Large-scale Graph Analytics and Frameworks

Area Exam Position Paper

Sudharshan Srinivasan
University of Oregon, Eugene, OR, USA
ssriniv2@cs.uoregon.edu

Abstract—**Graph analytics is a vital field of research for representing relations between different entities and understanding patterns of interactions for large groups. The scope of large-scale graph analytics is complex enough that there exist numerous challenges and a plethora of frameworks, with each addressing a subset of challenges. In this paper, we explore the area of graph analytics and its available frameworks. In specific, we discuss the topology of graph algorithms, their applications, and their drawbacks. We then explore the existing analytics frameworks for solving these algorithms in two broad classes. We first classify them based on the target architecture, and then we classify them based on the type of workload they address. Since the performance of analytics is highly sensitive to the nature of the problem, there isn't a single framework that addresses all classes of graph problems. This paper can help readers gain a better understanding of graph analytics and provides general guidance for choosing the modeling methods and the right frameworks that are suitable to particular graph problems.**

I. Introduction

Large-scale graph analytics require analyzing and extracting insights from vast and intricate graph data structures, which consist of nodes (vertices) and edges (connections) representing complex relationships between entities. This approach is indispensable in contemporary data-driven applications across diverse domains. As the volume and complexity of data continue to surge, large-scale graph analytics faces several challenges and considerations. Scalability is paramount, necessitating the development of scalable algorithms and distributed computing frameworks to handle graphs with millions or billions of nodes and edges. Efficient data storage models, such as adjacency lists or graph databases, are crucial for managing such extensive graph data.

Parallel processing, enabled by distributed computing frameworks like Apache Spark and Hadoop, is a cornerstone of large-scale graph analytics. These frameworks distribute computations across multiple nodes or machines, ensuring efficient analysis. A wide array of graph algorithms, from fundamental ones like BFS and DFS to advanced methods like centrality measures and community detection, are employed to extract valuable insights from the data. Efficient graph traversal and memory management are critical concerns, with techniques like memory mapping and graph compression used to optimize resource usage.

Distributed graph processing involves partitioning large graphs into smaller subgraphs, striking a balance between workload distribution and communication overhead. Real-time analytics are essential in dynamic environments, such as social media monitoring and fraud detection. Specialized graph databases offer efficient querying and analytics capabilities for large-scale graph data. Visualization tools aid in comprehending complex graph structures, while integrating machine learning techniques with graph analytics enables tasks like node classification and anomaly detection. Large-scale graph analytics plays a pivotal role in solving optimization problems, ensuring privacy and security, and making data-driven decisions across an array of domains as data complexity and volume continue to rise.

In this survey, we aim to categorize and summarize the literature on large-scale graph analytics frameworks and the architecture behind their models. In Section II, we introduce the landscape of graph algorithms and the categorization of edge and vertex-centric algorithms. In Section III, we discuss the various available frameworks classified with respect to the target architecture. In Section IV, we explore the concepts of dynamic graphs and the available frameworks for dynamic graph analytics.

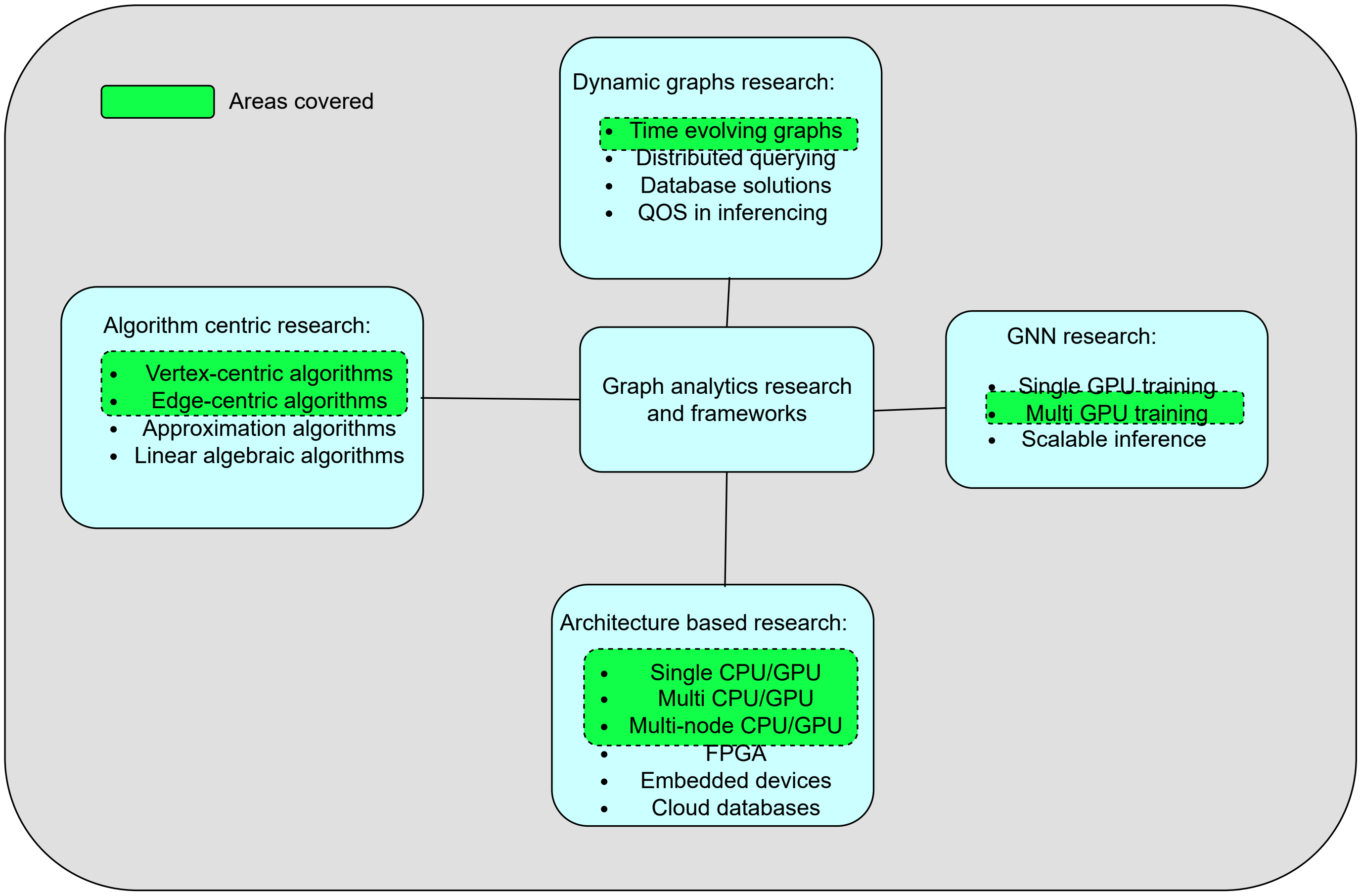

Fig. 1. Topics for graph analytics frameworks and research in literature. The scope of this survey extends to the highlighted areas.

In Section V, we explore distributed training frameworks for GNNs and GATs. We finally conclude on the current state and possible future directions of the field.

II. Landscape of graph algorithms

Large-scale graph algorithms can broadly be classified into three different approaches based on the view taken to solve these algorithms.

A. *Vertex-centric*

Vertex-centric programming models process graphs using vertices as the primary units of computation. In this model, each vertex is responsible for executing a program that updates its own state and sends messages to its neighbors. The overall algorithm proceeds iteratively, with each vertex executing its program and sending messages in each iteration. Vertex-centric graph processing has several advantages over other programming models for graph processing. First, the vertex-centric programming model is very intuitive and easy to understand. Each vertex only needs to worry about its own state and its neighbors. This makes it much simpler to program than other graph programming models, such as edge-centric programming. Second, the vertex-centric programming model is easily scalable to large graphs. This is because the vertices can be easily distributed across multiple processors. This makes it a good choice for processing large graphs that cannot fit in the memory of a single machine. Third, the vertex-centric programming model can be optimized to minimize the amount of communication between vertices. This is done by only sending messages to the vertices that need them. This can significantly improve the algorithm's performance, especially for large graphs. Overall, vertex-centric graph processing is a powerful and versatile programming model for processing graphs. It is simple to program, scalable, and efficient. These advantages make it a good choice for a wide variety of graph processing applications. Vertex-centric graph algorithms are usually referred to as Think-Like-A-Vertex(TLAV) algorithms.

Although vertex-centric algorithms are local and bottom-up, they have a provable, global result. TLAV frameworks are heavily influenced by distributed algorithms theory, including synchronicity and communication mechanisms [1]. Several distributed algorithm implementations, such as distributed Bellman-Ford single-source shortest path [1], are used as benchmarks throughout the TLAV literature. The introduction of TLAV frameworks has also spurred the adaptation of many popular Machine Learning and Data Mining (MLDM) algorithms into graph representations for high-performance TLAV processing of large-scale datasets [2].

Many graph problems can be solved using either a sequential or distributed approach. For example, the PageRank algorithm for calculating webpage importance has a centralized matrix form [3] as well as a distributed, vertex-centric form [4]. The sequential approach is often easier to implement, but it may not be as scalable to large graphs. The distributed approach is more scalable, but it may be more complex to implement. The best approach to use depends on the specific problem and the available computing resources.

The choice of approach will depend on the specific problem and the available computing resources. For example, if the graph is small and the computing resources are limited, then the sequential approach may be the best choice. However, if the graph is large and the computing resources are abundant, then the distributed approach may be the best choice.

