation. In *IEEE BigData 2014*. 160–167.

[170] Da Yan, James Cheng, and et al. 2014. Blogel: A Block-Centric Framework for Distributed Computation on Real-World Graphs. *Proc. VLDB Endow.* 7, 14 (2014), 1981–1992.

[171] Da Yan, James Cheng, and et al. 2015. Effective Techniques for Message Reduction and Load Balancing in Distributed Graph Computation. In *WWW 2015*. 1307–1317.

[172] Zhengyi Yang, Longbin Lai, and et al. 2021. HUGE: An Efficient and Scalable Subgraph Enumeration System. In *SIGMOD 2021*. 2049–2062.

[173] Abdurrahman Yasar, Sivasankaran Rajamanickam, and et al. 2022. A Block-Based Triangle Counting Algorithm on Heterogeneous Environments. *IEEE Trans. Parallel Distributed Syst.* 33, 2 (2022), 444–458.

[174] Ziqiang Yu, Xiaohui Yu, and et al. 2020. Distributed Processing of k Shortest Path Queries over Dynamic Road Networks. In *SIGMOD 2020*. 665–679.

[175] Matei Zaharia, Mosharaf Chowdhury, and et al. 2012. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *NSDI 2012*. 15–28.

[176] Matei Zaharia, Mosharaf Chowdhury, and et al. 2010. Spark: Cluster Computing with Working Sets. In *HotCloud 2010*.

[177] Matei Zaharia, Reynold S. Xin, and et al. 2016. Apache Spark: a unified engine for big data processing. *Commun. ACM* 59, 11 (2016), 56–65.

[178] Jianping Zeng and Hongfeng Yu. 2018. A Scalable Distributed Louvain Algorithm for Large-Scale Graph Community Detection. In *CLUSTER 2018*. 268–278.

[179] Dongxiang Zhang, Dingyu Yang, and et al. 2017. Distributed shortest path query processing on dynamic road networks. *VLDB J.* 26, 3 (2017), 399–419.

[180] Tian Zhou, Lixin Gao, and et al. 2021. A Fault-Tolerant Distributed Framework for Asynchronous Iterative Computations. *IEEE Trans. Parallel Distributed Syst.* 32, 8 (2021), 2062–2073.

[181] Huanzhou Zhu, Ligang He, and et al. 2020. WolfGraph: The edge-centric graph processing on GPU. *Future Gener. Comput. Syst.* 111 (2020), 552–569.

[182] Goran Zuzic, Gramoz Goranci, and et al. 2022. Universally-Optimal Distributed Shortest Paths and Transshipment via Graph-Based ℓ_1-Oblivious Routing. In *SODA 2022*. 2549–2579.

Received 29 February 2024; revised 16 July 2024; accepted 22 August 2024