Scalable Graph Analytics: A Study On Algorithms and Infrastructure

by

Sudharshan Srinivasan

A dissertation accepted and approved in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

in Computer Science

Dissertation Committee:

Allen Malony, Chair
Boyana Norris, Core Member
Jeewan Choi, Core Member
Marina Guenza, Institutional Representative

University of Oregon

Spring 2025

DISSERTATION ABSTRACT

Sudharshan Srinivasan

Doctor of Philosophy in Computer Science

Title: Scalable Graph Analytics: A Study On Algorithms and Infrastructure

Graph analytics has become an essential tool for extracting insights from complex, interconnected data across diverse domains such as social networks, biological systems, and natural language processing. However, as data volume and complexity continue to grow, traditional graph processing techniques face significant scalability challenges, limiting their efficiency and effectiveness. Concurrently, advancements in high-performance computing (HPC) and machine learning (ML) offer promising solutions to address these limitations by enhancing computational efficiency and analytical depth.

This dissertation investigates the fundamental challenges in scaling large-scale graph analytics and explores how the integration of HPC and ML can lead to more efficient, scalable, and adaptive analytical frameworks. Specifically, we examine how HPC infrastructure can improve the performance of graph processing algorithms, how ML models can be leveraged to address inherent limitations in traditional graph analytics, and how advancements in graph analytics can, in turn, refine and enhance machine learning techniques. By bridging the gap between these fields, this research aims to contribute to the development of next-generation, high-performance graph analytics frameworks capable of handling dynamic, large-scale datasets with greater efficiency and adaptability.

This dissertation includes previously published and unpublished coauthored material.

TABLE OF CONTENTS

Chapter Page

LIST OF FIGURES

LIST OF TABLES

LIST OF ALGORITHMS

Algorithm Page

CHAPTER I

INTRODUCTION

In recent years, graph analytics has emerged as a critical tool for analyzing complex, interconnected data across various domains, from social networks and biological systems to natural language processing. As the volume and complexity of data continue to grow, traditional graph processing methods often encounter significant scalability challenges, hindering the extraction of actionable insights. Concurrently, advancements in high-performance computing (HPC) and machine learning offer promising avenues to overcome these limitations by enhancing efficiency and analytical depth. This dissertation investigates the current challenges in scaling large-scale graph analytics and explores how the integration of HPC and machine learning models can address these issues, paving the way for more robust and scalable analytical frameworks.

1.1 Research Question and Objectives

Overarching Research Question: What are the current challenges of large-scale graph analytics, and how can advancements in high-performance computing and machine learning models address these challenges?

Key Objectives:

1. How can high-performance computing (HPC) infrastructure enhance the efficiency and scalability of large-scale graph analytics?

2. How can emerging machine learning models contribute to overcoming the limitations inherent in current graph analytics approaches?

3. In what ways can improvements in high-performance graph analytics, driven by ML, feedback to refine and advance machine learning models?

Section 1.2 introduces key preliminaries relevant to this dissertation. Section 1.3 outlines strategies to address the first research objective, focusing on the role of high-performance computing (HPC) in enabling efficient and scalable graph analytics. Sections 1.4, 1.6, and 1.5 explore approaches for the second and third research objectives, examining the bidirectional relationship between machine learning models and graph analytics. The remaining chapters will explore each of these sections in greater detail.

1.2 Background

Large-scale graph analytics has become a critical research area due to the increasing demand for analyzing complex, interconnected data in fields such as social networks, biology, cybersecurity, and transportation. Graphs provide a natural representation of relationships between entities, but their size and complexity introduce significant computational challenges. Traditional graph processing techniques often struggle to scale with the exponential growth of data, necessitating advancements in computational frameworks, algorithms, and infrastructure. This section provides background on key technologies, including high-performance computing (HPC), graph analytics, and machine learning models.

1.2.1 Multi-core and Multi-node Systems.

Multi-core and multi-node systems are essential in parallel computing, designed to enhance computational performance by leveraging multiple processing units.

A **multi-core system** consists of a single processor chip with multiple cores, each capable of executing independent instructions concurrently. This architecture allows for efficient parallelism within a single machine.

A **multi-node system**, also known as a distributed computing cluster, comprises multiple interconnected computers (nodes), each with its own memory,

processor, and operating system. Parallelism is achieved by distributing workloads across nodes, enabling scalability for large-scale computations.

While multi-node systems provide **scalability** and **memory optimization** [113], they introduce challenges such as Communication Overhead, where data exchange between nodes incurs network latency and bandwidth constraints. They also introduce load-balancing challenges where ensuring an even distribution of workloads across nodes is critical for efficiency.

1.2.2 Dynamic Graphs. Dynamic graphs evolve over time as vertices and edges are added or removed, making them useful for modeling real-world systems such as social networks, transportation networks, and financial markets [8].

Processing dynamic graphs presents several challenges. Keeping track of continuous updates and frequent topology changes can be computationally intensive. This poses scalability constraints where large and complex graphs require optimized processing frameworks.

Several dynamic graph processing frameworks exist, including **Pregel** [85], **GraphLab** [77], and **Apache Giraph** [79], each optimized for different use cases. Framework selection depends on factors such as graph size, update frequency, and performance requirements.

Dynamic graphs are typically modeled using **discrete** and **continuous** approaches. In **discrete models**, graph snapshots are captured at fixed time intervals, such as every 30 minutes or daily, providing a structured representation of changes. In contrast, **continuous models** track all changes in real-time, ensuring a fully accurate and time-sensitive representation of the graph's evolution.

Co-author Acknowledgement Chapter II contains both unpublished and published material with and without co-authorship. Specific contributions are

detailed at the beginning of the chapter, but co-authors include Dr. Boyana Norris and Dr. Allen Malony.

1.3 Algorithms for Large-Scale Dynamic Graphs

This section explores algorithmic solutions for large-scale graph analytics, focusing on scaling incremental algorithms for identifying **Strongly Connected Components (SCCs).**

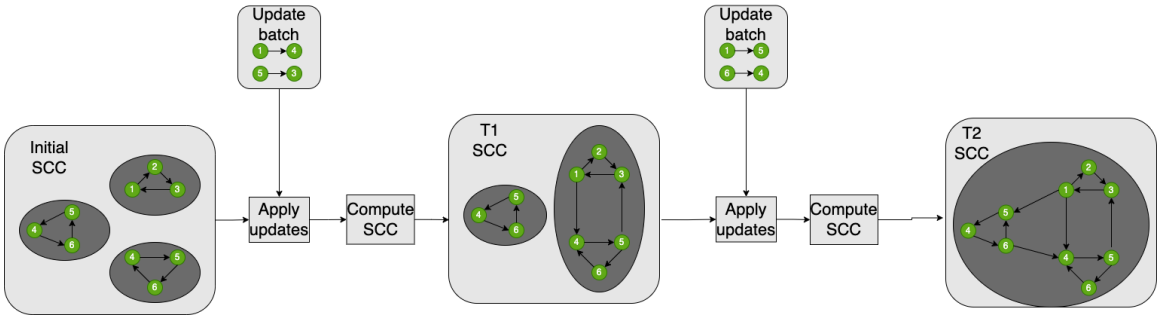


Figure 1. Example workflow for computing SCC on incremental graphs. At each timestep T, a new batch of updates is added, and SCC is recomputed.

An SCC in a directed graph is a subset of vertices where each vertex can reach every other vertex in the subset via directed paths in both directions. Efficiently identifying SCCs in large-scale graphs is crucial for applications such as network analysis and recommendation systems. Figure 4 outlines the workflow for computing SCC on incremental graphs.

To address scalability challenges, this dissertation proposes a **distributed SCC algorithm** for incremental graphs. This two-phase asynchronous approach stores intermediate results between iterations using a novel **meta-graph storage format**, reducing recomputation overhead for successive updates [135].

To the best of our knowledge, this is the first attempt at identifying SCCs in incremental graphs across distributed computing nodes. Experimental evaluations on real and synthetic datasets demonstrate up to **2.8× performance**

improvement over state-of-the-art methods by optimizing memory usage and communication bandwidth.

Co-author Acknowledgement Chapter III contains both unpublished and published material with and without co-authorship. Specific contributions are detailed at the beginning of the chapter, but co-authors include Dr. Boyana Norris, Dr. Arindam Khanda, Dr. Sriram Srinivasan, Dr. Sanjuktha Bhowmick, Dr. Sejal Das, and Aashish Pandey.

1.4 Model-based selection of graph frameworks

Over the years, significant progress has been made in developing software toolkits designed to address complex scientific problems. A diverse range of tools has emerged, facilitating the efficient resolution of extensive mathematical computations. One of the primary challenges in this domain is selecting the appropriate algorithm. An incorrect choice can lead to substantial performance degradation, particularly when iterated over multiple experiments within an application. This complexity is further compounded by the fact that researchers, while experts in their respective applications, may lack the detailed knowledge required to discern subtle experimental setup variations that could lead to significant performance improvements.

To address this selection challenge, we propose a model-based approach that leverages machine-learning techniques to optimize the selection of parallel graph processing packages. Our framework assists in package selection using classical machine learning algorithms, with the goal of either predicting execution time or classifying packages based on their suitability for a given graph and hardware configuration.

The fundamental motivation behind our selection approach is that the time required for feature extraction and model training is significantly lower than the time needed to execute the experiments directly. By leveraging machine learning for package selection, we aim to enhance efficiency in scientific computing workflows, enabling researchers to make informed decisions with minimal computational overhead.

Co-author Acknowledgement Chapter IV contains both unpublished and published material with and without co-authorship. Specific contributions are detailed at the beginning of the chapter, but co-authors include Dr. Boyana Norris and Dr. Sam Pollard

1.5 Learnable Hierarchies for Graph-based Multi-Agent Reinforcement Learning

Graph-based Multi-Agent Reinforcement Learning (MARL) systems have shown significant promise in solving complex decision-making tasks involving multiple agents. However, as the number of agents increases and their interactions grow more intricate, designing effective value function estimators that capture both local and global coordination remains challenging. This paper introduces a novel hierarchical critic approach that leverages tiered Graph Neural Networks (GNNs) to enhance the performance of graph-based MARL. The proposed method leverages hierarchical GNNs to accommodate the diverse policy requirements of different agents operating in dynamic environments. A key innovation is a learned allocation mechanism that assigns each agent to its most beneficial tier, maximizing the overall joint reward. By dynamically aggregating multi-tier information, the hierarchical critic captures both fine-grained and coarse-grained dependencies, to enable a scalable and generalizable approach to policy learning. Experimental

evaluations on the Multi-Agent Particle Environment (MPE) and Google Research Football (GRF) demonstrate that this tiered GNN-based hierarchical critic outperforms conventional flat GNN-based methods and other state-of-the-art MARL algorithms, achieving faster convergence, improved coordination, and stronger generalization to unseen scenarios. This work highlights the potential of hierarchical critics for advancing graph-based MARL environments with specialized agent roles.

1.6 Asynchronous GNN communications on multi-node systems

Graph Neural Networks (GNNs) have been widely researched in recent years to solve complex applications that require understanding interactions between different entities. The state-of-the-art model architectures like GraphSAGE [41] and GAT [145] implement a synchronous paradigm but still face considerable challenges, such as under-reaching and over-smoothing, when attempting to communicate over long distances in multi-node data-parallel environments. Asynchronous paradigms, on the other hand, are less explored. This chapter introduces a novel asynchronous paradigm for performing forward propagation of GNNs on multi-node systems. Our implementations use YGM [19] as a back-end communicator, and we evaluate our model on identifying the shortest distance parity problem.

The topics discussed in this chapter provide a foundational understanding of the challenges and advancements in large-scale graph analytics, particularly in the context of high-performance computing (HPC) and machine learning. Sections 1.2, 1.3, 1.4, 1.5 and, 1.6 introduced key concepts related to graph analytics, strongly connected components, model-based graph selection, asynchronous GNN communications, and hierarchical GNN architectures. The following chapters

explore these topics in greater depth: Chapter II elaborates on the theoretical background and computational frameworks essential for large-scale graph analytics. Chapter III presents an optimized algorithm for incremental SCC detection and its distributed implementations. Chapter IV presents a model-based selection of graph frameworks and their applications. Chapter V examines hierarchical GNN architectures and their effectiveness in multi-agent reinforcement learning environments. Finally, Chapter VI proposes an asynchronous GNN communication paradigm and its implications for multi-node systems. Together, these chapters form a cohesive exploration of innovative strategies for addressing the scalability and computational challenges inherent in large-scale graph processing.

CHAPTER II

BACKGROUND

Chapter II includes material that has been developed in collaboration with several co-authors as part of a broader research effort. The chapter incorporates content from papers published at SBAC-PAD 2023 and UO Area Exam 2024, which presents joint work on scalable graph analytics. Dr. Boyana Norris and Dr. Allen Malony contributed to the conceptualization and supervision of the core methodology.

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 chapter, 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 chapter 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.

2.1 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

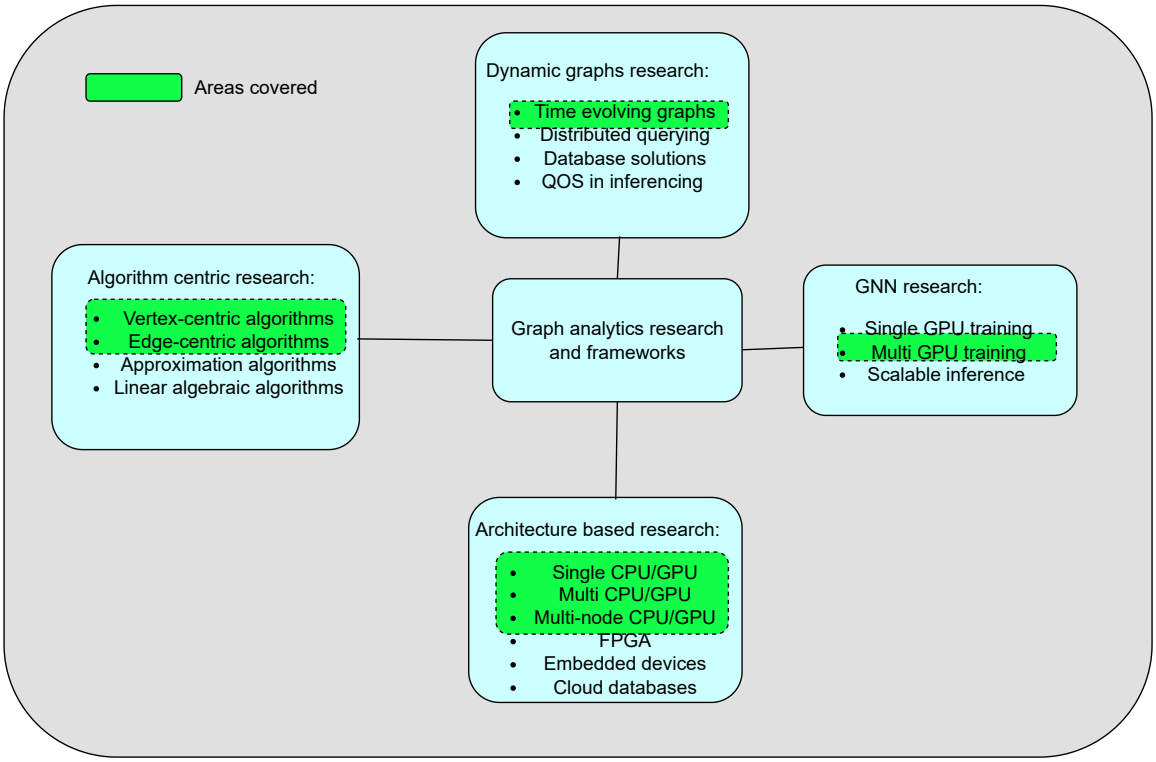


Figure 2. Topics for graph analytics frameworks and research in literature. The scope of this survey extends to the highlighted areas.