Vertex programs, in contrast, only depend on data local to a vertex and reduce computational complexity by increasing communication between program kernels. As a result, TLAV frameworks are highly scalable and inherently parallel, with manageable inter-machine communication. For example, runtime on the Pregel framework has been shown to scale linearly with the number of vertices on 300 machines [4]. Furthermore, TLAV frameworks provide a common interface for vertex program execution, abstracting away low-level details of distributed computation, like MPI, allowing for a fast, reusable development environment. A paradigm shift from centralized to decentralized approaches to problem-solving is represented by TLAV frameworks.

TLAV frameworks can first be classified based on the timing of execution into :

Synchronous: In this model, active vertices are executed conceptually in parallel over one or more iterations, called supersteps. Synchronization is achieved through a global synchronization barrier situated between each superstep that blocks vertices from computing the next superstep until all workers complete the current superstep. Each worker coordinates with the master to progress to the next superstep. Within a single processing unit, vertices can be scheduled in a fixed or random order because the execution order does not affect the state of the program[5].

Although synchronous systems are conceptually straightforward and scale well, the model has drawbacks. One study found that synchronization, for an instance of finding the shortest path in a highly partitioned graph, accounted for over 80% of the total running time [6], so system throughput must remain high to justify the cost of synchronization since such coordination can be relatively costly.

However, when the number of active vertices drops or the workload among workers becomes imbalanced, system resources can become underutilized. Iterative algorithms often suffer from “the curse of the last reducer,” otherwise known as the “straggler” problem, where many computations finish quickly, but a small fraction of computations take a disproportionately longer amount of time [7]. For synchronous systems, each superstep takes as long as the slowest vertex, so synchronous systems generally favor lightweight computations with small variability in runtime.

Finally, synchronous algorithms may not converge in some instances. In graph coloring algorithms, for example, vertices attempt to choose colors different from adjacent neighbors [8] and require coordination between neighboring vertices. However, during synchronous execution, the circumstance may arise where two neighboring vertices continually flip between each other's colors. In general, algorithms that require some type of neighbor coordination may not always converge with the synchronous timing model without the use of some extra logic in the vertex program [5].

Asynchronous: In the asynchronous iteration model, vertices can be executed at any time as long as there are available processor and network resources. This eliminates the ”straggler” problem, where a few slow vertices can hold up the entire computation. However, asynchronous execution can be more complex to implement and maintain, and it can also lead to redundant communication and excessive computation.

Research has shown that asynchronous execution can generally outperform synchronous execution [2], especially for imbalanced workloads. However, the performance gains can vary depending on the specific algorithm and the properties of the system.

Asynchronous systems typically use a pull model of execution, where each vertex only pulls data from its neighbors when it needs it [9]. This can help to reduce redundant communication. However, it can also lead to data races, where two vertices try to update the same data at the same time. This can be a challenge to avoid, and it can require additional mechanisms to ensure data consistency.

Overall, asynchronous execution provides more flexibility and can be more efficient for certain workloads. However, it also comes with some additional complexity and challenges [10] [2].

B. Edge-centric

The edge-centric graph model is a computational approach that focuses primarily on the edges of a graph rather than the vertices. It is a different perspective compared to the more common vertex-centric approach. Edge-centric computations often exhibit more sequential access patterns compared to vertex-centric models. This can lead to more efficient memory access and improved performance in some cases. Edge-centric models are particularly suited for algorithms that involve traversing relationships between edges, such as certain network analyses, recommendation systems, and certain types of graph clustering algorithms. Edge-centric models can also help reduce redundancy in computations, as you can perform operations directly on edges rather than repeatedly traversing vertices to access their edges. This can lead to more efficient algorithms. Some edge-centric algorithms can be easier to parallelize because the focus is on edges, which can be processed independently in many cases. This can take advantage of multi-core processors or distributed computing environments.

Select frameworks for edge-centric models are proposed in the literature. X-Stream [11] employs a graph computation model centered around edges. When compared to a vertex-centric approach, edge-centric access tends to be more sequential, even though traversing edges can still create a somewhat random and unpredictable access pattern. Additionally, running algorithms that follow vertices or edges typically leads to random access to the storage medium for the graph. This can frequently be the decisive factor in determining performance, regardless of the algorithm's complexity or its efficiency during runtime.

Pathgraph[12] is another framework that uses an edge-centric approach for its graph algorithms. They represent a sizable graph by employing a set of tree-based divisions and opt for a path-oriented approach instead of the more common vertex-centric or edge-centric methods. Initially, the parallel computation model brings about notable enhancements in memory and disk locality, particularly when executing iterative algorithms. Secondly, they create a streamlined storage system that goes a step further in optimizing sequential access while reducing random access on storage devices. Lastly, they put the path-oriented computation model into action by utilizing a scatter/gather programming approach. This approach parallelizes iterative computations at the partition tree level and carries out sequential updates for vertices within each partition tree.

Wolfgraph[13] updates on works from X-stream[11] by introducing a GPU-based graph framework. The data structure and graph partitioning in WolfGraph have been meticulously designed to reduce graph pre-processing efforts and enable efficient memory access consolidation. WolfGraph maximizes GPU utilization by concurrently processing all graph edges. Additionally, they've introduced a novel approach called ConcatenatedEdge-List (CEL) to handle graphs larger than the GPU's global memory capacity. With WolfGraph, users have the flexibility to define and integrate their custom graph processing methods seamlessly into the WolfGraph framework.

GraphChi[14] employs an innovative out-of-core data structure known as ”sharding” to minimize the need for random access to the hard disk. Before computations begin, GraphChi conducts an initial preprocessing of the graph data. This involves partitioning the input data into sub-graphs, referred to as ”shards.” Each shard consists of a set of vertices and all the incoming edges connected to these vertices. Within each shard, the edges are organized in ascending order based on the source vertex ID. The partitioning method employed by GraphChi ensures that the number of edges in each shard is roughly uniform, and the size of each shard is designed to fit within the available memory.

GraphChi[14] has also developed a technique called ”parallel sliding windows” (PSW). During computation, GraphChi loads the first shard into memory and then efficiently retrieves and loads the out-edges (where the source vertex is located in other shards) of the current shard from other shards into memory as needed. Once processing of the current shard is complete, it moves on to the next shard and repeats this process. The entire computation concludes when all shards have been processed. This approach of organizing the graph into shards and utilizing PSW ensures sequential reading from the hard disk, thereby optimizing the performance of hard disk I/O.

In summary, we have discussed the landscape of various graph algorithms by categorizing them as vertex-centric and edge-centric models. Vertex-centric models gain an advantage for their simplicity, ease of parallelism, and efficient data access. They are subcategorized as synchronous and asynchronous models based on their implementation's blocking or non-blocking nature. On the other hand, edge-centric models gain an advantage for their fine-grained control, reduced communication overhead, and efficiency for irregular graphs. Beyond this classification, they can further be categorized as linear algebraic and approximate models that are out of the scope of this survey.

III. Graph frameworks classified by architecture

Graph analytics programming models provide a way to specify how to analyze a graph. They typically include a set of operators that can be used to perform operations on graphs, such as finding paths, counting connected components, and finding communities. Graph analytics runtime systems provide a way to execute graph analytics programs. They typically include a graph storage engine, a graph processing engine, and a graph visualization engine.

Several frameworks exist for graph analytics on various target architectures.

A. Shared-memory CPU and single-GPU systems

Galois[16], [23], [24] is the state-of-the-art graph analytics framework for multi-core NUMA machines. It is designed to ease parallel programming, especially for applications with irregular parallelism and communication. Ligra [22], and Polymer [25] are similar analytics frameworks for multi-core NUMA machines. All three of these frameworks perform much better than existing distributed frameworks when the graph fits within a single node, but not for large-scale out-of-node graphs.

Various single GPU frameworks also exist for ease of programming analytics. Graphie [26] is a single GPU framework that stores the vertex attribute data in the GPU memory and streams edge data asynchronously to the GPU for processing. MultiGraph [27], [28] uses multiple data representation and execution strategies for dense versus sparse vertex frontiers. It also allows users with access to GPU configuration to fine-tune the warp counts. Novel representation techniques have also been extensively studied for single GPU implementations. CuSHA [29] presents a framework that uses a concept recently introduced for non-GPU systems that organizes a graph into autonomous sets of ordered edges called shards. It also presents another representation that enhances the use of shards to achieve higher GPU utilization for processing sparse graphs. Gunrock [30] implements a novel data-centric abstraction centered on operations on a vertex or edge frontier. It is a high-level programming model that allows programmers to quickly develop new graph primitives with small code size and minimal GPU programming knowledge. There also

TABLE I
SUMMARY OF FEATURES FOR SELECT GRAPH FRAMEWORKS ACROSS ARCHITECTURES

Framework	CPU-GPU hybrid	Distributed	Asynchronous	BSP	Rich API	Directed graph
GraphX[15]	✓	✓	✗	✓	✗	✓
Pregel[4]	✓	✓	✗	✓	✓	✗
Galois[16]	✗	✗	✗	✓	✓	✗
Groute[17]	✗	✓	✓	✗	✗	✗
Garaph[18]	✗	✓	✗	✓	✓	✗
Mizan[19]	✓	✓	✗	✗	✗	✓
PowerGraph [20]	✓	✓	✗	✓	✓	✗
Medusa[21]	✓	✗	✗	✓	✗	✗
Ligra[22]	✗	✗	✗	✓	✗	✗

exists wrapper compilers like IrGL [31] that produce CUDA code from an intermediate-level program representation. Several algorithms [32], [33], [34] have shown that GPUs can be efficiently utilized for irregular computations.