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 2.2, we introduce the landscape of graph algorithms and the categorization of edge and vertex-centric algorithms. In Section 2.3, we discuss the various available frameworks classified with respect to the target architecture. In Section 2.4, we explore the concepts of dynamic graphs and the available frameworks for dynamic graph analytics. In Section 2.5, we explore distributed training frameworks for GNNs and GATs. We finally conclude on the current state and possible future directions of the field.

2.2 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.

2.2.1 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 [81]. Several distributed algorithm implementations, such as distributed Bellman-Ford single-source shortest path [81], 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 [77].

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 [107] as well as a distributed, vertex-centric form [85]. 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 [85]. 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[157].

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 [15], 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 [140]. 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 [35] 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 [157].

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 [77], 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 [167]. 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 [147] [77].

2.2.2 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 [122] 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[162] 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[170] updates on works from X-stream[122] 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 ConcatenatedEdgeList (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[67] 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[67] 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.

2.3 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.

Table 1. Summary of features for select graph frameworks across architectures

Framework	CPU-GPU hybrid	Distributed	Asynchronous	BSP	Rich API	Directed graph
GraphX[158]	✓	✓	✗	✓	✗	✓
Pregel[85]	✓	✓	✗	✓	✓	✗
Galois[43]	✗	✗	✗	✓	✓	✗
Groute[7]	✗	✓	✓	✗	✗	✗
Garaph[82]	✗	✓	✗	✓	✓	✗
Mizan[60]	✓	✓	✗	✗	✗	✓
PowerGraph [36]	✓	✓	✗	✓	✓	✗
Medusa[168]	✓	✗	✗	✓	✗	✗
Ligra[125]	✗	✗	✗	✓	✗	✗

Several frameworks exist for graph analytics on various target architectures.

2.3.1 Shared-memory CPU and single-GPU systems. Galois[43, 111, 105] 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 [125], and Polymer [164] 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 [42] 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 [45, 46] 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 [61] 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 [153] 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 exists wrapper compilers like IrGL [108] that produce CUDA code from an intermediate-level program representation. Several algorithms [103, 102, 101] have shown that GPUs can be efficiently utilized for irregular computations.

2.3.2 Single-node multi GPUs and heterogeneous systems.

Several frameworks and libraries exist for graph processing on multiple GPUs. Groute [7] provides constructs for asynchronous multi-GPU programming and describes their implementation in a thin runtime environment. Medusa [168] 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 [27, 95, 109] provide libraries that allow programmers to easily extend single-GPU graph algorithms to achieve scalable performance on large graphs with billions of edges.

Several multi-GPU frameworks are also available that combine with CPUs for heterogeneous computations. Garaph [82] 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 [17] 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[7], 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.

2.3.3 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 [173] 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. LFGraph [48] offers low and balanced communication and computation, low preprocessing overhead, low memory footprint, and scalability for distributed graph analytics.

Pregel [85] approaches large-scale graph processing by expressing programs as sequences of iterations for computations and communication phases. Mizan [60] additionally extends the pregel framework by utilizing efficient load balancing techniques depending on the workload characteristics. GraphX [158] 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 [36] 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 [61], Ligra[125], Gunrock[153] and Galois[43, 111, 105] 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[82] and groute[7] offer improved memory utilization but still have a bottleneck when scaling to very large graphs as they still rely on rapid interconnects like NVlink[34]. Lastly, multi-node frameworks like Pregel [85], Mizan [60] and GraphX[158] 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.

2.4 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[85], GraphLab[77], and Apache Giraph[88]. 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.

2.4.1 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 [121], FVF [118], and Yang's model [161]. 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 [161] and Rossi's model [121], would clearly require a prohibitively large storage. FVF [118] 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 [118] 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 [118] is more efficient with regard to the used storage than the naive model described by Yang [121] and Rossi [121]. 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.

2.4.1.1 CSR representation. STINGER[25] 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[30]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 delegates 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[38] 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.[96] 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 [3], an in-memory key-value store. However, specific details about how the data is represented are not provided.

LLAMA [84], 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.

2.4.1.2 Tree representation.

The Aspen framework, as described in reference [22], 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 [51] 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.

2.4.2 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[92], 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[86] 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[87] 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[155], 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[156] that is similar but excludes some of the sophisticated memory management tailored for GPU warp scheduling.

2.5 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[31] and DGL[151] are the two most popular software frameworks in the GNN community. PyG[31] 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[151] 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.

2.5.1 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.

Table 2. Summary of features for select GNN frameworks

Framework	Multi-CPU	Multi-GPU	Mini-batch	Full-batch	Homogeneous graph	Static graph	Dynamic graph
PyG[31]	✓	✓	✓	✓	✓	✓	✓
NeuGraph[83]	✗	✓	✗	✓	✓	✓	✗
Roc[53]	✗	✓	✗	✓	✓	✓	✗
FlexGraph[150]	✓	✗	✗	✓	✓	✓	✗
MG-GCN[4]	✗	✓	✗	✓	✓	✓	✗
Dorylus[144]	✓	✗	✗	✓	✓	✓	✗
Dorylus[144]	✓	✗	✗	✓	✓	✓	✗
AliGraph[172]	✓	✗	✓	✗	✓	✓	✓
AGL[163]	✓	✗	✓	✗	✓	✓	✗
GraphTheta[75]	✓	✗	✓	✓	✓	✓	✗
DGL[151]	✓	✓	✓	✓	✓	✓	✓

There have been many attempts at software frameworks that use a full-batch training approach. NeuGraph[83] 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[53] 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 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[150] 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[4] 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[144] is a distributed multi-CPU training software framework proposed in 2021. It is categorized as the preset workload-based execution of

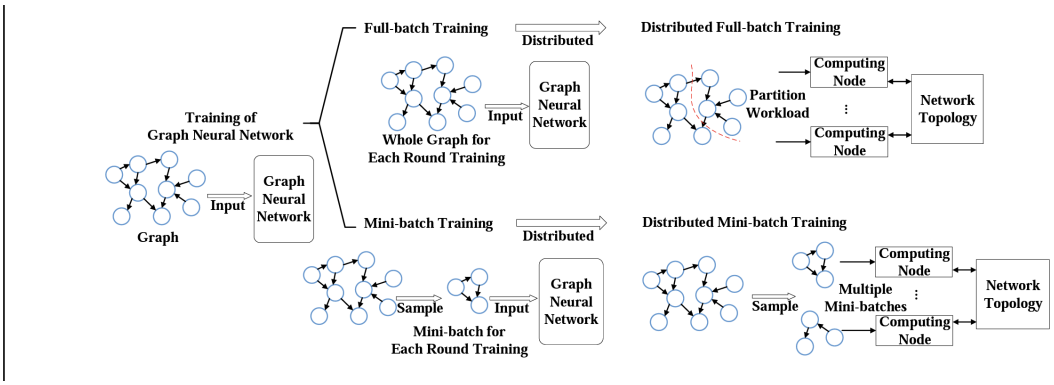


Figure 3. Workflows of full-batch and mini-batch training

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.

2.5.2 Mini-batch training. AliGraph[172] 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[163] 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[21]. 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[75] 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[18] 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.

2.5.3 Further optimizations. Considering PyG[31] and DGL[151] are open source, a number of optimizations have been studied and implemented. DistGNN[93] 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[97] 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 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[74] and DistDGL[20] 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[114] 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[114] 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[55] 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 immediately according to the destination number. 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[144], MG-GCN[4], FlexGraph[150] 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[163] and AliGraph[172] 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.

CHAPTER III

ADAPTIVE ALGORITHMS FOR INCREMENTAL GRAPHS

Chapter III includes material that has been developed in collaboration with several co-authors as part of a broader research effort. The chapter incorporates content from a paper published at SBAC-PAD 2023, which presents joint work on scalable graph analytics. Dr. Boyana Norris and Dr. Sanjuktha Bhowmick contributed to the conceptualization and supervision of the core methodology. Dr. Arindam Khanda and Dr. Sejal Das were involved in the design and performance evaluation of the parallel algorithms. Dr. Sriram Srinivasan provided insights into the systems-level implementation and optimization aspects. Aashish Pandey assisted with the experimental validation and ensured the reproducibility of the results.

Incremental graphs that change over time capture the changing relationships of different entities. Given that many real-world networks are extremely large, it is often necessary to partition the network over many distributed systems and solve a complex graph problem over the partitioned network. This chapter presents a distributed algorithm for identifying strongly connected components (SCC) on incremental graphs. We propose a two-phase asynchronous algorithm that involves storing the intermediate results between each iteration of dynamic updates in a novel meta-graph storage format for efficient recomputation of the SCC for successive iterations. To the best of our knowledge, this is the first attempt at identifying SCC for incremental graphs across distributed compute nodes. Our experimental analysis on real and synthesized graphs shows up to 2.8x performance improvement over the state-of-the-art by reducing the overall memory utilized and improving the communication bandwidth.

Dynamic graphs, Distributed systems, Strongly connected components.

3.1 Introduction

Detecting Strongly Connected Components (SCCs) in a large directed graph is a fundamental graph analytics problem. An SCC is defined as a subset of vertices in a directed graph with a path from any vertex to every other vertex in that subset. A graph can have many SCCs, but vertices are mutually exclusive to these SCCs. Detecting SCCs has many applications, such as pattern matching [165], topological sort [2], and graph analytics [128]. Although detecting SCCs by Depth First Search (DFS) on a directed graph works well for a sequential approach, performing a DFS can be expensive and computationally challenging in a parallel architecture [117].

Incremental graphs are extended graph data structures that undergo continuous updates, such as the addition of nodes and edges, which present numerous additional challenges. One of the foremost concerns is maintaining performance efficiency, as updates to the graph can impact the functionality of graph-based algorithms, potentially necessitating full recomputation. Another challenge lies in ensuring data consistency post-updates, as changes to a single node or edge could impact the overall graph or its segments. Memory management also poses a significant issue, particularly with large-scale graphs that rapidly consume memory resources, requiring efficient storage and retrieval methods such as graph partitioning or compression. Traditional graph algorithms designed for static graphs may not be practical for these dynamic, ever-evolving structures, hence calling for the development of dynamic algorithms. Query processing in such a fluid environment becomes complex, as results must be recalculated after every modification. If updates to the graph are performed by different processes

or threads concurrently, managing these simultaneous alterations to maintain data integrity and consistency becomes a substantial challenge, often needing locking or transaction management mechanisms. Lastly, like all data types, incremental graphs have concerns regarding data privacy and security. These issues become even more pressing in distributed environments where it is imperative to ensure that updates are authorized, and information remains uncompromised. Overcoming these challenges necessitates a blend of advanced algorithms, effective data management practices, and adept software engineering techniques, as explained in [9].

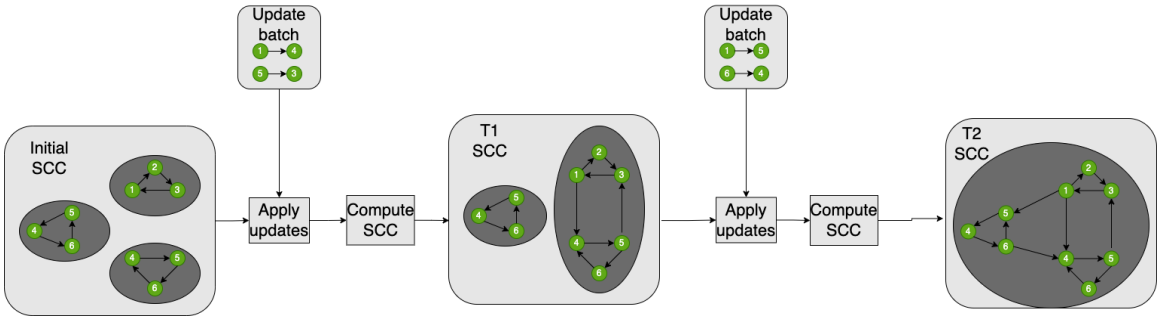


Figure 4. Example workflow for computing SCC on incremental graphs. At each timestep T, a new batch of updates is added and SCC is recomputed.

To overcome the challenges of parallelizing DFS on static networks, FW-BW (Forward-backward) approach was proposed [32]. To further improve the performance, trim techniques which fast reduce a large number of trivial SCCs (e.g., with one or two vertices, called trim-1 and trim-2, respectively) are introduced by[47]. Machine learning based optimizations have also been proposed in [90] and [112] for shared memory systems. To the best of our knowledge, there has only been one attempt to detect SCC on distributed networks [52]. For the most part, the state-of-the-art parallel approaches for detecting SCCs are optimized for shared-memory systems.

Figure 4, shows an example workflow of computing SCC for an incremental graph that changes over time. At each timestep T^n, a new batch of edge insertions are applied over the graph at time T^{n+1}. This type of edge addition is a very common workflow in scientific simulations that periodically keeps updating the graph databases. Currently, the standard approach to identify SCC in such a network is to recompute the SCC over the entire network every timestep, which is a costly operation. Although there have been attempts at an adaptive approach for other algorithms such as minimum spanning tree [132], single source shortest path [59],[133] and vertex coloring [58], adaptive approaches for identifying SCCs are limited in comparison. A recent approach for adaptive SCC detection is proposed in [8], which provides the Las Vegas algorithm for DAGs. However, they don't make considerations for parallel or distributed scalability. Considering incremental SCC detection is an unbounded problem, as explained in [29], there isn't a theoretical algorithm that solves it in polynomial time when there are both vertex and edge updates. As a result, we focus our work strictly on edge additions alone, keeping the number of vertices constant.

Our key contributions are as follows:

- A novel meta-graph storage format for caching intermediate SCC results after every new insertion batch. This format gives us a reduced-size graph for computing further SCCs after dynamic edge additions.
- DistSYNC, an asynchronous and distributed memory algorithm for identifying new SCCs after dynamic edge additions on the meta-graph format.
- A distributed memory implementation of DistSYNC using YGM[136], an asynchronous communication framework on top of MPI.

To the best of our knowledge, we propose the first distributed algorithm for incremental SCC detection. We also propose the first asynchronous incremental algorithm for SCC detection. The central concept behind meta-graph storage is the fact that we can treat identified components as meta-vertices and traverse over them to find new components rather than traversing every vertex in the graph. The rest of the paper is organized into five sections. In section 3.2, we introduce the required background information and related works. Section 3.3 dives into explaining the DistSYNC algorithm with the help of examples. The implementation details and experimental evaluations are covered in sections 3.4 and 3.5, respectively. We finally conclude this chapter with directions for future works.

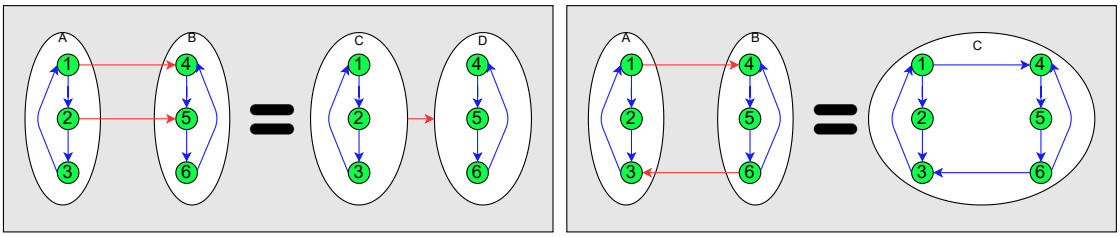
3.2 Related Works

Detecting SCCs in a network is a well-studied problem. This section discusses related work on detecting SCCs in a large-scale network. Our approach utilizes some of the concepts, so this section also provides prerequisite background information.

3.2.1 Sequential SCC. Tarjan's [143] implementation is the well-known sequential algorithm for detecting SCC. It uses DFS (depth-first search), and the complexity is $\mathcal{O}(V + E)$, where V is the number of vertices and E is the number of edges. There are many attempts to parallelize Tarjan's approach; however, all demonstrate poor scalability.

The Forward-Backward (FW-BW) algorithm [32] uses a recursive approach. The algorithm can be described as follows: Let V be the set of the vertices in the graph $G(V, E)$, $O(V)$ be the set of all outgoing edges in the graph, and $I(V)$ set of all incoming edges. Now for a given graph $G(V, O(V))$, a random pivot vertex

u is selected, and then a BFS is performed on $G(V, O(V))$ from a pivot vertex u to detect vertices (Let the set of these vertices be D) that can be reached from u. Next, another BFS is performed on $G(V, I(V))$ from the pivot vertex u, and a backward search is done where those vertices that can reach u are selected and inserted into P. The intersection of D and P forms SCC, which has the pivot element. Now from the original graph, the vertices identified in SCC are removed, and the FW-BW approach is recursively called on the remaining sets and the disjoint sets obtained after removing the vertices part of SCC from D and P. In the best-case scenario, it takes $\mathcal{O}(n \log n)$ to detect SCC. This approach was further improvised using trimming, which removes the vertices with zero in-degree and out-degree. Trimming reduces the number of vertices in FW-BW sets and speeds up the overall performance.



(a) Lemma 1 (b) Lemma 2

Figure 5. Example for Lemmas 1 and 2. Both the right-hand sides have the same structural properties as the left-hand sides.

3.2.2 Shared Memory SCC. Ji et al. [52] proposed a novel synchronization paradigm, called R-sync, to spanning tree-based detecting of SCC. This approach provides many benefits, such as early termination for conventional bottom-up traversal. The early termination allows them to check only a few neighbors and reduces the traversal compared to the conventional synchronization approach.

Hong et al. [47] identified the potential limitation of the FW-BW-Trim approach on large real-world networks. They proposed an extension of the FW-BW approach, which considers the characteristics of the dataset instances, such as the small-world property. Their implementation was the first attempt to develop a parallel algorithm to detect SCC and outperform the sequential Tarjan [143] implementation. Based on the small-world property, they have identified that wiring a few edges in the diameter of a real-world graph can shrink its size. Their main idea is to expand trimming operation and decomposing after the initial SCC is found based on partitioning on weakly connected components.

Slota et al. [127] proposed a shared memory multistep approach that uses a parallel BFS and graph coloring. They have used variants of FW-BW and applied Orzan's coloring method. To minimize synchronization, they avoid using locks. Their experiments on real-world graphs show better scalability on low-diameter networks. The coloring approach is also similar to FW-BW [131] with some modifications. Instead of just using one pivot, it uses multiple pivots. We use this approach to perform multi-threaded SCC within distributed processes.

3.2.3 GPU Implementation. Li et al. [71] proposed a GPU implementation of detecting SCC using the FB-BW-Trim algorithm. They present a hybrid method that allows the adoption of different parallelism strategies for various graph properties. Barnet et al. were the first to implement the FB-Trim algorithm using CUDA. Stuhl [137] extended the work by introducing an extended graph traversal implementation. Sthul ran experiments on the synthetic network and demonstrated good performance on synthetic networks and when running on real-world networks, except for one. The reason for poor scalability was due to the nature of the real-world graphs and skewed component sizes. Li et al. [71]

implement a hybrid method that detects SCC in two phases. In the first phase, the algorithm is only focused on detecting a single large SCC. In the second phase, the remaining small-sized subgraphs are processed. It is shown that identifying small-sized SCCs takes more time than identifying a single large SCC.

3.2.4 Dynamic Graphs. Attempts at getting batched/snapshot-based frameworks for graph algorithms are explored in [50] and [16]. STINGER [26] is a shared memory solution that can ingest structural changes at a rate of 10 million events per second with an updating kernel peak rate of around 1 million events per second. A Shared memory parallel algorithm for weakly connected components on dynamic graphs is given in [91] while a distributed implementation is given by [123]. There have been no parallel implementations for strongly connected components on graphs with edge insertions, let alone distributed algorithms.

3.3 Methodology

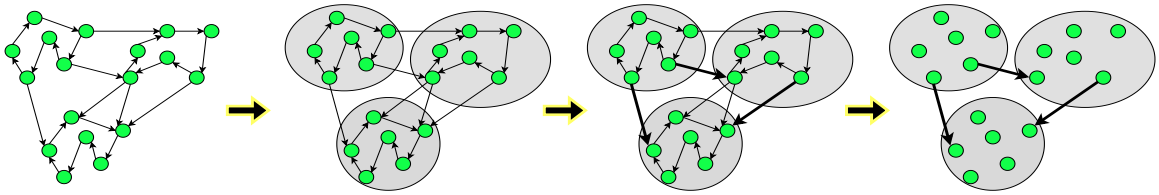


Figure 6. Converting initial graph to meta-graph. The initial graph is shown in the leftmost part. They are then segregated into different SCCs, which we refer to as meta-vertices. Only one representational edge that traverses two meta-vertices is converted as a meta-edge while the rest is discarded. We finally get a meta-graph with three meta-vertices and three meta-edges

Before discussing the details of the DistSYNC algorithm, we must first explain the motivation behind this algorithm, which stems from two lemmas on the structural property of graphs and strongly connected components.

Lemma 1. *Given two SCCs A and B, if there is a forward edge between any vertex in SCC A to a vertex in SCC B, we could say there is a forward path between all vertices in SCC A to all vertices in SCC B.*

Proof. If there exists a forward edge $e(a_k, b_k) \in E$ such that $\text{SCC}(a_k) = A$ and $\text{SCC}(b_k) = B$, then there exists a direct forward path $a_k \xrightarrow{Fwd} b_k$. Let, $SCC(A) = \{a_0, a_1, \ldots, a_k, \ldots, a_i\}$ and, $SCC(B) = \{b_0, b_1, \ldots, b_k, \ldots, b_j\}$. By definition of SCC, $a_x \xrightarrow{Fwd} a_k$ and $b_k \xrightarrow{Fwd} b_y \; \forall x, y$ where $x \in [0, i]$ and $y \in [0, j]$ where " $\xrightarrow{Fwd}$ " *denotes the existence of forward path.* By transitive property, $a_x \xrightarrow{Fwd} a_k \xrightarrow{Fwd} b_k \xrightarrow{Fwd} b_y$. Thus, $a_x \xrightarrow{Fwd} b_y : \forall x, y \; SCC(a_x) = A; SCC(b_y) = B$ □

In Figure 5a, there are two SCCs with labels A and B. Each SCC has three vertices, each labeled 1 through 6. There exists a forward edge between vertices 1 and 2 in SCC A to vertices 4 and 5 in SCC B. Since there is a forward path between all vertices within an SCC, we can say that vertex 1 in SCC A is reachable from vertex 3. Similarly, there is a forward path between vertex 4 and vertex 6. This added to the fact that there is a forward path between vertex 1 and 4 means there is a forward path between vertex 3 and vertex 6 through vertices 1 and 4 even though there is no direct edge between them.

Because of this structural property, maintaining any other edge across two SCCs is redundant information when identifying the components. As we can see on the RHS of Figure 5a, all the inter-SCC connections are replaced with one forward path between the two SCCs. By doing so, we can reduce the number of inter-SCC edges while representing the same structural information of the graph. This lemma could be extrapolated to any SCCs that are bigger than the provided example as long as there is one forward path that connects both the SCCs acting as a one-way bridge between the SCCs.

Lemma 2. *Given two SCCs, A and B, if there is a forward and backward path between them, then the vertices in both the SCCs could be merged into a single component.*

Proof. Let $S_1 = \{a_0, a_1, \ldots a_f, \ldots a_b, \ldots a_m\} \in SCC(A)$, $S_2 = \{b_0, b_1, \ldots b_f, \ldots b_b, \ldots b_n\} \in SCC(B) : \forall a, b \in V$.

Given $e_f(a_f, b_b) \in E : a_f \xrightarrow{Fwd} b_b$*(direct forward path)*

also, $e_b(b_f, a_b) \in E : a_b \xleftarrow{Bwd} b_f$*(direct backward path)*

Now, $\forall x \in [0, m]$

$a_x \xrightarrow{Fwd} S_1 \cup S_2$	*S1 by definition of SCC; S2 through* e_f *and Lemma 1*
$a_x Bwd S_1 \cup S_2$	*S1 by definition of SCC; S2 through* e_b *and Lemma 1*

Similarly,$\forall y \in [0, n]$

$b_y[Bwd]Fwd S1 \cup S2$	"$[Bwd]Fwd$" *denotes existence of path from both direction*

Therefore, $a_x[Bwd]Fwdb_y$

Hence, we can merge A and B into single SCC C such that C consists of all vertices in $S_1 \cup S_2$ □

On the LHS of Figure 5b, there are two SCCs with three vertices, each labeled 1 through 6. We can see that there is a forward path from vertex 1 to vertex 4 while there is a backward path to vertex 3 from vertex 6. Once again, from both the fact that there is a forward path between any two vertices within the same SCC and there is a forward and backward path between the two SCCs, we can say there is a forward path between any two vertices from both the SCCs and hence all

the vertices belong to the same SCC. This is represented on the right side of Figure 5b.

By applying Lemma 2, we can represent two SCCs residing in different processes as a single component, thereby reducing the unique components we need to represent the same structural information of the graph. Like Lemma 1, this can be extrapolated to any number of SCCs of any size as long as a forward and backward path exists among them.

3.3.1 Meta-graphs. We introduce the meta-graphs $G''(MN, ME)$ as an abstraction on top of the existing graph where the vertices (referred to as meta-nodes(MN) or meta-vertices), are the SCCs. The edges of G'', which are referred to as meta-edges(ME), are the directed edges that connect two meta-vertices. They are created by first identifying the various SCCs in the original graph. These SCCs act as the meta-vertex, with each meta-vertex comprising all the vertices that belong to that SCC. From lemma 1, we know that for computing SCC, when multiple redundant edges traverse two different SCCs, then they could be represented using a single edge while still holding on to the same structural property. Thus a single representational edge that traverses the two SCCs/meta-vertex is chosen as the meta-edge while the rest is discarded.

We take into account the direction of the edge so that only edges that traverse in the same direction are considered redundant, and a representational edge is picked from them. Meta-graphs are much smaller than the original graph because all the vertices belonging to an SCC can be represented using a single label (color). For instance, the original graph with 18 vertices and 23 edges in Figure 6 is converted to a meta-graph with three meta-vertices and three meta-edges. It is also to be noted that initially, a meta-graph is directed and acyclic (DAG), as multiple

SCCs that form a cycle cannot exist without being absorbed into a single SCC. We will be leveraging this fact to overlay dynamic edge additions on top of this meta-graph to see if new cycles are formed to identify updated SCC. Storing graphs in a meta-graph format enables us to recompute the SCC on a reduced-size meta-graph rather than the larger original graph. Also, a lot of real-world graphs consist of many large SCCs, which in turn could be represented using a single label. Hence, meta-graph abstraction drastically reduces the size of these graphs. This is one of the key factors for the improved performance of our algorithm.

3.3.2 Forward color propagation and backward confirmation messages. Forward color propagation sends the color of a pivot meta-vertex to its next neighbors, along with the accumulated size of all the meta-vertices in that chain originating from the pivot. These are implemented as functions that can be executed by the neighbor vertex using a remote procedure call (RPC). The neighbors recursively keep calling that function and pass it further to their neighbors until the base condition is reached. This is a mechanism we will use in DistSYNC to identify whether a chain is a cycle. Likewise, backward confirmations are recursive messages sent to the previous sender of a forward color propagation message notifying the presence of a cycle for that chain along with its total size. Meta-vertices asynchronously fire these messages to destination meta-vertices when they are asked to and go on to wait for further messages.

3.3.3 DistSYNC. The critical steps in our distributed algorithm for SCC are bookkeeping the forward and backward messages sent from different pivots. Figure 7 explains the workflow of our algorithm using a sample graph with four initial SCCs, denoted by four colors, which we partitioned across two distributed ranks. The four components are denoted using the starting letters of

their respective colors, namely, R, G, B, Y. It is to be assumed that the vertices within the same color are interconnected, but for convenience, we show only the edges that go across different colors denoted by a single arrow. Newly updated edges that arrive in the first time step are denoted by dotted arrows. We can see that in the initial phase, there are two existing edges, RG and GB, along with two updated edges, BR and BY.

In *phase one*, we build a meta-graph with four meta-vertices denoting the four colors and four meta-edges denoting the new and existing edges between them. Then to trigger the start of forward checks in *phase two*, all the meta-vertices that are a source to a newly updated edge that traverses to another meta-vertex are considered pivots. They then forward propagate their colors and size to the respective destination meta-vertex. In Fig 7, the solid yellow connectors denote the forward propagation through BR and BY as they are newly updated meta-edges. R, upon receiving a propagated message from B, forwards that same color downstream to G along with the combined size of both of them. This is shown in fig 7 with a solid yellow connection RG. Throughout the forward chain, everyone propagates the pivot color along with the updated size in the chain. When B receives a forward message from G with itself as the pivot, it recognizes that it is a cycle. When a cycle is detected, it checks if its current size is less than the size of the chain and triggers a backward confirmation to the sender of that message. In this case, the size of the chain is 16, which is greater than that of B. So B updates its colors and triggers a backward confirmation with the new size to G, which in turn does the same to R. So R and G are now essentially subsumed by B, with 16 being their new size.