B. *Single-node multi GPUs and heterogeneous systems*

Several frameworks and libraries exist for graph processing on multiple GPUs. Groute [17] provides constructs for asynchronous multi-GPU programming and describes their implementation in a thin runtime environment. Medusa [21] is another multi-GPU framework that enables developers to leverage the capabilities of GPUs by writing sequential C/C++ code. It offers a small set of user-defined APIs and embraces a runtime system to automatically execute those APIs in parallel on the GPU. Other frameworks like [35], [36], [37] provide libraries that allow programmers to easily extend single-GPU graph algorithms to achieve scalable performance on large graphs with billions of edges.

There also exist several multi-GPU frameworks that combine with CPUs for heterogeneous computations. Garaph [18] proposes a vertex replication degree customization scheme that maximizes the GPU utilization given vertices, degrees, and space constraints. It also adopts a balanced edge-based partition ensuring work balance over CPU threads, and also a hybrid of notify-pull and pull computation models optimized for fast graph processing on the CPU. Lastly, Garaph uses a dynamic workload assignment scheme that takes into account both the characteristics of processing elements and graph algorithms. Similarly, Falcon [38] is a domain-specific language(DSL) for implementing graph algorithms that abstracts the hardware and provides constructs to write explicitly parallel programs at a higher level. It can work with general algorithms that may change the graph structure.

Apart from groute[17], which is an Asynchronous system, all the other frameworks use BSP-style synchronization. It is important to note that these systems are still limited to a single machine and struggle to handle large-scale graphs in the order of 10 billion edges or more.

C. *Multi-node systems*

The ability to scale an application to multiple processes and machines is paramount for large-scale graph analytics with billions of vertices and edges. Many graph frameworks and libraries exist for processing on distributed systems. Gemini [39] is the state-of-the-art distributed graph processing system that applies multiple optimizations targeting computation performance to build scalability on top of efficiency. It incorporates a chunk-based partitioning scheme enabling low-overhead scaling out designs while maintaining a dual representation scheme to compress accesses to vertex indices. LF-Graph [40] offers low and balanced communication and computation, low preprocessing overhead, low memory footprint, and scalability for distributed graph analytics.

Pregel [4] approaches large-scale graph processing by expressing programs as sequences of iterations for computations and communication phases. Mizan [19] additionally extends the pregel framework by utilizing efficient load balancing techniques depending on the workload characteristics. GraphX [15] is a distributed framework that addresses the challenges of graph construction and transformation. It combines the advantages of both data-parallel and graph-parallel systems by efficiently expressing graph computation within the Spark data-parallel framework. PowerGraph [20] framework provides abstraction, which exploits the internal structure of graph programs to address challenges with power-law graphs.

To summarize this section, we have explored the different graph frameworks in the literature by categorizing them based on their target architecture. Single memory CPU and GPU frameworks like CuSHA [29], Ligra[22], Gunrock[30] and Galois[16], [23], [24] provide high-performance APIs that leverage shared memory for efficiency but tradeoff scalability and memory utilization when encountering larger graphs. Single-node multi-GPU frameworks like Garaph[18] and groute[17] offer improved memory utilization but still have a bottleneck when scaling to very large graphs as they still rely on rapid interconnects like NVlink[41]. Lastly, multi-node frameworks like Pregel [4], Mizan [19] and GraphX[15] offer great scalability and efficiency for large-scale graph algorithms but lack the same performance as single-node frameworks when the size of the graph is small enough to fit within memory as synchronizing data across ranks adds excess overheads.

IV. Frameworks for dynamic graphs

A dynamic graph is a graph whose topology can change over time. This means that vertices and edges can be added or removed, and the relationships between vertices can change. Dynamic graphs are often used to model real-world systems, such as social networks, transportation networks, and financial markets.

There are a number of challenges associated with processing dynamic graphs. One challenge is that the graph topology can change frequently, which can make it difficult to keep track of the latest changes. Another challenge is that the graph can be very large and complex, which can make it difficult to process efficiently. There are a number of basic dynamic graph processing frameworks available, such as Pregel[4], GraphLab[2], and Apache Giraph[42]. These frameworks provide a number of tools and algorithms for processing dynamic graphs.

The choice of a dynamic graph processing framework depends on the specific application. Some factors to consider include the size and complexity of the graph, the frequency of changes to the graph topology, and the desired performance.

Various models and frameworks have been proposed in the literature for dynamic graph processing. A dynamic graph model is a mapping $G_t = (V, E)$ that yields the state of the graph (i.e., the set of nodes and set of edges) at a given time instant t. Both directed and undirected dynamic graphs can be represented by most of the existing discrete and continuous models.

In a discrete model, snapshots are taken periodically at every fixed time period (e.g., every 30 minutes, every day, and every week). This type of model provides complete, accurate mapping at specific time instants and gives the nearest state (e.g., time-based, changes-based) at any other instant.

On the other hand, the continuous model keeps track of all changes by representing everyone of them. Therefore, it can map every instant into a completely accurate valid graph state.

Based on the nature of how the models ingest their input, we can categorize dynamic graph frameworks as coarse-grained snapshot-based models that input updated information on batches at defined intervals or as fine-grained streaming models that continuously update the graph as new updates arrive.

A. Snapshot-based models

These models store a sequence of snapshots of the graph, where each snapshot represents the state of the graph at a particular time instant. The snapshots can be taken at regular intervals or irregular intervals, depending on the application. Varieties of this category have been proposed in Rossi's model [43], FVF [44], and Yang's model [45]. This category models dynamic graphs as a sequence of snapshots $G[_{1,2}] = \{G_1, G_2, G_3, ...G_n\}$. Each snapshot G_i is a static graph that represents the valid state of the dynamic graph at time point t_i. The snapshot is represented by a triple $\{V_i, E_i, T_i\}$ and is stored by its time point t_i.

Storing a sequence of snapshots naively, as in Yang's model [45] and Rossi's model [43], would clearly require a prohibitively large storage.

FVF [44] proposes the "Find-Verify-and-Fix" (FVF) framework, which takes a sequence of snapshots that are produced in a compressed storage model as input. This compressed storage model stores a set of key snapshots and the associated set of deltas. The set of key snapshots is intended to be much smaller than the original set of all snapshots. A set of deltas stores only the changes that are needed to completely construct a snapshot from its related key snapshot by merging the key snapshot with the proper delta. Compressed Storage Models (SMs) have been discussed in FVF [44] for storing these clusters. However, the most efficient one is called SM-FVF. It saves four deltas for each cluster C, which has k snapshots.

In conclusion, the compressed model in FVF [44] is more efficient with regard to the used storage than the naive model described by Yang [43] and Rossi [43]. However, the naive model is faster than the compressed model in query performance due to the consumed construction time in the compressed model. The compressed model in FVF stores only the changes that are needed to construct a snapshot from its related key snapshot. This can significantly reduce the amount of storage required. However, the construction of the compressed model can be time-consuming. The naive model stores all snapshots of the graph. This can lead to a large amount of storage, but the query performance is much better than the compressed model.

We can further categorize snapshot-based models on their representation of their input graph.

1) CSR representation: STINGER[46] is a data structure and software framework that adapts and extends the CSR format to support graph updates. Unlike the static CSR design, where the IDs of the neighbors of a given vertex are stored contiguously, neighbor IDs in STINGER are divided into contiguous blocks of a pre-selected size. These blocks form a linked list, so STINGER uses a blocking design. The block size is identical for all blocks except for the last blocks in each list. One neighbor vertex ID in the neighborhood of a vertex v corresponds to one edge (v, u).

STINGER supports both vertices and edges with different types. One vertex can have adjacent edges of different types. One block always contains edges of one type only. In addition to the associated neighbor vertex ID and type, each edge has its weight and two timestamps. The timestamps can be used in algorithms to filter edges, for example, based on the insertion time. In addition to this, each edge block contains certain metadata, for example, the lowest and highest timestamps in a given block.

STINGER also provides the edge type array (ETA) index data structure. ETA contains pointers to all blocks with edges of a given type to accelerate algorithms that operate on specific edge types.

To increase parallelism, STINGER updates a graph in batches. For graphs that are not scale-free, a batch of around 100,000 updates is first sorted so that updates to different vertices are grouped together. In the process, deletions may be separated from insertions (they can also be processed in parallel with insertions). For scale-free graphs, there is no sorting phase since a small number of vertices will face many updates which leads to workload imbalance. Instead, each update is processed in parallel. Fine-locking on single edges is used for synchronization of updates to the neighborhood of the same vertex.

To insert an edge or to verify if an edge exists, one traverses a selected list of blocks, taking O(d) time. Consequently, inserting an edge into a graph with N vertices takes O(Nd) work and depth. STINGER is optimized for the Cray XMT supercomputing systems that allow for massive thread-level parallelism. However, it can also be executed on general multi-core commodity servers.

Unlike other works, STINGER and its variants do not provide a framework but a library to operate on the data structure. Therefore, the user is in full control, for example, to determine when updates are applied and what programming model is used.

DISTINGER[47]represents a distributed variant of STINGER designed for "shared-nothing" commodity clusters. DISTINGER builds upon the STINGER architecture but introduces several key changes.