A case where the size of the meta-vertex is greater than the size taken from the backward confirmation message could only happen if that vertex has been updated to a new color after it sent out the previous forward message. In that case, it doesn't propagate backward confirmation but instead asks the successor who sent the backward confirmation to revert to its previous state. By that point, it would already have sent forward propagation messages with the new chain sizes and pivot.

The algorithm terminates when there are no more forward, backward, or revert messages to send through the entire network, which would mean every meta-node is correctly updated to reflect the new component it belongs to and the new size. Individual forward, backward, and revert messages with different pivots would flow back and forth through the network until everyone hits a stable state and there are no more new messages. Since this is completely asynchronous, none of the meta-vertices wait in anticipation of a backward confirmation after a forward check. The meta-vertices process these messages as they are received and go back to listening for further messages until the algorithm terminates.

In essence, DistSYNC is an extended version of cycle detection but over a meta-graph. Detecting cycles is expensive, but we highlight two key advantages that ensure that DistSYNC is efficient. Since it operates over a meta-graph, the number of edges it traverses to detect a cycle is dramatically reduced. Also, the number of pivot points that trigger forward checks is limited to only the meta-nodes that have newly updated meta-edges. This number is dependent on the size of the update batch; a larger batch means more pivot points that slow down the entire process, as we discuss in Section 3.5, but we can cleverly partition the update batch to maximize efficiency. In the worst case, every meta-edge can be traversed to find the cycle, giving it an amortized time and communication complexity of

$\mathcal{O}(MV + ME)$. Due to the usage of distributed hash tables from YGM, each process only stores the meta-vertices that belong to it. Hence, the space complexity is $\mathcal{O}((V + E)/N)$ where N is the number of processes.

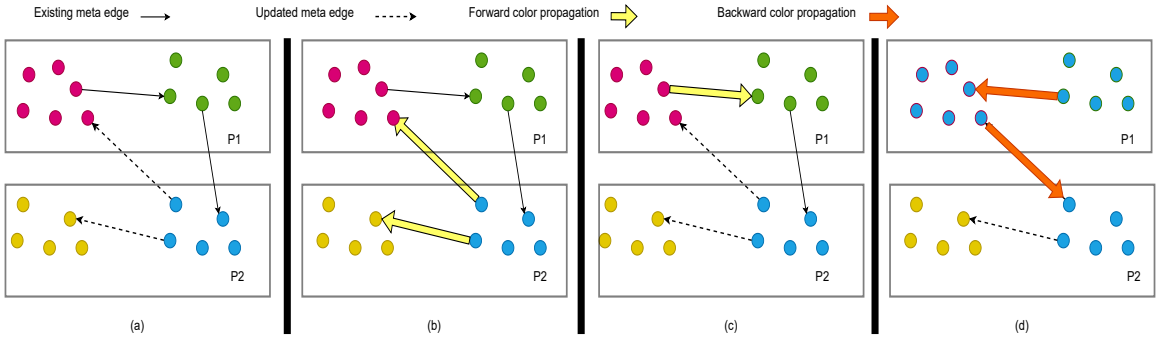


Figure 7. Example workflow of DistSYNC. The initial meta-graph is shown in (a) with four different SCCs/meta-vertices split across two ranks, p1 and p2. Blue forward propagates its color to yellow and red based on the updated edge in (b). In (c), red, in turn, propagates blue forward to its neighbor green. Green changes its color to blue and confirms it backward to red in (d). Red does the same and propagates it backward to blue. Now, everyone in the cycle is blue.

Algorithm 1 explains the procedures required for forward and backward propagation, respectively, while algorithm 2 puts everything together and explains the full DistSYNC algorithm. An arrow denotes a remote-procedural call where the sender asks the LHS of the arrow to execute the function on the RHS with the given arguments. After constructing the initial meta-graph, every process spawns a non-blocking listener thread to receive forward and backward messages from other processes. This is shown in lines 1-3 in algorithm 2. Meanwhile, the application thread skips to line 4 and initiates a forward check for all newly updated meta-edges. The forward check is done in lines 1-14 of algorithm 1.

Algorithm 1 Forward and backward propagation

```
1: procedure CheckForward(pivot, sz)
2:     if self == pivot then
3:         if sz > size then
4:             size = sz
5:             Forwardsender
   →ConfirmBackward(pivot, sz)
6:         end if
7:     end if
8:     if self != pivot then
9:         sz= sz + size
10:        for Each forward neighbor y do
11:            y →CheckForward(pivot, sz)
12:        end for
13:    end if
14: end procedure
15: procedure ConfirmBackward(pivot, sz)
16:    if sz > size then
17:        color(self) = color(pivot)
18:        size = sz
19:        Forwardsender
   →ConfirmBackward(pivot, sz)
20:    end if
21:    if sz < size then
22:        confirmationsender →Revert(pivot)
23:    end if
24: end procedure
25: procedure Revert(pivot)
26:    revert to the previous state
27:    confirmationsender →Revert(pivot)
28: end procedure
```

Algorithm 2 DistSYNC

Require: list of meta-edges ME and meta-nodes MN

```
1: for All meta-nodes ∈ MN do
2:     Listen to forward or backward message
3: end for
4: for each edge x, y ∈ ME do
5:     if Meta-edge x, y is new then
6:         y →CheckForward(x, size)          ▷ x asks y to execute CheckForward
7:     end if
8: end for
9: Barrier()
```

When a destination meta-vertex receives a forward check message, it propagates that color and updated size to all its immediate neighboring meta-vertices. In lines 1 and 2 of Algorithm 1, a meta-vertex checks if it is the received pivot and if the size is bigger than itself, which determines that a cycle is detected. If so, it triggers a backward confirmation in line 5. If its size is greater than the received size, it means that it has since been updated, so it doesn't send a backward confirmation. If the meta-vertex is not a pivot, it just routes the forward message to its neighbors with the updated size shown on line 11.

3.4 Implementation

This section will go over the details of implementing DistSYNC algorithm with the design choices and selection of data structures for the most efficient approach. The distributed algorithm was implemented in C++ using YGM, an asynchronous communication framework with a built-in suite of distributed data structures. YGM uses MVAPICH under the hood for scaling the application across distributed processes.

3.4.1 Partitioning. This step may not be a part of the two phases of DistSYNC, but it is essential that the graph is partitioned efficiently by avoiding load imbalance and minimizing inter-process communication. For partitioning

the input graph, we use ParMetis[57], a multi-way partitioning library. ParMetis internally uses Kernighan-Lin algorithm [44] which allocates partitions to vertices such that the number of inter-partition edges is minimized, i.e., the inter-process communication load is reduced.

3.4.2 YGM. The presence of non-uniform communication patterns in large-scale graph algorithms makes regular MPI an inefficient framework for this algorithm. Also, the absence of distributed data structures like hash maps and sets makes it hard to code complicated distributed algorithms. For these reasons, we chose to use YGM as our base framework. YGM uses RPC-style fire-and-forget semantics for its communication interface. Messages in YGM have three basic components: a function to execute, arguments to pass to the function, and an MPI rank at which to evaluate the function. The procedures for forward and backward checks are encapsulated into asynchronous messages and fired to a destination rank that holds the receiving meta-vertex. The destination rank receives that message and executes that procedure with its arguments. In this particular case, the CheckForward() and ConfirmBackward() procedures from algorithm 1 are sent as asynchronous RPC messages.

The fact is that DistSYNC, like any graph algorithm, generates large numbers of small messages. Hence, YGM provides message buffering capabilities that bundle together multiple small messages between a sender and receiver to reduce the total number of remote MPI messages underneath and thus improve bandwidth. Lastly, to communicate non-fixed width data structures, YGM serializes the structured messages to variable length byte arrays. This gives us the flexibility of communicating complicated structures without having to take a performance penalty.

Table 3. Details of benchmark datasets

Dataset	Initial edges	# of edges in update	# of SCCs	Largest SCC size	Best iSpan time(MS)	Best DistSYNC time(MS)
Flicker(Fl)	1,151,463	1,158,925	487,659	9,752	21.3	6.24
Facebook(Fb)	67,255,691	67,255,782	1,576,432	963,487	230.4	98.2
Orkut(Or)	122,346,784	122,346,157	2,975,565	1,865,468	349.11	146.52
Roadnet-USA(Rusa)	93,568,872	93,568,112	3,501,682	443,923	248.6	279.4
RMAT26(R26)	67,108,864	67,107,927	1,136,282	1,082,223	215.6	75.4
RMAT27(R27)	734,217,728	734,279,598	9,987,245	92,742,613	2472.7	733.4

3.4.3 Initial SCC computation. Initially, each process needs to compute the local SCCs of its allocated subgraphs. For computing the initial SCCs, we use Multistep [127], a shared memory implementation that uses a combination FW-BW-Trim and their novel BFS coloring algorithm. It is implemented in C++ with OpenMP directives, and we created a wrapper module to interface Multistep within our framework. It takes in an edge list representation of the allocated subgraph and produces a vector with indices representing the vertex IDs and SCC IDs as values.

3.4.4 Data stuctures. To perform DistSYNC algorithm for dynamic graphs, we need to accurately track forward and backward meta-edges from every meta-vertex. But these vertices would reside across different ranks, so traditional data structures would not be sufficient. For this purpose, we make use of the suite of distributed data structures provided by YGM. In specific, we use distributed hash maps and hash sets. YGM internally stores these tables across many ranks and provides constant time lookup for every entry into the map. If a process tries to look up a particular key that isn't stored in that process, it queries the other process that holds that key and returns its value. This seamlessly happens underneath while the interface provides a global view of that hash table and hence enables every process to look up every key-value pair of the distributed map.

The color associated with each vertex is stored in a YGM hash map accessible by every process. The list of neighbors for every meta-vertex is also

stored in YGM hash map. This lets each process forward and backward propagate colors to its neighbors by looking them up in constant time with at most two hops. Lastly, every process builds a set of all forward propagated colors it received. Hence, when it receives a backward propagated color, it checks this set for an equivalent match and updates its color if it finds one. This is a way of checking forward and backward paths before updating colors.

3.5 Experiments and Results

In this section, we will discuss the results from running distributed experiments starting with the strong scaling results in comparison with the iSpan[52] as the baseline. Then we compare the speedups of DistSYNC with varying batch sizes of dynamic updates. Then lastly, we discuss the performance metrics of DistSYNC using YGM, including average memory utilized and interprocess communication bandwidth.

3.5.1 Experimental setup. We ran the distributed experiments on Intel Xeon dual E5-2690v4 processors with 28 cores per node. The timing for these experiments is recorded after the creation of initial meta-graphs to recompute the SCC for a dynamic batch of edge additions. For our benchmark datasets, we used four real-world graphs; Flicker (Fl), Facebook (Fb), Orkut (Or), and Roadnet-USA (Rusa), along with two synthetic graphs; RMAT26 (R26) and RMAT27 (R27). The RMAT was generated with the probabilities (a=0.45, b=0.15, c=0.15, d=0.25) and scale-free degree distribution. Table 3 gives more details about the datasets.

The baseline iSpan is not a dynamic model that can handle batches of updates; hence, in the iSpan experiments, we recompute the SCC over the entire graph, i.e., the initial graph plus the update batch. This is currently the only available way to compute distributed SCC on incremental graphs. The graph

properties give us an idea of how big the meta-graphs will be. For example, larger SCCs would mean more vertices can be grouped under a single meta-vertex and, in turn, reduce the total number of meta-vertices. This reduction in the total number of SCCs enables us to create a meta-graph that is much smaller than the original graph and hence significantly reduces the cost of recomputing SCC.