To begin with, it employs a designated master process to facilitate communication between the DISTINGER instance and external systems. This master process translates external vertex IDs at the application level into the internal IDs utilized within DISTINGER. Additionally, the master process maintains a roster of slave processes and del-

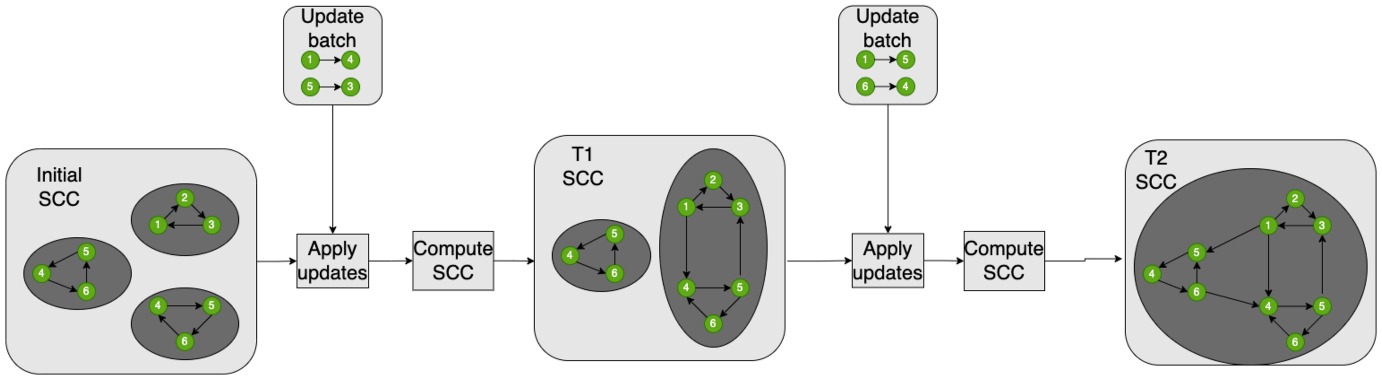

Fig. 2. Workflow of snapshot-based models

egates incoming queries and updates to the appropriate slaves responsible for managing the relevant section of the processed graph. Each slave is tasked with maintaining and updating a portion of the vertices along with their associated edges. The graph itself is divided using a straightforward hash-based approach. Communication between processes is accomplished through MPI, utilizing established optimizations like message batching and the concurrent execution of computation and communication tasks.

cuSTINGER[48] extends STINGER for CUDA GPUs. The main design change is to replace lists of edge blocks with contiguous adjacency arrays, i.e., a single adjacency array for each vertex. Moreover, contrary to STINGER, cuSTINGER always separately processes updates and deletions to better utilize massive parallelism in GPUs. cuSTINGER offers several "meta-data modes": based on the user's needs, the framework can support only unweighted edges, weighted edges without any additional associated data, or edges with weights, types, and additional data such as timestamps. However, the paper focuses on unweighted graphs that do not use timestamps and types, and the exact GPU design of the last two modes is unclear.

The system developed by Monda et al.[49] places its focus on three key aspects: data replication, graph partitioning, and load balancing. Consequently, it operates in a distributed manner. On each computing node, a replication manager makes localized decisions, primarily by analyzing graph queries, to determine which vertex should be replicated and on which computing nodes its copies should be stored.

The primary contribution of this system lies in its introduction of a fairness criterion. This criterion mandates that, at the very least, a certain configurable fraction of neighboring vertices must be replicated on some computing node. This approach serves to alleviate strain on network bandwidth and enhances the responsiveness of queries that require fetching neighborhoods, which is a common requirement in social network analysis.

As for data storage, the framework utilizes Apache CouchDB [50], an in-memory key-value store. However, specific details about how the data is represented are not provided.

LLAMA [51], which stands for Linked-node analytics using Large Multiversioned Arrays, shares similarities with STINGER in how it processes graph updates in batches. However, it distinguishes itself from STINGER by generating a new snapshot of graph data for each batch using a copy-on-write approach.

Specifically, LLAMA represents the graph using a variant of CSR (Compressed Sparse Row) that relies on large multiversioned arrays. In contrast to CSR, the array that maps vertices to per-vertex structures is divided into smaller parts known as data pages. Each data page can belong to a different snapshot and contains pointers to the single edge array that stores graph edges. To create a new snapshot, new data pages and a new edge array are allocated to hold the delta representing the update. This design allows older snapshots to share some data pages and parts of the edge array, enabling lightweight updates. For instance, if there is a batch of edge insertions into the neighborhood of vertex v, this batch may become a part of v's adjacency list within a new snapshot but only represents the update and relies on the old graph data. Contiguous allocations are employed for all data structures to enhance allocation and access efficiency.

LLAMA also places a strong emphasis on out-of-memory graph processing. To achieve this, snapshots can be persisted on disk and mapped into memory using mmap. The system is implemented as a library, which means users are responsible for ingesting graph updates and can employ a programming model of their choice. LLAMA does not impose any specific programming model but offers a simple API for iterating over the neighbors of a given vertex v, whether they are the most recent ones or belong to a specific snapshot.

2) Tree representation: The Aspen framework, as described in reference [52], employs an innovative data structure known as the C-tree to store graph structures. This C-tree is built upon a purely functional compressed search tree. In this context, a functional search tree is a data structure for searching that can be expressed solely through mathematical functions. This unique characteristic makes the data structure immutable since a mathematical function consistently produces the same result for the same input, regardless of any associated state.

Functional search trees offer several advantages, including lightweight snapshots, provably efficient execution times, and support for concurrent processing of queries and updates. The C-tree takes the concept of purely functional search trees further by addressing issues related to poor space utilization and low locality.

In the C-tree, elements represented by the tree are stored in chunks, and each chunk is stored contiguously in an array, resulting in improved data locality. To optimize space usage, chunks can be compressed using difference encoding, as each block holds a sorted set of integers.

A graph is represented as a tree of trees: a purely functional tree stores the set of vertices (vertex-tree), and each vertex stores its edges in its own C-tree (edge-tree). Additional information is stored in the vertex-trees, allowing for quick querying of fundamental graph properties, such as the total number of edges and vertices, in constant time. While the trees can be extended to store properties like weights, this aspect is not covered in the described work.

For algorithms that operate on the entire graph, such as Breadth-First Search (BFS), it is possible to precompute a flat snapshot. Instead of accessing all vertices by querying the vertex-tree, an array is used to directly store pointers to the vertices. This approach incurs some initial overhead but reduces access times to edges, ultimately decreasing the execution times of various algorithms.

The framework does not impose a specific programming model. Its API allows for any number of parallel readers and a single writer. Readers and writers are never blocked, and the framework ensures strict serializability. Updater routines enable both the addition and removal of edges or vertices, and they are applied in batches, not exposed to running algorithms. Instead, algorithms operate on an immutable snapshot.

Tegra [53] enables graph analysis based on graph updates within any defined time window. This means that Tegra must maintain the complete history of the graph, unlike many other systems that often store only a single state (and the snapshots upon which graph algorithms are executed). Consequently, this system faces distinct challenges: it needs to share graph data across different time windows and synchronize state among concurrently running queries.

To achieve these objectives, Tegra relies on a novel computation model known as the Incremental Computation by entity Expansion (ICE) model. Many graph algorithms are iterative and converge to a solution, allowing the reuse of certain parts of the previous solution when the graph is updated. While others have already addressed such algorithms, they are often limited to graph expansion (i.e., no removals are allowed) to ensure correctness. ICE extends this approach and recomputes graph algorithms on the subgraphs affected by the recomputation. Therefore, removals of vertices and edges can also be considered. As tracking state and subsequent recomputation may result in high overhead, a cost model is employed, and the framework switches to full recomputation when necessary.

To support the ICE model, Tegra's core data structure is an adaptive radix tree, a tree data structure that facilitates efficient updates and range scans. It efficiently maps a graph by storing it in two trees (a vertex tree and an edge tree) and generates lightweight snapshots by creating a new root node that holds the differences. For scalability, the graph is partitioned among compute nodes based on the hash of the vertex ID. Users can interact with Tegra through the provided API and can manually create new snapshots of the graph. The system can also automatically create snapshots when a certain limit of changes is reached. Consequently, queries and updates (which can be ingested from main memory or graph databases) can run concurrently. The framework also stores the changes that occurred between snapshots, allowing the restoration of any state and the application of computations on any window.

As snapshots consume substantial memory, they are written to disk using the least recently used (LRU) policy. The framework is implemented on top of Apache Spark, which handles scheduling and work distribution.

B. Continues streaming models

These models are also referred to as fine-grained synchronization, which differs from coarse-grained synchronization(snapshot based models) where updates are merged with the main graph representation during specific phases by the fact that updates are integrated into the main dataset as soon as they arrive. This process often involves interleaving updates with queries and relies on synchronization protocols based on fine-grained locks and/or atomic operations. An example of fine-grained synchronization is Differential Dataflow[54], where the ingestion strategy allows for concurrent updates and queries by leveraging a combination of logical time to maintain knowledge of updates (referred to as deltas) and progress tracking. Specifically, in the design of differential dataflow, collections of key-value pairs enriched with timestamps and delta values are used, and dynamic data is viewed as either additions or removals from input collections, with their evolution tracked using logical time.

Alternatively, some systems may not support concurrent queries and updates. Instead, they alternate between incorporating batches of graph updates and graph queries, meaning updates are applied to the graph structure while queries wait, and vice versa. This type of architecture can achieve a high update processing rate because it doesn't need to resolve the problem of ensuring the consistency of graph queries running interleaved with updates concurrently.

Few frameworks have been implemented for continuous streaming models. DZiG[55] is a high-performance streaming graph processing system designed to maintain efficiency in scenarios with sparse computations while ensuring BSP semantics. DZiG's core components consist of an innovative incremental processing technique that is sparsity-aware, allowing computations to be expressed recursively to detect and eliminate updates, thus preserving sparsity safely. It also provides an adaptive processing model that automatically adjusts the incremental computation strategy to minimize overhead when computations become sparser.

GraphBolt[56] is a streaming graph processing system that ensures BSP (Bulk Synchronous Parallel) semantics while handling incremental updates. It employs a dependency-driven incremental processing approach, starting by tracing dependencies to understand how intermediate values are computed. It then uses this knowledge to propagate changes throughout the intermediate values incrementally. GraphBolt offers a versatile incremental programming model to accommodate various graph-based analytics tasks that facilitate the creation of incremental versions of intricate aggregations.

Other frameworks include faimGraph[57], which represents a fully dynamic graph data structure tailored for Graphics Processing Units (GPUs). It excels in delivering high update rates while maintaining a minimal memory footprint thanks to autonomous memory management directly within the GPU. This data structure is fully dynamic, accommodating updates for both edges and vertices. By conducting memory management on the GPU itself, it achieves swift initialization times and efficient update procedures without requiring additional intervention or reallocation from the host. Their optimized approach performs parallel initialization and achieves considerably faster speeds compared to previous methods. It can handle up to 200 million edge updates per second for both sorted and unsorted update batches. Furthermore, it can execute over 300 million adjacency queries and millions of vertex updates per second. Efficient memory management techniques, such as a queuing approach, ensure that currently unused memory can be repurposed by the framework, enabling the storage of tens of millions of vertices and hundreds of millions of edges in GPU memory. Preceding work from the same authors includes aimGraph[58] that is similar but excludes some of the sophisticated memory management tailored for GPU warp scheduling.

V. Frameworks for distributed GNN

Distributed GNN is a technique for training Graph Neural Networks (GNNs) on large graphs that are too big to fit on a single machine. It works by dividing the graph into smaller partitions, which are then trained on separate machines. The results of the individual training runs are then combined to produce a single model. Distributed GNN has several advantages over training GNNs on a single machine. First, it can be used to train GNNs on graphs that are too large to fit on a single machine. Second, it can speed up training time, as the computation can be parallelized across multiple machines. Third, it can improve the accuracy of the model, as the model can be trained on more data. The focus in this area has been on addressing the challenges with regards to **communication overhead, load imbalance,** and **model accuracy.**

PyG[59] and DGL[60] are the two most popular software frameworks in the GNN community. PyG[59] is a geometric deep learning extension library for PyTorch to enable deep learning on irregular structure data such as graphs. It supports both CPU and GPU computing, providing convenience for using GPU to accelerate the computing process. Through the message-passing application programming interface (API), it is easy to express various GNN models, as neighbor aggregation is a kind of message propagation.

DGL[60] is a framework specialized for deep learning models on graphs. It abstracts the computation of GNNs into a few user-configurable message-passing primitives, thus helping users express GNNs more conveniently. It achieves good performance by exploring a wide range of parallelization strategies. It also supports both CPU and GPU computing.

Other distributed approaches can mainly be categorized into Full-batch and mini-batch training.

A. Full-batch training

Full-batch training utilizes the whole graph to update model parameters in each round. In each epoch, the GNN model must aggregate representations of all neighboring vertices for every vertex at once. As a result, the model parameters are updated only once at each epoch. Multiple computing nodes need to synchronize gradients before updating model parameters so that the models across the computing nodes remain unified. Thus, a round of distributed full-batch training includes two phases: model computation (forward propagation + backward propagation) and gradient synchronization. The model parameter update is included in the gradient synchronization phase.

There have been many attempts at software frameworks that use a full-batch training approach. NeuGraph[61] is a distributed GNN training software framework proposed in 2019 using a multi-GPU hardware platform. It is categorized as the dispatch-workload-based execution of distributed full-batch training. It proposes SAGANN, an abstract model for the programming of GNN operations. SAGANN splits each layer of model computation into four stages, namely: Scatter, ApplyEdge, Gather, and ApplyVertex. The Scatter stage means the vertices scatter their features to their output edges. The Gather stage means the vertices gather the value from their input edges. There are two user-defined functions used in the ApplyEdge stage and ApplyVertex stage for users to declare neural network computations on edges and vertices respectively. Through the abstraction of SAGANN, users can easily express various GNN models and execute them in a parallelized way. NeuGraph optimizes the training process based on this abstract model, using techniques including vertex-centric workload partition, transmission planning, and feature-dimension workload partition.

Roc[62] is a distributed multi-GPU software framework for fast GNN training and inference, proposed in 2020. It is categorized as the dispatch-workload-based execution of distributed full-batch training. Its optimization of distributed training mainly focuses on balanced workload generation and transmission planning. In terms of balanced workload generation, an online linear regression

TABLE II
SUMMARY OF FEATURES FOR SELECT GNN FRAMEWORKS

Framework	Multi-CPU	Multi-GPU	Mini-batch	Full-batch	Homogeneous graph	Static graph	Dynamic graph
PyG[59]	✓	✓	✓	✓	✓	✓	✓
NeuGraph[61]	✗	✓	✗	✓	✓	✓	✗
Roc[62]	✗	✓	✗	✓	✓	✓	✗
FlexGraph[63]	✓	✗	✗	✓	✓	✓	✗
MG-GCN[64]	✗	✓	✗	✓	✓	✓	✗
Dorylus[65]	✓	✗	✗	✓	✓	✓	✗
Dorylus[65]	✓	✗	✗	✓	✓	✓	✗
AliGraph[66]	✓	✗	✓	✗	✓	✓	✓
AGL[67]	✓	✗	✓	✗	✓	✓	✗
GraphTheta[68]	✓	✗	✓	✓	✓	✓	✗
DGL[60]	✓	✓	✓	✓	✓	✓	✓

cost model is proposed to achieve efficient graph partition. The cost model is being tuned by collecting runtime data. According to this cost model, the training resources and time required for the subgraph can be estimated to guide the workload partition for workload balance. Regarding transmission planning, a recursive dynamic programming algorithm is introduced to find the optimal global solution to decide which part of the data should be cached in GPU memory for reuse.

FlexGraph[63] is a distributed multi-CPU training software framework proposed in 2021. It is categorized as the preset workload-based execution of distributed full-batch training. In order to express more kinds of GNNs including GNNs for heterogeneous graphs, it proposes a novel programming abstraction, namely NAU. NAU splits one GNN layer's computation into three stages: Neighbor Selection, Aggregation, and Update stages, each with a user-defined function. Based on NAU, FlexGraph optimizes the training process using graph pre-partition and partial aggregation techniques.

In terms of graph pre-partition, it introduces a cost model to estimate the runtime overhead of the workload so as to guide the workload partition for workload balance. In terms of partial aggregation, FlexGraph partially aggregates the features of vertices collocated at the same partition when possible. In addition, it overlaps partial aggregations and communication to reduce the transmission overhead.

MG-GCN[64] is a distributed multi-GPU training software framework proposed in 2021. It is categorized as the preset workload-based execution of distributed full-batch training. It focuses on efficiently parallelizing the sparse matrix-matrix multiplication (SpMM) kernel on a multi-GPU hardware platform. It uses a matrix partitioning method to distribute raw data to multiple GPUs, and each GPU is responsible for completing the workload of its own local matrix. It involves efficiently reusing memory buffers to reduce the memory footprint of training GNN models, and overlaps communication and computation to reduce communication overhead.

Specifically, the memory buffer in the computing node is used to cache the data reused by the forward propagation and backward propagation, thereby reducing data transmission. As for the communication and computation overlap, it uses two GPU streams for computation and communication, respectively.

Dorylus[65] is a distributed multi-CPU training software framework proposed in 2021. It is categorized as the preset workload-based execution of distributed full-batch training. Its main focus is on how to train GNNs at a low cost, so it adopts serverless computing. Serverless computing refers to "cloud function" threads, such as AWS Lambda and Google Cloud Functions, that can be used massively in parallel at an extremely low price. The hardware platform of Dorylus consists of CPUs and serverless threads. CPUs mainly perform the Aggregation operation, while the serverless threads are used for the Combination operation due to more regular computation and simpler workload partition in the Combination operations.

It adopts a fine-grained workload partition to adapt to the situation that the available hardware resources of serverless threads are quite limited. In addition, asynchronous training is used to make full use of computing resources and reduces stagnation.

B. *Mini-batch training*

AliGraph[66] is a distributed multi-CPU training framework proposed in 2019. It is categorized as the joint-sample-based execution of distributed mini-batch training. It supports not only GNNs for homogeneous graphs and static graphs but also GNNs for heterogeneous graphs and dynamic graphs.

In terms of storage, it adopts a graph partitioning method to store graph data in a distributed manner. The structure and features of the subgraph in each computing node are stored separately. In addition, two caches are added for the features of vertices and edges. Furthermore, it proposes a caching strategy to reduce the communication overhead between computing nodes; that is, each computing node caches the outgoing neighbors of frequently-used vertices.