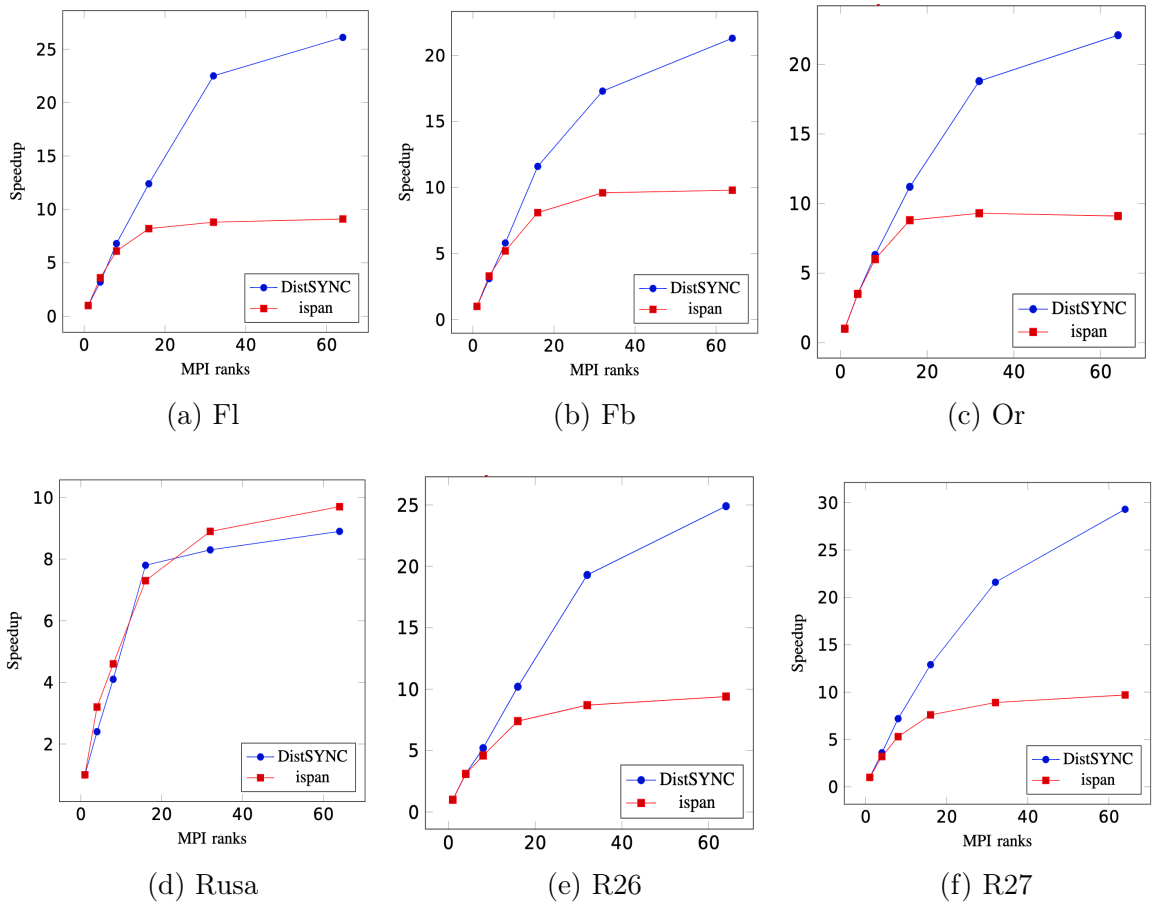
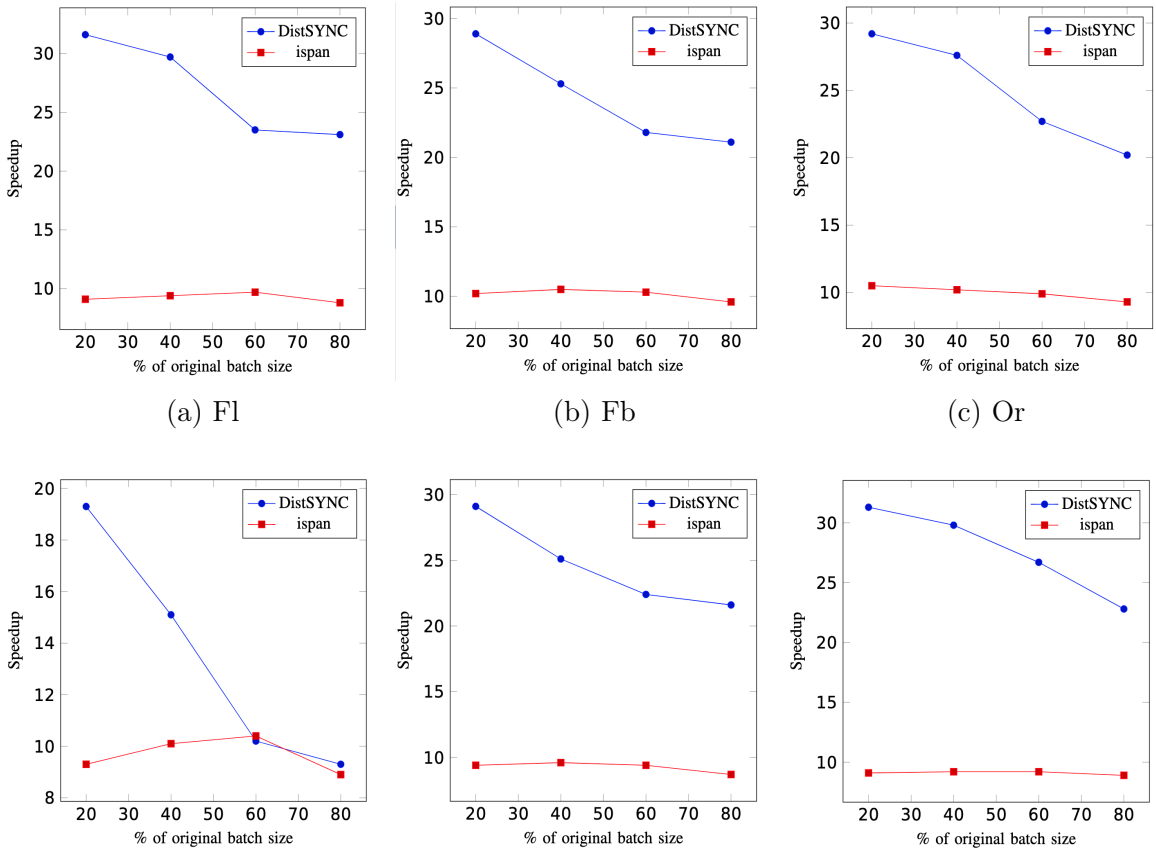


Figure 8. Speedups for DistSYNC (blue) and baseline iSpan (red) with respect to single-threaded Tarjan's implementation.

The first and primary metric for analyzing performance is strong parallel scaling. Figure 8 shows the strong scaling results of DistSYNC and iSpan for a single-threaded implementation of Tarjan's algorithm. DistSYNC scales well, with max speedups ranging from 8x to 28x for all graphs on up to 64 processes. In

comparison, iSpan scales with max speedups ranging from 6x to 9x. DistSYNC can also outperform iSpan in all but one dataset, namely, Roadnet-USA(Rusa), with its speedups flatlining at 8x. This is because the size of the largest SCC is significantly smaller, and the number of SCCs is large. This dataset essentially has a lot of small SCCs. This means the meta-graph also has a lot of meta-vertices, thereby reducing the impact of constructing a meta-graph.



(a) Fl (b) Fb (c) Or

(d) Rusa (e) R26 (f) R27

Figure 9. Speedup on 32 cores with varying batch sizes for DistSYNC (blue) and baseline iSpan (red) with respect to single-threaded Tarjan's implementation.

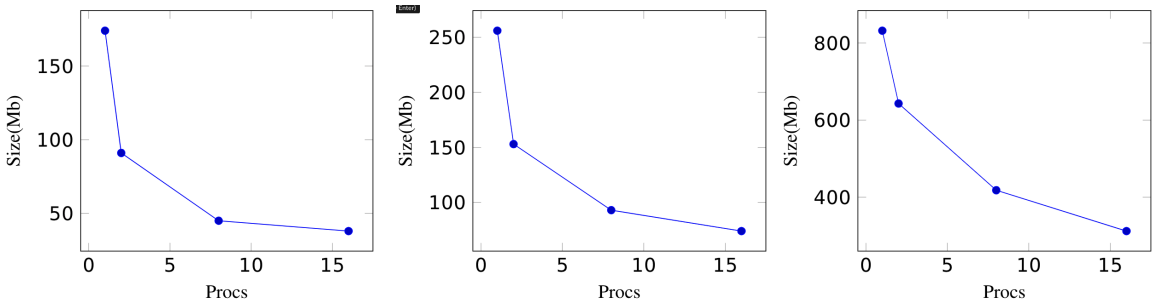
DistSYNC performs at its worst when the original graph consists of many small SCCs, but those types of graphs are much rarer because of the small-world

property of graphs, as explained in [47], where vertices of real-world graphs usually cluster together to form few large SCCs followed by several medium to small SCCs.

Apart from the number of SCCs, the other factor influencing the performance of DistSYNC is the size of the update batches. Figure 9 shows us the speedups of DistSYNC and iSpan at 32 nodes when varying the size of the update batch. The X-axis denotes the percentage of the initial batch size. For instance, 20% of the batch of edges are kept for updates while the remaining 80% are added to the initial graph to keep the overall size constant. We perform these experiments on 20, 40, 60, and 80% batch sizes.

We observe that DistSYNC is at its fastest for smaller update batch sizes, and the speedup gradually decreases as we increase the batch size. This is because at smaller batch sizes, the number of new edges to be added is smaller, and hence, fewer forward and backward messages are triggered by each process. We note that the overall size of both batches combined remains constant. Only the allocation of edges between the initial batch and the updated batch varies in order to highlight the impact of the size of the update batch on the final performance of DistSYNC.

In contrast, the performance of iSpan remains fairly constant across all batch sizes as the size of updates will not influence it, considering it is not a truly dynamic model and recomputes the SCC for the entire graph every time there is a new batch. We perform this set of experiments to elucidate the benefits of using a dynamic algorithm for graphs with incremental updates. These benefits are amplified further when there is more than one iteration of updates because, for every iteration, only the new batch of meta-edges will be considered for pivots while the previous iterations will be baked into the meta-graph. This is unlike any of the



(a) Fl (b) Or (c) R27

Figure 10. Average memory utilized per process as the number of processes increases.

currently available state-of-the-art parallel SCC algorithms, where the entire graph and incoming updates need to be recomputed for every update.

3.5.2 Memory Utilization. Distributed implementations always have the added advantage over shared memory implementations of reducing memory utilized per process to perform massive-scale computations. In this section, we highlight memory utilization to demonstrate the benefits of using YGM as the distributed substrate for DistSYNC. Figure 10 gives us the average memory utilized per process as we increase the number of processes. We can see that for all the datasets, the average memory utilized decreases fairly consistently as we scale up. A significant contributor to this decrease is the use of distributed data structures provided by YGM. Considering that DistSYNC uses hash tables to keep track of forward and backward connectivity for each vertex while using hash sets to keep track of all vertices in a meta-vertex, the number of hash entries can be significant if we use a traditional hash table. The YGM hash table stores only entries of vertices that belong to that process while looking up external entries by exchanging messages between the processes. This is seamlessly handled under the hood by YGM, while all the entries appear as one unified hash table for the user. To the

best of our knowledge, the state-of-the-art iSpan replicates the entire graph on all the ranks, so memory utilization doesn't scale with increasing ranks.

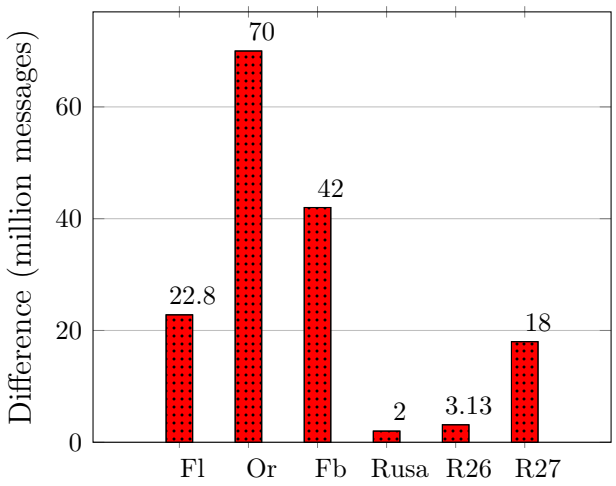


Figure 11. Reduction in the number of MPI messages by coalescing in YGM.

3.5.3 Message coalescing. YGM also enables us to coalesce multiple messages between two processes and send them as a single message. These messages are stored in a fixed-size 512 Kb buffer in each process. When the buffer is filled up or when it is forced to flush, it is sent to the destination process, which handles all the incoming messages. These messages are usually forward or backward check messages that contain individual lookups of vertices in hash tables. Since these are extremely small messages, sending them individually in MPI would significantly increase the total number of remote communications and result in poor bandwidth utilization. The order in which these messages are handled wouldn't matter, as this is a completely asynchronous algorithm. At worst, a process would start checking the backward path while still waiting for confirmations on forward paths from other processes, in which case, it would simply have to trigger a new forward check once it is updated.

Figure 11 shows the reduction in the number of MPI messages by coalescing small messages in YGM. In the Orkut graph, the difference is amplified as

it consists of a relatively small number of densely packed meta-vertices that frequently communicate with a small subset of processes. Coalescing these messages ensures the buffers are filled before sending it as an MPI message. Likewise, Roadnet-USA has the least difference due to its relatively large number of meta-vertices communicating with a larger subset of processes and, hence, sending sparse buffers before filling them fully. As a result, the total number of messages is increased. This is another reason why DistSYNC is less performant for many small SCCs.

3.6 Conclusion

This chapter introduces DistSYNC, an asynchronous algorithm leveraging distributed, multithreaded CPU parallelism for identifying Strongly Connected Components(SCC) in incremental networks with edge additions. We have also supported the algorithm with the implementation details of the distributed framework. We show that our approach can offer performance speedups of up to 30x over single-threaded Tarjan's implementation and up to 2.8x over the state-of-the-art. In the future, we plan to extend DistSYNC with edge deletions. We are also working on caching meta-graphs in persistent memory like NVMe ssd. This would enable us to read and write meta-graphs much more quickly in between each timestep for efficient recomputation.

CHAPTER IV

MODEL-BASED ALGORITHM SELECTION

4.1 Introduction

Chapter IV incorporates content from a paper published at ICPE 2019, which presents joint work on scalable graph analytics. Dr. Boyana Norris contributed to the conceptualization and supervision of the core methodology. Dr. Sam Pollard was involved in the design of the Easy-Parallel-Graph framework and the collection of hardware metrics.

Our research is motivated by the current state of parallel graph processing. Fields such as social network analysis [56] and computational biology [110] require the analysis of ever-increasing graph sizes. The wide variety of problem domains is resulting in the proliferation of parallel graph processing frameworks. The most comprehensive survey, released in 2014, identified and categorized over 80 different parallel graph processing systems [24] not including domain-specific languages such as Gremlin [119].

An overarching issue among these systems is the lack of comprehensive comparisons. One possible reason is the considerable effort involved in getting each system to run: satisfying dependencies and ensuring data are correctly formatted can be time consuming tasks. Moreover, some systems are developed with large, multi-node clusters in mind while others only work with shared memory, single-node computers. An example of the former is GraphX [159], which incurs some overhead while gaining fault tolerance and an example of the latter is Ligra [126], a framework requiring a shared-memory architecture. In addition, there is a growing number of systems designed to run on GPUs [169].

We take inspiration from the Graph500 benchmark [98], which clearly specifies every step of the breadth first search (BFS) algorithm and how it should be timed. The Graph500 can be used to rank systems fairly on a single, well-defined benchmark. In this context, *system* is defined as the hardware, operating system, middleware, and algorithmic implementation which accomplishes a certain task, in this case BFS.

While leaderboards indicate computational milestones, the top implementations are not easily generalizable to new datasets and hardware. Reference implementations, however, are critical for advancement; the complexity of today's hardware and operating systems requires empirical evidence to assess an algorithm's performance. In addition, there is motivation to define basic building blocks for graph computations [1, 11]. However, new algorithm designers and users alike have no easy way to accurately determine what constitutes a high-performance implementation.

To address these issues, we introduce *easy-parallel-graph-*$*$ (EPG*), a framework which simplifies the installation, comparison, and performance analysis of four widely-implemented graph algorithm building blocks: breadth first search (BFS), single source shortest paths (SSSP), PageRank (PR), and Triangle Counting (TC). With this framework we select a small number of graph processing libraries over which we homogenize all aspects of execution to compare performance fairly.

EPG* automates the following:

- Installation and building of graph processing packages.
- Downloading or generation of real-world and synthetic datasets.
- Running and collecting experimental results.

Table 4. Categories of Dynamic Graph Algorithms

Category	**Goal**	**Examples**
Dynamic Connectivity	Maintain information about connected components	Link-cut trees, Euler tour trees
Dynamic Shortest Path	Update shortest paths as edges/nodes change	Dynamic Dijkstra, Even-Shiloach Tree
Dynamic Centrality	Maintain centrality scores as structure evolves	Incremental betweenness/degree centrality
Dynamic Spanners	Maintain subgraphs that approximate distances	Thorup-Zwick Spanners
Dynamic Matching	Track approximate or exact matchings	Baswana-Gupta Algorithm
Dynamic MST	Maintain minimum spanning trees	Frederickson's topology trees
Temporal Path Algorithms	Incorporate timestamps or durations into path queries	Time-respecting BFS, temporal reachability
Streaming Graph Algorithms	Handle high-volume updates in real-time	Sketch-based triangle counting, graph sketches

– Statistical analysis and visualization of performance characteristics.

– Creating machine learning models for selecting well-performing graph algorithm implementations based on input graph properties.

In addition, the research described in this paper includes the following contributions.