AGL[67] is a distributed multi-CPU training software framework proposed in 2020. It is categorized as the individual-sample-based execution of distributed mini-batch training. To speed up the sampling process, it introduces a distributed pipeline to generate k-hop neighborhood in the spirit of message passing, which is implemented with MapReduce infrastructure[69]. In this way, in the sampling phase, mini-batch data can be rapidly generated by collecting the k-hop neighbors of the target vertices.

The computing nodes are partitioned into workers and parameter servers in the training phase. The workers perform the model computation on the mini-batches, while the parameter servers maintain the current version of the model parameters. It also uses the commonly used optimization techniques for better efficiency, such as the transmission pipeline.

There exists a software framework dedicated to both distributed full-batch training and distributed mini-batch training. GraphTheta[68] is a distributed multi-CPU training software framework proposed in 2021. It supports three training methods: mini-batch, full-batch, and cluster-batch training. The cluster-batch training method was proposed by Chiang[70] in 2019. It first partitions a large graph into a set of smaller clusters. Then, it generates a batch of data either based on one cluster or a combination of multiple clusters.

Apparently, cluster-batch restricts the neighbors of a target vertex to only one cluster, which is equivalent to conducting a globalized convolution on a cluster of vertices. There is a parameter server in GraphTheta, which is responsible for managing multi-version model parameters. The worker obtains the model parameters from the parameter server and transfers the generated gradient back to the parameter server for the update of model parameters. Multi-version parameter management makes it possible to train GNNs asynchronously as well as synchronously.

C. *Further optimizations*

Considering PyG[59] and DGL[60] are open source, a number of optimizations have been studied and implemented. DistGNN[71] proposes an approach that optimizes the well-known Deep Graph Library(DGL) for full-batch training on CPU clusters via an efficient shared memory implementation, communication reduction using a minimum vertex-cut graph partitioning algorithm, and communication avoidance using a family of delayed update algorithms. In the algorithm, the set of vertices that may be queried by other computing nodes is partitioned into r subsets. For each epoch computation, only the data of one subset is transmitted. The transmitted data is not required to be received at this epoch but after r epochs. This means that the computing nodes do not use the latest global data of vertices, but locally existing data of them. This algorithm allows communication to overlap with more computational processes, thereby reducing communication overhead.

SAR[72] draws on the idea of activation rematerialization and proposes sequential aggregation and rematerialization for distributed GNN training. The specific execution flow is as follows. In forward propagation, each computing node only receives activation from one other computing node at a time. After the aggregation operation is completed, the activation is removed immediately. Then, the computing node receives the activation from the next computing node and continues the aggregation operation. This makes the activation of each vertex only exist in the computing node where it is located, and there will be no replicas. In backward propagation, the computation is also performed sequentially as above. Each computing node transmits activation sequentially to complete the computation. Through

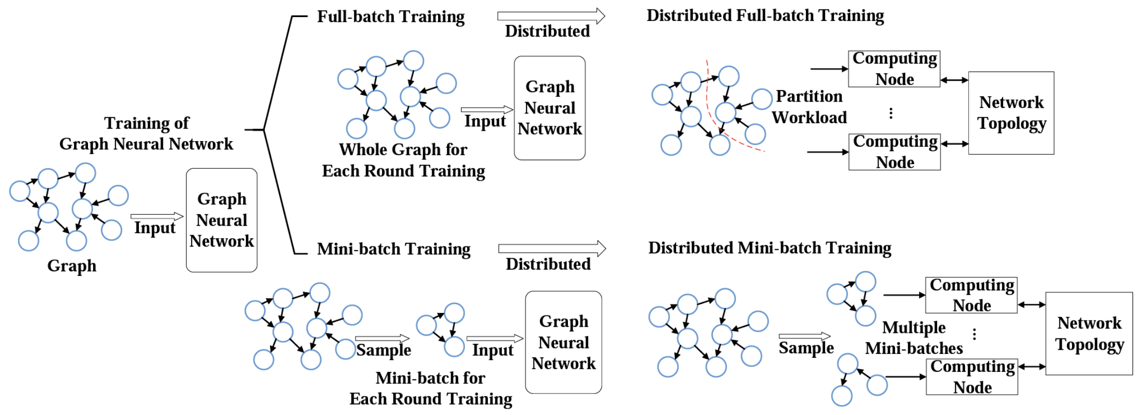

Fig. 3. Workflows of full-batch and mini-batch training

this method, memory will not overflow as long as the memory capacity of the computing node is larger than the size of two subgraphs. This allows SAR to scale to arbitrarily large graphs by simply adding more workers.

PaGraph[73] and DistDGL[74] use a Locality-aware partitioning to improve on DGL. This refers to partitioning the graph into subgraphs with good locality; that is, vertices and their neighbors have a high probability of being in the same subgraph, so most of the data required for sampling is local to the computing node. This partitioning is conducted in the preprocessing phase. Locality-aware partitioning aims to make the computation of the nodes more independent by reducing the communication between them. According to the workflow of mini-batch training, the model computation is restricted to using the minibatch data and does not involve access to the entire raw graph data. That means the access to the raw graph data is almost completely in the sampling phase. By focusing on the locality of the graph, each vertex and its neighbors are clustered into a subgraph as much as possible. This makes the workers mainly access their own subgraph during the sampling phase, reducing the remote queries of neighboring vertices and thus improving the independence of each worker's computation.

LLCG[75] proposes a method to improve the scalability of GNN training by making the computation of each computing node more independent. Each worker first performs sampling and model computation on its own subgraph without accessing the data of remote workers. This makes the computation of each worker more independent and reduces the communication overhead.

A parameter server periodically collects the model parameters from each worker and performs the average operation. The parameter server then refines the average result by sampling the mini-batch on the whole graph and conducting model computation. This refinement step ensures that the model parameters are accurate, even though each worker only has access to a local subgraph. Overall, LLCG[75] is a promising approach to scaling GNN training to large graphs. It achieves good scalability by making the computation of each worker more independent while still ensuring the accuracy of the model parameters through the refinement step.

Dynamic mini-batch allocation to compute nodes is another optimization technique that improves the scalability and performance of GNN. This is implemented by SALIENT[76] where the CPUs responsible for sampling and the GPUs responsible for model computation are not in a static correspondence. The number of each worker is sequentially stored in the queue. The minibatch generated by each CPU will be assigned a destination worker number according to the number stored in the queue during the generation. After the generation, the minibatch will be sent to the corresponding worker according to the destination number immediately. This effectively avoids the problem faced by static allocation, which is that the CPUs and GPUs may not be able to keep up with each other.

In this section, we explored the literature for training distributed GNNs by categorizing them as full-batch and mini-batch training models. The advantages of full-batch models like Dorylus[65], MG-GCN[64], FlexGraph[63] are that they are deterministic as they use the entire graph and its features for each forward and backward pass during training. On hardware optimized for matrix operations (e.g., GPUs), full-batch GNNs can also achieve high computational efficiency. However the disadvantages are that they are memory-intensive, add a computational overhead, and are slow to converge. Mini-batch training models like AGL[67] and AliGraph[66] on the other hand are memory efficient, scalable, and converge faster. However they are less deterministic and introduce stochasticity due to the random sampling of minibatches, which can result in noisier gradients and potentially slower convergence in some cases. In practice, the choice between full-batch and minibatch training depends on the size of your graph, the available hardware, and your specific application requirements. Full-batch GNNs are suitable for smaller graphs when computational resources are not a concern. Minibatch GNNs are more appropriate for larger graphs where memory and computational efficiency are critical, but you may need to carefully tune hyperparameters and potentially deal with increased noise in gradients.

Hybrid approaches, combining full-batch and minibatch training elements, are also used to strike a balance between efficiency and convergence speed, allowing GNNs to scale to even larger graphs while maintaining some level of determinism and control.

VI. Conclusion and discussion

In this survey, we studied several large-scale graph analytics research and frameworks in the literature by organizing them into four major categories. We first explored the landscape of different graph algorithms as vertex and edge-centric classifications while discussing the advantages of both approaches in Section II. We then categorized graph frameworks based on their target architecture in Section III. Single-node CPU and GPU frameworks offer the most performance for smaller graphs but struggle with scalability and memory utilization as the graphs increase in size. Multi-node frameworks offer great scalability for large-scale graphs but add excessive communication and synchronization overheads. In Section IV, we explored various dynamic graph frameworks and discussed the pros and cons of snapshot-based and continuous streaming models. Lastly in Section V, we discussed the various frameworks for distributed training of GNNs and the tradeoffs between full-batch and mini-batch training. Even with the extensive research in this area, plenty of uncovered topics and challenges could be addressed in future research.

Asynchronous multi-node algorithms : Multi-node graph algorithms have mainly concentrated on synchronous implementations due to the ease of programming and exhaustive ecosystem for BSP models. There exist a few asynchronous models for multi-node systems like [77],[78] and [17] amongst others but there is still plenty of scope for improvement on efficient memory and resource utilization.

GNN training on dynamic graphs : The majority of distributed training models for GNNs are geared towards static graphs. Dynamic models like [79],[80], and [81] exist but are mostly limited to single-node training.

Scalable GNN inferencing : Inferencing for large-scale graphs has a lot of potentially unexplored areas that could be refined. There are existing research like [82], [83] and [67] that could be further built upon to improve realtime forward-pass across multi-node GPUs.

Higher order representation for dynamic graphs : Higher-order graph representations are an extension of traditional graph representations that capture more complex relationships and patterns in data. While traditional graphs consist of nodes and edges, higher-order graphs introduce additional layers of structure by incorporating higher-order interactions between nodes, edges, or subgraphs. These representations are especially useful for modeling complex systems, such as social networks, knowledge graphs, and biological networks. Here are some key concepts and examples of higher-order graph representations. Limited related works in this area include [84], [85] and [86]. Applications of higher-order graph representations include community detection, link prediction, anomaly detection, recommendation systems, and the analysis of complex systems in fields like biology, social science, and network science. These representations provide a more nuanced understanding of relationships and patterns in data, allowing for more accurate and insightful analyses.

References

[1] N. A. Lynch, *Distributed algorithms*. Elsevier, 1996.

[2] Y. Low, J. E. Gonzalez, A. Kyrola, D. Bickson, C. E. Guestrin, and J. Hellerstein, "Graphlab: A new framework for parallel machine learning," *arXiv preprint arXiv:1408.2041*, 2014.

[3] L. Page, S. Brin, R. Motwani, and T. Winograd, "The pagerank citation ranking: Bring order to the web," Technical report, stanford University, Tech. Rep., 1998.

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

[5] C. Xie, R. Chen, H. Guan, B. Zang, and H. Chen, "Sync or async: Time to fuse for distributed graph-parallel computation," *ACM SIGPLAN Notices*, vol. 50, no. 8, pp. 194–204, 2015.

[6] Q. Chen, S. Bai, Z. Li, Z. Gou, B. Suo, and W. Pan, "Graphhp: A hybrid platform for iterative graph processing," *arXiv preprint arXiv:1706.07221*, 2017.

[7] S. Suri and S. Vassilvitskii, "Counting triangles and the curse of the last reducer," in *Proceedings of the 20th international conference on World wide web*, 2011, pp. 607–614.

[8] J. Gonzalez, Y. Low, A. Gretton, and C. Guestrin, "Parallel gibbs sampling: From colored fields to thin junction trees," in *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*. JMLR Workshop and Conference Proceedings, 2011, pp. 324–332.

[9] Y. Zhang, Q. Gao, L. Gao, and C. Wang, "Priter: A distributed framework for prioritized iterative computations," in *Proceedings of the 2nd ACM Symposium on Cloud Computing*, 2011, pp. 1–14.

[10] G. Wang, W. Xie, A. J. Demers, and J. Gehrke, "Asynchronous large-scale graph processing made easy." in *CIDR*, vol. 13, 2013, pp. 3–6.

[11] A. Roy, I. Mihailovic, and W. Zwaenepoel, "X-stream: Edge-centric graph processing using streaming partitions," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, 2013, pp. 472–488.

[12] P. Yuan, W. Zhang, C. Xie, H. Jin, L. Liu, and K. Lee, "Fast iterative graph computation: A path centric approach," in *SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2014, pp. 401–412.

[13] H. Zhu, L. He, M. Leeke, and R. Mao, "Wolfgraph: The edge-centric graph processing on gpu," *Future Generation Computer Systems*, vol. 111, pp. 552–569, 2020.

[14] A. Kyrola, G. Blelloch, and C. Guestrin, "{GraphChi}:{Large-Scale} graph computation on just a {PC}," in *10th USENIX symposium on operating systems design and implementation (OSDI 12)*, 2012, pp. 31–46.

[15] R. S. Xin, J. E. Gonzalez, M. J. Franklin, and I. Stoica, "Graphx: A resilient distributed graph system on spark," in *First international workshop on graph data management experiences and systems*, 2013, pp. 1–6.

[16] M. A. Hassaan, M. Burtscher, and K. Pingali, "Ordered vs. unordered: a comparison of parallelism and work-efficiency in irregular algorithms," *Acm Sigplan Notices*, vol. 46, no. 8, pp. 3–12, 2011.

[17] T. Ben-Nun, M. Sutton, S. Pai, and K. Pingali, "Groute: An asynchronous multi-gpu programming model for irregular computations," *ACM SIGPLAN Notices*, vol. 52, no. 8, pp. 235–248, 2017.

[18] L. Ma, Z. Yang, H. Chen, J. Xue, and Y. Dai, "Garaph: Efficient {GPU-accelerated} graph processing on a single machine with balanced replication," in *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, 2017, pp. 195–207.

[19] Z. Khayyat, K. Awara, A. Alonazi, H. Jamjoom, D. Williams, and P. Kalnis, "Mizan: a system for dynamic load balancing in large-scale graph processing," in *Proceedings of the 8th ACM European conference on computer systems*, 2013, pp. 169–182.

[20] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "{PowerGraph}: Distributed {Graph-Parallel} computation on natural graphs," in *10th USENIX symposium on operating systems design and implementation (OSDI 12)*, 2012, pp. 17–30.

[21] J. Zhong and B. He, "Medusa: Simplified graph processing on gpus," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 6, pp. 1543–1552, 2013.

[22] J. Shun and G. E. Blelloch, "Ligra: a lightweight graph processing framework for shared memory," in *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*, 2013, pp. 135–146.

[23] K. Pingali, D. Nguyen, M. Kulkarni, M. Burtscher, M. A. Hassaan, R. Kaleem, T.-H. Lee, A. Lenharth, R. Manevich, M. Méndez-Lojo *et al.*, "The tao of parallelism in algorithms," in *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, 2011, pp. 12–25.

[24] D. Nguyen, A. Lenharth, and K. Pingali, "A lightweight infrastructure for graph analytics," in *Proceedings of the twenty-fourth ACM symposium on operating systems principles*, 2013, pp. 456–471.

[25] K. Zhang, R. Chen, and H. Chen, "Numa-aware graph-structured analytics," in *Proceedings of the 20th ACM SIGPLAN symposium on principles and practice of parallel programming*, 2015, pp. 183–193.

[26] W. Han, D. Mawhirter, B. Wu, and M. Buland, "Graphie: Large-scale asynchronous graph traversals on just a gpu," in *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, 2017, pp. 233–245.

[27] C. Hong, A. Sukumaran-Rajam, J. Kim, and P. Sadayappan, "Multigraph: Efficient graph processing on gpus," in *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, 2017, pp. 27–40.

[28] S. Hong, S. K. Kim, T. Oguntebi, and K. Olukotun, "Accelerating cuda graph algorithms at maximum warp," *Acm Sigplan Notices*, vol. 46, no. 8, pp. 267–276, 2011.

[29] F. Khorasani, K. Vora, R. Gupta, and L. N. Bhuyan, "Cusha: vertex-centric graph processing on gpus," in *Proceedings of the 23rd international symposium on High-performance parallel and distributed computing*, 2014, pp. 239–252.

[30] Y. Wang, A. Davidson, Y. Pan, Y. Wu, A. Riffel, and J. D. Owens, "Gunrock: A high-performance graph processing library on the gpu," in *Proceedings of the 21st ACM SIGPLAN symposium on principles and practice of parallel programming*, 2016, pp. 1–12.

[31] S. Pai and K. Pingali, "A compiler for throughput optimization of graph algorithms on gpus," in *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2016, pp. 1–19.

[32] R. Nasre, M. Burtscher, and K. Pingali, "Morph algorithms on gpus," in *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*, 2013, pp. 147–156.

[33] ——, "Data-driven versus topology-driven irregular computations on gpus," in *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*. IEEE, 2013, pp. 463–474.

[34] ——, "Atomic-free irregular computations on gpus," in *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units*, 2013, pp. 96–107.

[35] E. Elsen and V. Vaidyanathan, "Vertexapi2–a vertex-program api for large graph computations on the gpu," 2014.

[36] D. Merrill, M. Garland, and A. Grimshaw, "Scalable gpu graph traversal," *ACM Sigplan Notices*, vol. 47, no. 8, pp. 117–128, 2012.

[37] Y. Pan, Y. Wang, Y. Wu, C. Yang, and J. D. Owens, "Multi-gpu graph analytics," in *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2017, pp. 479–490.

[38] U. Cheramangalath, R. Nasre, and Y. Srikant, "Falcon: A graph manipulation language for heterogeneous systems," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 12, no. 4, pp. 1–27, 2015.

[39] X. Zhu, W. Chen, W. Zheng, and X. Ma, "Gemini: A {Computation-Centric} distributed graph processing system," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, 2016, pp. 301–316.

[40] I. Hoque and I. Gupta, "Lfgraph: Simple and fast distributed graph analytics," in *Proceedings of the First ACM SIGOPS Conference on Timely Results in Operating Systems*, 2013, pp. 1–17.

[41] D. Foley and J. Danskin, "Ultra-performance pascal gpu and nvlink interconnect," *IEEE Micro*, vol. 37, no. 2, pp. 7–17, 2017.

[42] C. Martella, R. Shaposhnik, D. Logothetis, and S. Harenberg, *Practical graph analytics with apache giraph*. Springer, 2015, vol. 1.

[43] R. A. Rossi, B. Gallagher, J. Neville, and K. Henderson, "Modeling dynamic behavior in large evolving graphs," in *Proceedings of the sixth ACM international conference on Web search and data mining*, 2013, pp. 667–676.

[44] C. Ren, E. Lo, B. Kao, X. Zhu, and R. Cheng, "On querying historical evolving graph sequences," *Proceedings of the VLDB Endowment*, vol. 4, no. 11, pp. 726–737, 2011.

[45] Y. Yang, J. X. Yu, H. Gao, J. Pei, and J. Li, "Mining most frequently changing component in evolving graphs," *World Wide Web*, vol. 17, pp. 351–376, 2014.