– Manual code analysis to ensure all implementations time execution phases similarly and follow the same stopping criteria.

– Reproducible analysis of the performance, energy consumption, and scalability of graph algorithm implementations.

– Recommendation of well-performing algorithms selected among different packages based on properties of the input graphs. To illustrate our approach,

we show that for a given performance objective, number of edges processed per second, the average performance of the algorithms recommended by the model is between 6% (for PageRank) and 700% (for BFS) better than the average over all algorithms. The same approach can be applied to other objective functions, such as energy or power.

To be clear, we are not proposing a new benchmark suite or providing new reference implementations. Instead, we introduce a method to compare and recommend existing software packages without requiring the user to be familiar with the underlying implementation details. We provide EPG* as open-source at `https://github.com/HPCL/easy-parallel-graph`. To illustrate the use of this framework, we include results from our experiments on both real-world and synthetic datasets.

Section 4.2 describes the architecture of EPG* and the algorithms, datasets, and graph processing packages it uses. Section 4.3 presents some visualizations from EPG* and discusses possible explanations for the results. Section 4.4 explains the machine learning models we use and their results. Section 4.6 overviews related work. Section 4.7 outlines future research directions and is followed by the conclusion.

4.2 Methodology

Our framework breaks the process of characterizing performance into five principal phases shown in Figure 12. The five phases are as follows.

1. Installing modified, stable forks of each software package to ensure reproducibility.

2. Given a graph file, generating the input necessary to run each software package.

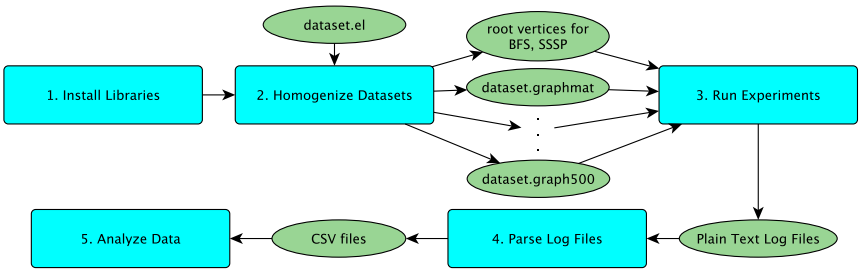


Figure 12. Overview of EPG*. Cyan boxes represent processing steps, green ellipses represent intermediate data, and yellow ellipses represent outputs.

3. Given a graph and the number of threads, running each algorithm using each software package multiple times.

4. Processing the log files to collect performance data.

5. Analysis of results, by either

   (a) training a model (machine learning)

   (b) generating figures (human analysis)

Step 5b includes analysis of power usage, energy consumption, execution time characteristics, such as the number of iterations (for PageRank), time for I/O, and the time to construct the graph data structures in memory. Moreover, our approach requires little knowledge of the inner workings of each system; the data are collected by either parsing log files (for execution time) or hardware counter sampling of model-specific registers (for power). This allows arbitrary datasets, packages, and algorithms to be analyzed, as long as the output is formatted consistently.

4.2.1 Installing Libraries. The libraries are stable forks of the given repositories which are configured to ensure the experiments execute in the same manner.

Our experiments use the author-provided implementations with modifications only to insert performance analysis hooks or to ensure homogeneous stopping criteria; we assume the developers of each system will provide the best performing implementation.

4.2.2 Datasets. Homogenizing the datasets creates copies of the graph files and auxiliary files in various formats. This is both to ensure they are correctly formatted for each system and to speed up file I/O whenever possible.

We refer to a graph's *scale* when describing the size of the graph. Specifically, a graph with scale S has 2^S vertices. For example, many of our experiments were performed on graphs with $2^{22} = 4,194,304$ vertices. We measure parallel efficiency and speedup by varying the number of threads from one to the total number of threads available on our research server, 72.

Each experiment is run multiple times to account for algorithmic and OS variation. For BFS and SSSP, which start at a given root, we select the roots randomly in accordance with the Graph500 such that each root has a degree greater than 1. For PageRank, we simply run the algorithm multiple times. Thus we get distributions of execution times, typically visualized with boxplots.

When possible, we measure the dataset construction time as the time to translate from the input file data in RAM to the graph representation on which the algorithm can be performed. This is not possible for PowerGraph and GraphBIG because they read in the input file and build a graph simultaneously.

We use the Graph500 synthetic graph generator, which creates a Kronecker graph [69]. For building the machine learning model, we use a grid search across 64 different parameterizations. For the visualizations, we use the same parameters as the Graph500: $A = 0.57, B = 0.19, C = 0.19$, and $D = 1 - (A + B + C) = 0.05$ and

set the average degree of a vertex as 16. Hence, a Kronecker graph with scale S has 2^S vertices and approximately 16×2^S edges.

We use 25 different real-world graphs and a grid search of 64 different Kronecker graphs to train our models. The real-world graphs come from the Stanford Network Analysis Project (SNAP) [70] or KONECT [65]. Adding additional graphs is as easy as adding the URL and dataset name to a configuration file.

4.2.3 Graph Processing Systems. This study explores five shared memory parallel graph processing frameworks and one distributed memory framework operating on a single node (Powergraph). The first three are so-called "reference implementations" while the remaining three are included because of their performance and popularity. Other popular libraries such as the Parallel Boost Graph Library [39] are not considered here because the authors do not provide reference implementations of the algorithms we chose to analyze. We consider the following frameworks.

1. The Graph500 [98] consists of a specification and reference implementation. We use a modified version most similar to 2.1.4 using OpenMP for parallelism. The Graph500 uses a compressed sparse row (CSR) representation.

2. The Graph Algorithm Platform (GAP) Benchmark Suite [6] is a set of reference implementations for shared memory graph processing. The author of GAP has contributed to the Graph500 so the BFS implementations are similar. Additionally, GAP uses OpenMP to achieve parallelism and uses a CSR representation.

3. GraphBIG [100] benchmark suite. We consider only the shared memory solutions but GraphBIG also provides GPU benchmarks. GraphBIG uses a CSR representation for graphs and OpenMP for parallelism.

4. GraphMat [138], a library and programming model along with reference implementations of common algorithms. GraphMat uses a doubly-compressed sparse row representation and OpenMP for parallelism.

5. PowerGraph [37], a library and programming model for graph-parallel computation. Parallelism is achieved via a combination of OpenMP and light-weight, user-level threads called fibers. Powergraph uses a novel storage scheme on top of CSR.

6. Galois [106] is a framework for programming graph implementations in C++. Once a program has been adapted for Galois, parallelism is achieved implicitly via fine-grained task-based runtime.

4.2.4 Algorithms. We consider four parallel algorithms: Breadth First Search (BFS), Single Source Shortest Paths (SSSP), Triangle Counting (TC), and PageRank (PR), though not all algorithms are implemented on all systems. We inspected the source code of the surveyed parallel graph processing systems to ensure the same phases of execution are measured across all implementations.

We take inspiration from the Graph500 [98]. Benchmark 1 (BFS) measures two kernels: creation of the graph data structure given an unsorted edge list stored in RAM and BFS[1]. Where possible, we measure data structure contruction time in addition to execution time. We select SSSP because it is also included in the Graph500 specifications. We select PageRank and Triangle Counting because of

[1]For a complete specification, see `http://graph500.org/specifications`.

their popularity; most libraries provide reference implementations. One challenge with using PageRank is the stopping criterion; all implementations have been modified to use $\|p_t - p_{t-1}\|_1$ (the absolute sum of differences) where p_t is the PageRank vector at step t. Verification of the PageRank results is beyond the scope of this paper, although this may explain some of the large performance discrepancies.

Our approach is not specific to a particular algorithm; measuring the execution time, data structure construction time, and power consumption can be applied easily to other implementations. We hope this motivates others to use the framework for their own work.

4.2.5 Machine Specifications. We performed the visualization experiments on our 36-core (72 thread) node with 256GB DDR4 RAM and two Intel Xeon E5-2699v3 CPUs. The operating system is GNU/Linux version 4.4.0-22. Our code was compiled with GCC version 4.8.5 with the exception of GraphMat which was compiled with the Intel compiler version 17.0.0. The learned models were trained on dual-socket Intel E5-2690v4 nodes with 128GB DDR4 RAM.

4.3 Performance Analysis

We analyze the performance of the algorithms described in Sec. 4.2.4 in terms of execution time, scalability over multiple threads, and power and energy consumption. All figures except those measuring scalability (Figure 16) use 32 threads because 32 threads resulted in the lowest execution times on average.

Our results for the Kronecker graph of scale 22 (4,194,304 vertices and about $16 \times 2^{22} \approx 33.5$M edges) are presented in Figs. 13, 14, and 15.

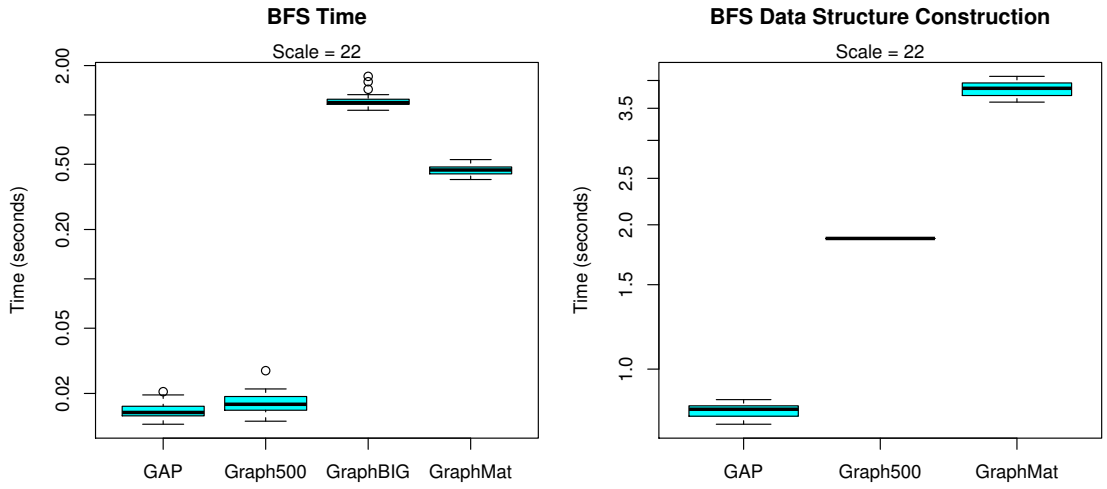


Figure 13. The y-axes are logarithmic. The left box plot shows the time to compute BFS on 32 random roots while the right plot shows the times to construct the graph for each system. The Graph500 only constructs its graph once. GraphBIG reads in the file and generates the data structure simultaneously so is omitted.

4.3.1 Synthetic Graphs. Our results for the Kronecker graph of scale 22 (4,194,304 vertices and approximately $16 \times 2^{22} \approx 33.5M$ edges) are presented in Figs. 13, 14, and 15.

Figures 13 and 14 show performance results for a Kronecker graph with scale 22. The box plots give an idea of the execution time distributions. There is less variance in the execution times of SSSP (between 0.1 and 1.7 seconds) compared to BFS (0.01 and 1.7 seconds), but GAP is the clear winner in both cases. The data structure construction times for GAP and GraphMat are consistent; in both cases, the platforms create the same data structure for both algorithms. These results are consistent with [138], which lists GraphMat as better performing than PowerGraph for SSSP.

The behavior of PageRank is slightly different. As with SSSP and BFS, the GAP Benchmark Suite is the fastest, but it also requires the fewest iterations. We

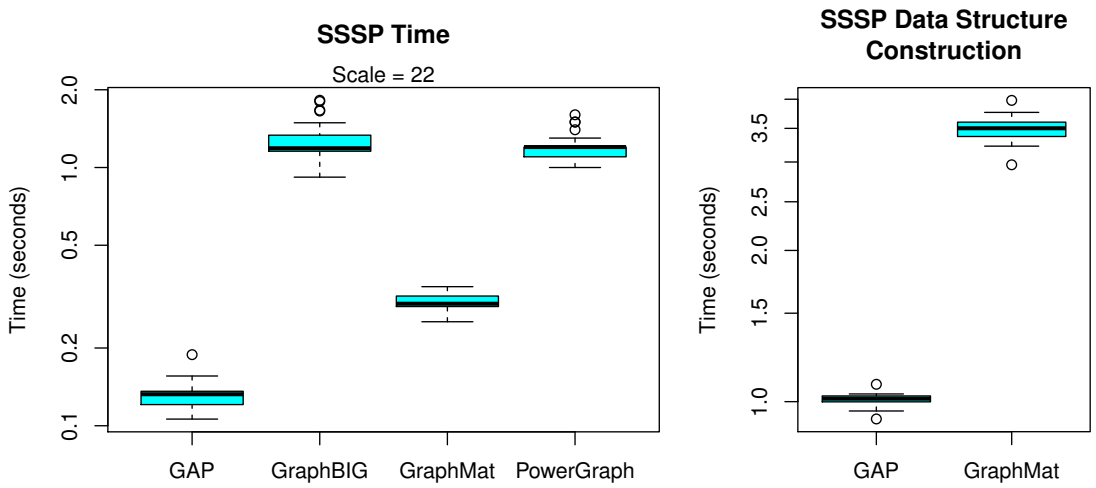


Figure 14. The y-axes is logarithmic. The left box plot shows the time to compute the SSSP starting at the same 32 roots as Figure 13. Both PowerGraph and GraphBIG construct their data structures at the same time as they read the file.

attempt to define similar stopping criteria for each system, but GraphMat executes until no vertices change rank; effectively its stopping criterion requires the ∞-norm to be less than machine epsilon. This results in an increased number of iterations. We adjusted the other systems to use $\|p_t - p_{t-1}\|_1 < \epsilon$ as the stopping criterion, where t is the iteration and n is the number of vertices. We use $\epsilon = 6 \times 10^{-8}$ because this value is approximately machine epsilon for single precision floating-point data to make the stopping criteria for all implementations as similar as possible to GraphMat, whose implementation cannot be easily use a different stopping criterion since it does not compute any $\|p_t - p_{t-1}\|$.

Moreover, the logarithmic scale in Figures 13 and 15 makes it difficult to see the variance in execution time among algorithms. For example, the standard deviation of the runtimes for PageRank (Figure 15) are less than half that of SSSP.