[46] D. Ediger, R. McColl, J. Riedy, and D. A. Bader, "Stinger: High performance data structure for streaming graphs," in *2012 IEEE Conference on High Performance Extreme Computing*. IEEE, 2012, pp. 1–5.

[47] G. Feng, X. Meng, and K. Ammar, "Distinger: A distributed graph data structure for massive dynamic graph processing," in *2015 IEEE International Conference on Big Data (Big Data)*. IEEE, 2015, pp. 1814–1822.

[48] O. Green and D. A. Bader, "custinger: Supporting dynamic graph algorithms for gpus," in *2016 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2016, pp. 1–6.

[49] J. Mondal and A. Deshpande, "Managing large dynamic graphs efficiently," in *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, 2012, pp. 145–156.

[50] J. C. Anderson, J. Lehnardt, and N. Slater, *CouchDB: the definitive guide: time to relax*. " O'Reilly Media, Inc.", 2010.

[51] P. Macko, V. J. Marathe, D. W. Margo, and M. I. Seltzer, "Llama: Efficient graph analytics using large multiversioned arrays," in *2015 IEEE 31st International Conference on Data Engineering*. IEEE, 2015, pp. 363–374.

[52] L. Dhulipala, G. E. Blelloch, and J. Shun, "Low-latency graph streaming using compressed purely-functional trees," in *Proceedings of the 40th ACM SIGPLAN conference on programming language design and implementation*, 2019, pp. 918–934.

[53] A. P. Iyer, Q. Pu, K. Patel, J. E. Gonzalez, and I. Stoica, "{TEGRA}: Efficient {Ad-Hoc} analytics on evolving graphs," in *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, 2021, pp. 337–355.

[54] F. D. McSherry, R. Isaacs, M. A. Isard, and D. G. Murray, "Differential dataflow," Oct. 20 2015, uS Patent 9,165,035.

[55] M. Mariappan, J. Che, and K. Vora, "Dzig: Sparsity-aware incremental processing of streaming graphs," in *Proceedings of the Sixteenth European Conference on Computer Systems*, 2021, pp. 83–98.

[56] M. Mariappan and K. Vora, "Graphbolt: Dependency-driven synchronous processing of streaming graphs," in *Proceedings of the Fourteenth EuroSys Conference 2019*, 2019, pp. 1–16.

[57] M. Winter, D. Mlakar, R. Zayer, H.-P. Seidel, and M. Steinberger, "faimgraph: high performance management of fully-dynamic graphs under tight memory constraints on the gpu," in *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2018, pp. 754–766.

[58] M. Winter, R. Zayer, and M. Steinberger, "Autonomous, independent management of dynamic graphs on gpus," in *2017 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2017, pp. 1–7.

[59] M. Fey and J. E. Lenssen, "Fast graph representation learning with pytorch geometric," *arXiv preprint arXiv:1903.02428*, 2019.

[60] M. Y. Wang, "Deep graph library: Towards efficient and scalable deep learning on graphs," in *ICLR workshop on representation learning on graphs and manifolds*, 2019.

[61] L. Ma, Z. Yang, Y. Miao, J. Xue, M. Wu, L. Zhou, and Y. Dai, "{NeuGraph}: Parallel deep neural network computation on large graphs," in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, 2019, pp. 443–458.

[62] Z. Jia, S. Lin, M. Gao, M. Zaharia, and A. Aiken, "Improving the accuracy, scalability, and performance of graph neural networks with roc," *Proceedings of Machine Learning and Systems*, vol. 2, pp. 187–198, 2020.

[63] L. Wang, Q. Yin, C. Tian, J. Yang, R. Chen, W. Yu, Z. Yao, and J. Zhou, "Flexgraph: a flexible and efficient distributed framework for gnn training," in *Proceedings of the Sixteenth European Conference on Computer Systems*, 2021, pp. 67–82.

[64] M. F. Balin, K. Sancak, and U. V. Catalyurek, "Mg-gcn: A scalable multi-gpu gcn training framework," in *Proceedings of the 51st International Conference on Parallel Processing*, 2022, pp. 1–11.

[65] J. Thorpe, Y. Qiao, J. Eyolfson, S. Teng, G. Hu, Z. Jia, J. Wei, K. Vora, R. Netravali, M. Kim *et al.*, "Dorylus: Affordable, scalable, and accurate {GNN} training with distributed {CPU} servers and serverless threads," in *15th USENIX Symposium*

on Operating Systems Design and Implementation (OSDI 21), 2021, pp. 495–514.

[66] R. Zhu, K. Zhao, H. Yang, W. Lin, C. Zhou, B. Ai, Y. Li, and J. Zhou, "Aligraph: A comprehensive graph neural network platform," *arXiv preprint arXiv:1902.08730*, 2019.

[67] D. Zhang, X. Huang, Z. Liu, Z. Hu, X. Song, Z. Ge, Z. Zhang, L. Wang, J. Zhou, Y. Shuang *et al.*, "Agl: a scalable system for industrial-purpose graph machine learning," *arXiv preprint arXiv:2003.02454*, 2020.

[68] Y. Liu, H. Li, G. Zhang, X. Zeng, Y. Li, B. Huang, P. Zhang, Z. Li, X. Zhu, C. He *et al.*, "Graphtheta: A distributed graph neural network learning system with flexible training strategy," *arXiv preprint arXiv:2104.10569*, 2021.

[69] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[70] W.-L. Chiang, X. Liu, S. Si, Y. Li, S. Bengio, and C.-J. Hsieh, "Cluster-gcn: An efficient algorithm for training deep and large graph convolutional networks," in *Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data mining*, 2019, pp. 257–266.

[71] V. Md, S. Misra, G. Ma, R. Mohanty, E. Georganas, A. Heinecke, D. Kalamkar, N. K. Ahmed, and S. Avancha, "Distgnn: Scalable distributed training for large-scale graph neural networks," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2021, pp. 1–14.

[72] H. Mostafa, "Sequential aggregation and rematerialization: Distributed full-batch training of graph neural networks on large graphs," *Proceedings of Machine Learning and Systems*, vol. 4, pp. 265–275, 2022.

[73] Z. Lin, C. Li, Y. Miao, Y. Liu, and Y. Xu, "Pagraph: Scaling gnn training on large graphs via computation-aware caching," in *Proceedings of the 11th ACM Symposium on Cloud Computing*, 2020, pp. 401–415.

[74] C. M. Da Zheng, M. Wang, and J. Zhou, "Qidong su, xiang song, quan gan, zheng zhang, and george karypis. distdgl: distributed graph neural network training for billion-scale graphs. in 2020 ieee/acm 10th workshop on irregular applications: Architectures and algorithms (ia3)," 2020.

[75] M. Ramezani, W. Cong, M. Mahdavi, M. T. Kandemir, and A. Sivasubramaniam, "Learn locally, correct globally: A distributed algorithm for training graph neural networks," *arXiv preprint arXiv:2111.08202*, 2021.

[76] T. Kaler, N. Stathas, A. Ouyang, A.-S. Iliopoulos, T. Schardl, C. E. Leiserson, and J. Chen, "Accelerating training and inference of graph neural networks with fast sampling and pipelining," *Proceedings of Machine Learning and Systems*, vol. 4, pp. 172–189, 2022.

[77] R. Pearce, "Highly asynchronous visitor queue graph toolkit," Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States), Tech. Rep., 2012.

[78] B. Priest, T. Steil, G. Sanders, and R. Pearce, "You've got mail (ygm): Building missing asynchronous communication primitives," in *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 2019, pp. 221–230.

[79] H. Li and L. Chen, "Cache-based gnn system for dynamic graphs," in *Proceedings of the 30th ACM International Conference on Information & Knowledge Management*, 2021, pp. 937–946.

[80] M. Guan, A. P. Iyer, and T. Kim, "Dynagraph: dynamic graph neural networks at scale," in *Proceedings of the 5th ACM SIGMOD Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA)*, 2022, pp. 1–10.

[81] A. Pareja, G. Domeniconi, J. Chen, T. Ma, T. Suzumura, H. Kanezashi, T. Kaler, T. Schardl, and C. Leiserson, "Evolvegcn: Evolving graph convolutional networks for dynamic graphs," in *Proceedings of the AAAI conference on artificial intelligence*, vol. 34, no. 04, 2020, pp. 5363–5370.

[82] L. Zeng, P. Huang, K. Luo, X. Zhang, Z. Zhou, and X. Chen, "Fograph: Enabling real-time deep graph inference with fog computing," in *Proceedings of the ACM Web Conference 2022*, 2022, pp. 1774–1784.

[83] Z. Wang, Y. Wang, C. Yuan, R. Gu, and Y. Huang, "Empirical analysis of performance bottlenecks in graph neural network training and inference with gpus," *Neurocomputing*, vol. 446, pp. 165–191, 2021.

[84] J. A. Dykes, "Exploring spatial data representation with dynamic graphics," *Computers & Geosciences*, vol. 23, no. 4, pp. 345–370, 1997.

[85] J. Zhu, B. Li, Z. Zhang, L. Zhao, and H. Li, "High-order topology-enhanced graph convolutional networks for dynamic graphs," *Symmetry*, vol. 14, no. 10, p. 2218, 2022.

[86] V. Perri and I. Scholtes, "Hotvis: Higher-order time-aware visualisation of dynamic graphs," in *Graph Drawing and Network Visualization: 28th International Symposium, GD 2020, Vancouver, BC, Canada, September 16–18, 2020, Revised Selected Papers 28*. Springer, 2020, pp. 99–114.