The difficulty in comparing iteration counts for PageRank underscores an important challenge for any comparison of graph processing systems. The

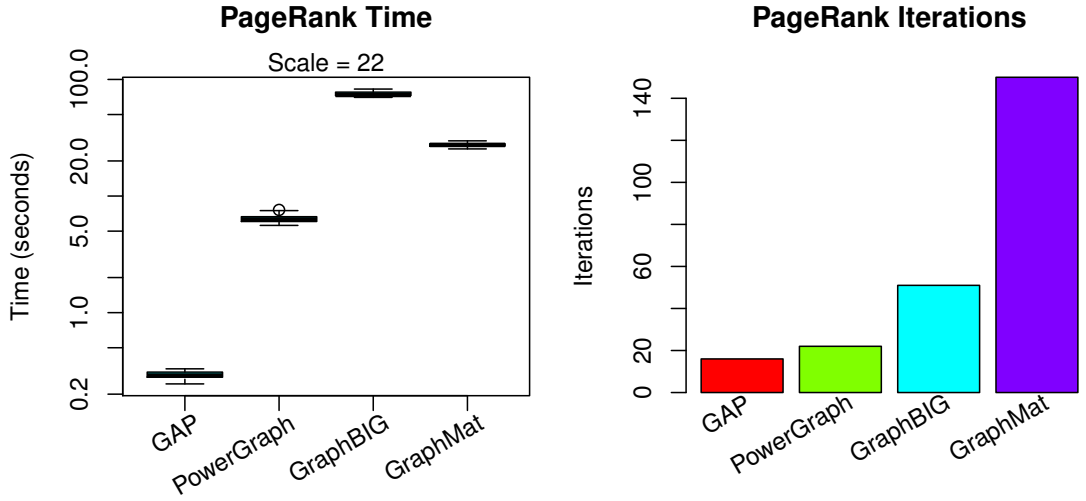


Figure 15. The y-axis is logarithmic only for the left figure. GraphMat continues to run until none of the vertices' ranks change. For the others, we use the stopping condition that the sum of the changes in the weights is no more than 6×10^{-8}, or approximately machine epsilon for single precision floating point numbers.

assumptions under which the various platforms operate can have a dramatic effect on the program. For example, the GAP Benchmark Suite can be recompiled to store weights as integers or floating-point values. This may affect performance in addition to execution time behavior in cases where weights like 0.2 are cast to 0. Similarly, how a graph is represented in the system (e.g., weighed or directed) may have performance and algorithmic implications but is not always readily apparent.

4.3.2 Scalability. The parallel speedup shown in Figure 16 is computed as T_1/T_n where T_1 is the sequential time and T_n is the execution time on n threads, with four trials per experiment.

Figure 16 generally shows poor scaling for this size problem, a challenge facing most parallel graph algorithms. In addition, 2^{23} vertices is a relatively small graph by today's standards and thus library designers focus on scalability for larger

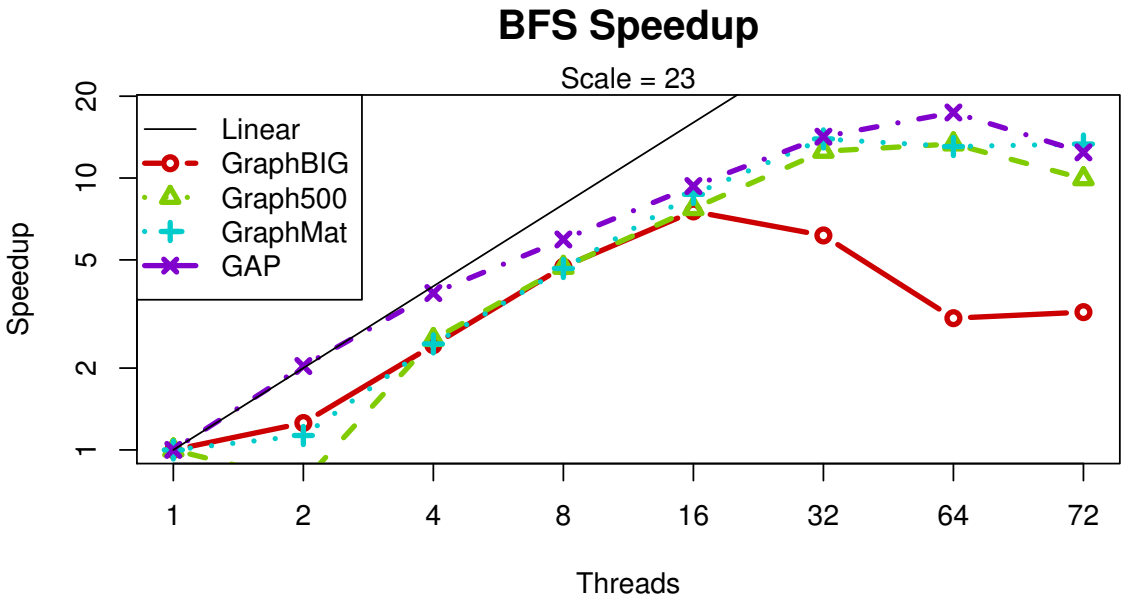


Figure 16. Speedup of BFS for a scale 23 graph. The black solid line represents ideal speedup. Both axes are logarithmic with an exception at 72 threads for readability.

graphs. Another limitation may be the ability for OpenMP to efficiently handle such a large number of threads per machine.

Overall, GAP is the most scalable with GraphMat close behind for larger threads and even slightly beating GAP at 72 threads. While care was taken to ensure there was no other load on the system when the experiments were performed, the efficiency for two threads is surprisingly low for the Graph500. Because the Graph500 spends a shorter amount of time executing in general (there is no file I/O and performs multiple BFS passes one right after another), it is more sensitive to spikes in CPU usage.

4.3.3 Real-World Datasets. Additionally, one may want to know the cost of performing additional computations on a graph once the data structures have been built. Figure 17 compiles average performance results for EPG*.

	BFS	CDLP	LCC	PR	SSSP	WCC
cit-Patents (3774768 vertices, 16518948 edges)	0.8 s	11.8 s	15.5 s	4.5 s	N/A	1.3 s
dota-league (61170 vertices, 50870313 edges)	1.1 s	3.9 s	1073.7 s	2.6 s	3.0 s	1.0 s
graph500-22 (2396781 vertices, 67108864 edges)	1.8 s	7.4 s	1802.7 s	4.7 s	N/A	2.4 s

Figure 17. Real world experiments using EPG*. PowerGraph does not provide BFS, Galois does not provide Triangle Counting.

Comparison of these two datasets in Figure 17 brings up some surprising variations in the data. For example, PowerGraph is the slowest for SSSP but not TC and PR. This could be because of the efficient edge-cut partitioning scheme in place on PowerGraph, which handles the high degree vertices more efficiently but does not pay off for the low amount of work done per vertex with SSSP. GraphBIG has the widest variation; the slowest for PageRank but the fastest for BFS, beating out even GAP which uses a more modern algorithm for BFS called Direction-optimizing BFS [5]. One reason for this lack of performance from GAP is our lack of tuning; we use the default parameterization of $\alpha = 15$ and $\beta = 18$, which may not be optimal for all graphs. One possible explanation for the poor performance of GraphMat compared is the overhead of the sparse matrix operations. These may pay off for larger datasets and we see good performance on the denser Dota-League dataset for GraphMat across all algorithms.

We use two real-world datasets for visualization: Dota-League and cit-Patents. Dota-league contains 61,670 vertices and 50,870,313 edges and models interactions between players in the online video game Defense of the Ancients. This was sourced from the Game Trace Archive[40] and is modified for a similar project,

Graphalytics[2]. This dataset is useful because it is both weighted and more dense than the usual real-world dataset with an average out-degree of 824.

Cit-Patents is is a widely-used network of citations from the National Bureau of Economic Research (NBER) and is less dense than Dota-League with 3,774,768 vertices and 16,518,948 edges. We stress that though these two datasets are presented, any network in the SNAP data format can be used in EPG*.

Table 5. Execution time, power, and energy use of BFS implementations (values averaged over computations for 32 roots) for a scale 22 Kronecker graph, executed on 32 threads. Sleeping energy refers to the power (in Watts) consumed during the `unistd` C `sleep` function, multiplied by the wallclock time. Essentially, this measures the energy that would have been consumed when the CPU and memory are (nearly) idle. "Increase over sleep" is the ratio of the first and third columns.

	GAP	Graph500	GraphBIG	GraphMat
Time (s)	0.01636	0.01884	1.600	1.424
Average Power per Root (W)	72.38	97.17	78.01	70.12
Energy per Root (J)	1.184	1.830	112.213	111.104
Sleeping Energy (J)	0.4046	0.4660	39.591	35.234
Increase over Sleep	2.926	3.928	2.834	3.153

4.3.4 Power and Energy Consumption. We use the Performance Application Programming Interface (PAPI) [10] to gain access to Intel's Running Average Power Limit (RAPL), which provides a set of hardware counters for measuring energy usage. We use PAPI to obtain average energy in nanojoules for a given time interval. We modify the source code for each project to time only the actual BFS computation and give a summary of the results in Table 5. In our case, the fastest code is also the most energy efficient, although with this level of granularity we could detect circumstances where one could make a tradeoff between energy and execution time.

[2]This dataset is available at `https://atlarge.ewi.tudelft.nl/graphalytics/`.

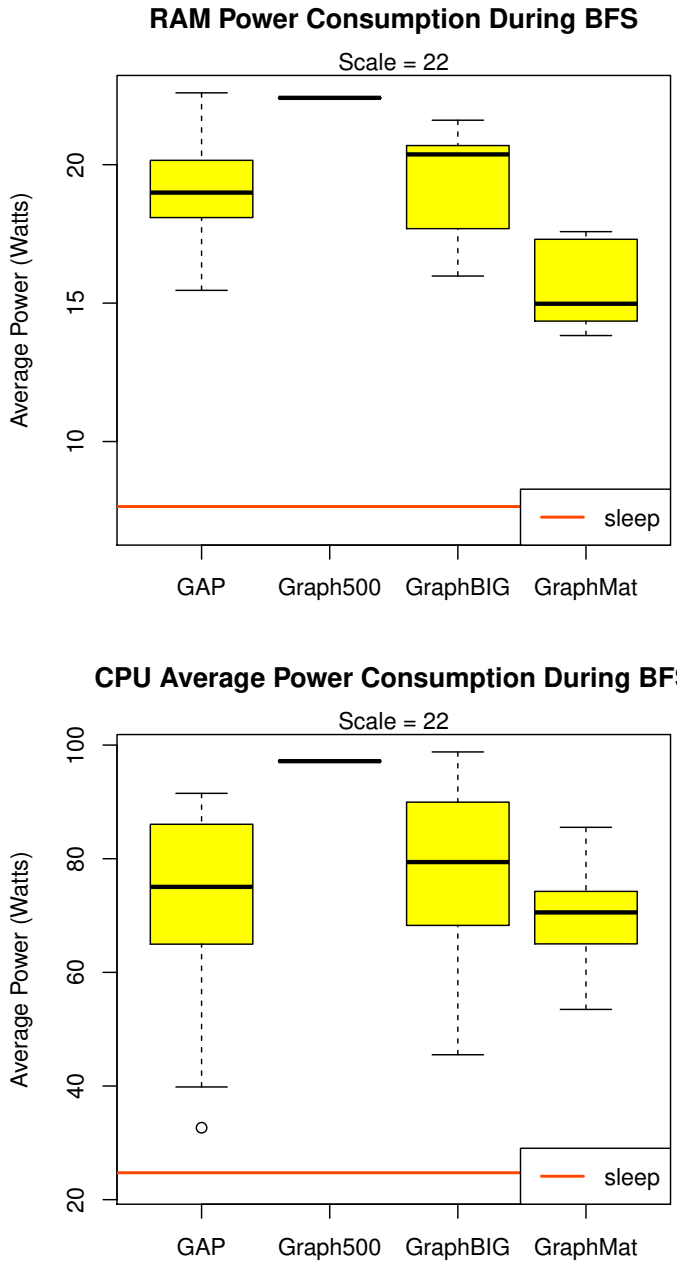


Figure 18. Similar to Fig. 13 we plot RAM and CPU Power Consumption for each of the 32 roots. Since the Graph500 runs multiple roots per execution, we only get a single data point. The baseline was computed by monitoring power consumption while a program containing only one call to the C `unistd` library function `sleep(10)` (ten seconds).

RAPL also allows the measurement of DRAM power, the results of which are shown in Figure **??**. The right plot describes the second row of Table 5 in more detail. We notice a smaller spread of RAM power consumption, but still a noticeable difference. GraphMat exhibits the lowest average RAM power consumption (but not the shortest execution time). This information can be useful when choosing an algorithm to use in limited power scenarios, where a slower algorithms that will not exceed the power cap is preferred to a faster one that may exceed it. Some drawbacks to measuring power via RAPL is it requires modification of the source code to measure the specific region and usually requires superuser permission.

4.4 Machine Learning Recommendations

Running experiments is by no means a trivial task. Graphs that are complex enough could be more time and resource consuming than the end-user could afford, especially in cases where the overall objective is to just identify a suitable graph package. Because EPG* automates the collection of large amounts of algorithm performance data, it enables the creation of machine learning models to determine which algorithm would perform best on new graphs without having to explicitly run the experiments. We refer to this performance dataset as the *learning set*, which is generated from the logs of all the experiments. It contains the graph properties and the execution times of the different implementations of each algorithm, and it serves as the input to our machine learning infrastructure for classifying graph algorithm implementations based on their performance.

4.4.1 Features of the graph. Along with the package and algorithm, the execution time of an experiment is directly dependent on the features of the graph. Hence these features would be serving as the attributes on which the

machine learning model would run on. The features of the real-world graphs that were ultimately chosen include the numbers of edges, vertices, and threads, the average clustering coefficient, the number of triangles, diameter (longest to shortest path ratio), the 90th percentile effective diameter, and the numbers of nodes and edges in largest strongly (SCC) and weakly connected components (WCC).

4.4.2 Preprocessing. The log-based data cannot be directly used as a learning dataset and must be preprocesed prior to training the ML model. The experiment set is a collection of experimental results for all graph datasets with multiple algorithms and packages. Because our objective is to find the best package for a given algorithm, the best experiment set was partitioned by algorithm in order to be trained individually, resulting in a separate model for each algorithm. Then the execution times of all data points were normalized to the z-score to get lower weights from the model. For similar reasons, nodes and edges in the largest SCC and WCCs were scaled to values between 0 and 1. Some of the real-world datasets were missing values for certain features, which we filled with the median values for that feature. Last, for categorical features such as the package and algorithm names, we mapped strings to natural numbers.

4.4.3 TEPS. The Graph500 uses a metric called traversed edges per second (TEPS) when reporting performance. We report TEPS because it normalizes execution time across various datasets. However, the work done per edge varies across algorithms (e.g., PageRank traverses each edge many times), so care must be taken to not mix these across algorithms. Hence, we consider each algorithm separately.

4.4.4 Machine Learning Models. We applied linear regression to produce a model of the execution time. The assumption was that if there was a

polynomial relation between the features and execution time, then linear regression would capture this relation and give us an accurate enough prediction of execution times for a given experiment. The learning set is split into a training set and test set in the ratio of 70%-30%. The linear model was then trained for each algorithm with the normalized and non-normalized version of training set. To further improve the accuracy, a ridging based linear regression model was incorporated in order to remove outliers which might influence the accuracy. The analysis of results are further discussed in the next section.

Although the execution time is influenced by the features of the graph, it is not linearly dependent on them. As a result, a regression based learning model didn't achieve the required benchmark of accuracy even though it produces an acceptable enough estimation of what the execution time could be for a package. But the major application of EPG* is to suggest the best package for a given dataset, algorithm and hardware configuration. Hence, we decided to build a classifier model.

The first classifier we implemented used logistic regression, since it is simple and works well for learning sets with little noise. Similar to regression-based models, the learning set was split into a training and test set with 70-30 ratio and made to run individually for each algorithm. Each data point in the learning set is tagged either "good" or "bad" based on the normalized execution time. A data point was tagged good if its TEPS was within an algorithm-specific threshold of the mean TEPS.

Although our initial goal was to implement a classification system, logistic regression did not achieve satisfactory results because, like linear regression, it encounters difficulties handling the many discrete features of our data and the

potentially nonlinear relationship between the graph properties and performance-based objective function.

For this reason, we chose a supervised decision tree-based model, namely Random Forest, to carry out the learning process as it gives higher accuracy without as much potential for over-fitting. Similar to logistic regression, the learning set split into 70%-30% training and testing subsets with each data point labeled "good" or "bad" based on the TEPS for each experiment. We built a separate classifier for each algorithm.

4.5 Machine Learning Results

In this section we present the results for three approaches to generating ML models for the performance of graph algorithms.

4.5.1 Regression. As mentioned above, we implemented the linear regression model for four different configurations, namely, with and without ridging for learning set that were both normalized and not normalized. Figure 19 shows how the results progressively improved with each configuration leading up to the best one being models that applied ridging to normalized (R+N) learning set. The R^2 value for this configuration is significantly better than others across all algorithms, peaking at 0.43 for SSSP.

Although R^2 is a common metric for evaluating a linear model, it is not sufficient for our purposes. This can be intuited by noticing our goal to predict runtime accurately; a linear model can have a low R^2 yet still predict runtime with reasonable accuracy. We measure Root Mean Square Deviation (RMSD) to measure the degree of difference in runtime (the middle plot in Figure 19). However, RMSD is hard to compare across algorithms; BFS has a much lower runtime than TC. Thus, we use and Normalized Root Mean Square Deviation

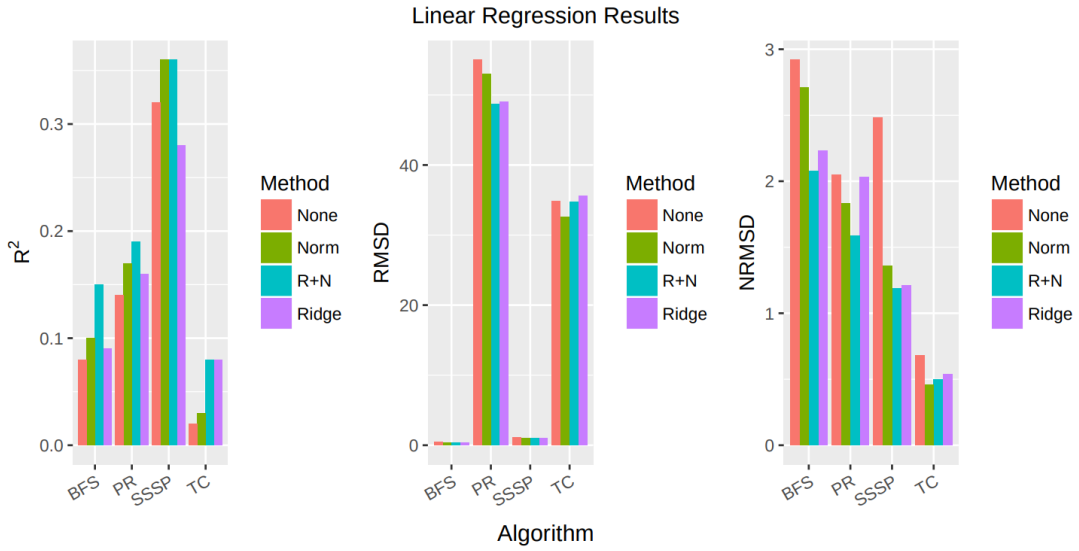


Figure 19. Linear regression with and without ridging and normalization

(NRMSD) (the right plot). This gives us a better idea how far the runtimes vary as a percentage of overall runtime. We note that our model predicts runtime within a factor of 2 in the worst case (BFS) and a factor of 0.5 in the best case (TC). These provide evidence that ridging and normalization (R+N) improves regression models.

4.5.2 Classification. The classification model is more accurate than linear regression in identifying the well-performing graph, algorithm, and hardware configuration. Table 6(a) shows the confusion matrix for logistic regression across all algorithms. Although the model achieves an overall accuracy of 64%, it categorizes all data points as "bad" for TC and "good" for PR, resulting in a high percentage of false negatives and false positives for TC and PR, respectively.

This motivated us to consider methods which work well even with a relatively small training set. Table 6(b) shows the confusion matrix of a random forest classifier across all algorithms. Random forest provides a significant

Table 6. Confusion matrices for all algorithms. The columns are predictions, the rows are observations.

(a) Logistic Regression

		good	bad
TC	good	0	49
	bad	0	113
BFS	good	133	9
	bad	91	4
PR	good	161	0
	bad	81	0
SSSP	good	27	85
	bad	17	113

(b) Random Forest

		good	bad
TC	good	43	1
	bad	0	118
BFS	good	110	0
	bad	0	147
PR	good	75	0
	bad	1	193
SSSP	good	51	11
	bad	9	198

improvement over logistic regression. Not only does it achieve an overall mean accuracy of 97% across all algorithms, the percentage of false positives and negatives is also significantly reduced. The model achieves perfect classification for BFS and a near perfect classification for TC and PR with the learning set described in Sec. 4.2.2.

Another area where this model excels is the low performance penalty for false positive predictions. With SSSP, the only algorithm with more than one false positive prediction, the mean TEPS for the nine false positives is slower than the mean TEPS by 0.03%. Intuitively, this implies that even when the model wrongly classifies a data point as good, it is only negligibly slower than average.

Similar to performance penalty, performance improvement for true positive predictions is also better for this model. Performance improvement determines by how much the proposed model outperforms a hypothetical model that picks packages at random. We can see from Table 7 that for each algorithm, the mean of normalized run times (TEPS) across all experiments is significantly lower than

the mean for experiments labeled as good. The improvement ranges from 7% to 700%. The improvement is least significant for PageRank because the different implementations performed very similarly. This is evidenced by the coefficient of variation (standard deviation/mean) of TEPS: for BFS it is 1.5 and PR is 0.21.

Table 7. TEPS for each algorithm. Improvement is computed as the mean TEPS of data labeled as good divided by the mean TEPS for all data. Larger TEPS indicates better performance.

Algorithm	Min	Max	Mean (all)	Mean (good)	Improvement (%)
TC	1.2e2	2.3e3	6.8e2	1.6e3	66.6
BFS	2.8e3	3.1e10	1.1e8	8.1e8	700.7
PR	7.6e1	1.1e5	4.4e3	4.7e3	6.8
SSSP	2.3e2	2.6e10	8.2e7	1.1e8	34.1

4.6 Related Work

Most performance analysis of graph processing systems come from the empirical results of each new library designer's publications. For example, GraphMat [138], PowerGraph [37], and GraphBIG [100] present a comparison of performance between their system and a selection of other software packages and in general any new approach will compare itself to existing implementations. For a collection of references to the original papers of each new library or framework, see [24]. Instead, we focus on related research which provide analysis of existing software.

Nai et al. [99] provide a detailed performance analysis using GraphBIG as their reference implementation. Their approach also compares GraphBIG to the GraphLab and Pregel execution models (gather-apply-scatter). Their analysis considers architectural performance measurements such as cache miss rates to measure bottlenecks and computational efficiency for a variety of datasets. While thorough, these analyses focus on covering a wide range of datasets and

appear difficult to apply to a new approach. For example, GraphMat reduces computation to sparse matrix operations which may not suffer from the same memory bottlenecks indicated in [99].

Research by Song et al. [130] and LeBeane et al. [68] considers the important problem of partitioning on heterogeneous architectures and provides profiling to reduce execution time, improve load balance, and reduce energy consumption. Our framework instead focuses on shared memory packages.

Beyond the aforementioned performance results, a number of reports focus on performance analysis not tied to a particular software package. Satish et al. [124] analyze the performance of several systems on datasets on the order of 30 billion edges and [80] uses six real-world datasets and focuses on the vertex-centric programming model. These implementations are hand tuned and provide recommendations for future improvements.

The most prominent example of a graph processing performance analysis tool not tied to a particular implementation is Graphalytics [12]. Similar to our work, Graphalytics automates the setup and execution of graph packages for performance analysis. Graphalyics relies on Apache Maven and Java to wrap the execution of each graph processing software package, which requires a nontrivial amount of Java code to add a new package. More importantly, the one-time data structure construction times are not separated from execution times in their results. This one-time setup cost is often higher than the algorithm's execution time, so the performance data may not apply to situations that involve several operations on a single graph.

With a plugin to Graphalytics called Granula [104], one can explicitly specify a performance model to analyze specific execution behavior, such as the

amount of communication or execution time of particular kernels of execution. While this requires in-depth knowledge of the source code and execution model and knowledge of the Granula API, it enables automated compilation, execution, and detailed performance analysis.

Beyond this plethora of performance data, the Graph Algorithm Platform Benchmark Suite (GAP) [6],[134],[112],GraphBIG [100], and the Graph500 [98] consider themselves reference implementations or benchmark suites with which other implementations can be compared. If no fewer than three software packages call themselves, "reference implementations," but which one are we to trust? We believe a single reference implementation cannot capture the complexities inherent in graph processing and provide performance information that fits many different use cases.

With respect to algorithm classification, while similar approaches have been applied in other areas (e.g., iterative sparse linear system solution algorithms), the work most similar to ours appears in [94], where machine learning models are used to predict the scalability of the Graph500, which has a very limited set of algorithms.

4.7 Future Work

Our choice of algorithms is based on the common subset of methods analyzed in prior work and implemented in the selected software packages. The standardization of graph algorithm building blocks (graph kernels) is being developed by the Graph BLAS Forum [89]). Once this standardization is finalized there will be more motivation from both library designers and performance analyzers to implement and profile each kernel.

Parallel SSSP and BFS algorithms contain parameters (Δ for SSSP and α and β for BFS), which can affect the overall performance. We plan to add heuristic parameter tuning as performed in [5] to the next iteration of our framework to exploit these algorithmic advances.

Algorithms such as triangle counting and betweenness centrality are widely implemented and relatively computationally intensive, but not supported by either Graphalytics nor EPG*; hence, these algorithms are suitable future targets for EPG*.

One overarching issue with these software packages is the speed at which they change. On one end of the spectrum, the code base of PowerGraph was made closed-source in 2015, resulting in over 400 forks with fixes and additions. On the other end, GraphMat's update to version 2.0 is in general incompatible with version 1.0, and so has not yet been adopted by Graphalytics or our framework. Tracking of most recent versions of packages is still not supported by EPG* or other frameworks.

With respect to power and energy profiling, while our current implementation supports measurements based on PAPI's interface to RAPL, which is only available on Intel platforms, the interface is simple and easy to adapt to other platforms in future without requiring PAPI support. In particular, fine-grained measurements provided through potentially available custom hardware [116] can be enabled through the same interface.

Ultimately, the automated aspect of EPG* allows us to generate a large amount of performance data, which can, in turn, be used to build machine learning models. These models can then be used to predict performance given a new dataset. For each new algorithm, objective function, and architecture, this involves

feature selection, model building, and validation, but once this is complete this will allow a model to be used in lieu of expensive experiments.

4.8 Conclusion

In this paper we presented EPG*, a multifaceted tool for studying the performance and power characteristics of parallel graph processing implementations. EPG* accomplishes this by ensuring graphs, stopping criterion, and execution environments are uniform across implementations. With this infrastructure, EPG* may be used to automatically run a large number of experiments, which in turn can be either analyzed with visualizations or used with the provided machine learning models to recommend optimal packages based on a graph's properties. While the comparison presented here can help one choose among alternatives for the selected packages and algorithms, the problem of selecting a graph processing framework for a given large-scale problem remains far from simple. Furthermore, increasing hardware heterogeneity demands performance analysis be easily repeatable on the target architecture. Because of this, our framework is designed to be easy to use and we make our code freely available at `https://github.com/HPCL/easy-parallel-graph` to encourage further experimentation.

Overall, the GAP Benchmark Suite is the best-performing system across all chosen datasets and in general the most scalable for the graphs used in this study (at most 2^{23} vertices). In addition, GAP is the most recent project. However, GAP is not meant for development of new applications unlike Powergraph, Galois, and Graphmat. Two more potentially important considerations are cost and portability: GraphMat requires the Intel compiler collection which may be cost-prohibitive for some users. 4 In addition to providing an easy-to-use framework for evaluating and

comparing graph algorithms, we have automated the generation of accurate ML classifiers that enable the selection of specific implementations best suited to the user's performance objective, hardware, and input problem characteristics. We have demonstrated its effectiveness in recommending algorithm implementations with the goal of maximizing the commonly used TEPS performance metric, resulting in 97% overall classification accuracy and mean performance improvement ranging between 7% and 700%.

CHAPTER V

LEARNABLE HIERARCHIES FOR GRAPH-BASED MULTI-AGENT REINFORCEMENT LEARNING ENVIRONMENTS

Graph-based Multi-Agent Reinforcement Learning (MARL) systems have shown significant promise in solving complex decision-making tasks involving multiple agents. However, as the number of agents increases and their interactions grow more intricate, designing effective value function estimators that capture both local and global coordination remains challenging. This paper introduces a novel hierarchical critic approach that leverages tiered Graph Neural Networks (GNNs) to enhance the performance of graph-based MARL. The proposed method leverages hierarchical GNNs to accommodate the diverse policy requirements of different agents operating in dynamic environments. A key innovation is a learned allocation mechanism that assigns each agent to its most beneficial tier, maximizing the overall joint reward. By dynamically aggregating multi-tier information, the hierarchical critic captures both fine-grained and coarse-grained dependencies, to enable a scalable and generalizable approach to policy learning. Experimental evaluations on the Multi-Agent Particle Environment (MPE) and Google Research Football (GRF) demonstrate that this tiered GNN-based hierarchical critic outperforms conventional flat GNN-based methods and other state-of-the-art MARL algorithms, achieving faster convergence, improved coordination, and stronger generalization to unseen scenarios. This work highlights the potential of hierarchical critics for advancing graph-based MARL environments with specialized agent roles.

5.1 Introduction

Graph-based Multi-Agent Reinforcement Learning (MARL) systems have shown significant promise in solving complex decision-making tasks involving multiple interacting agents, such as autonomous vehicle coordination, robot swarms, and strategic games [129, 64, 63]. By modeling multi-agent interactions as graphs, these systems can effectively capture the relationships among agents, enabling structured representation and efficient learning.

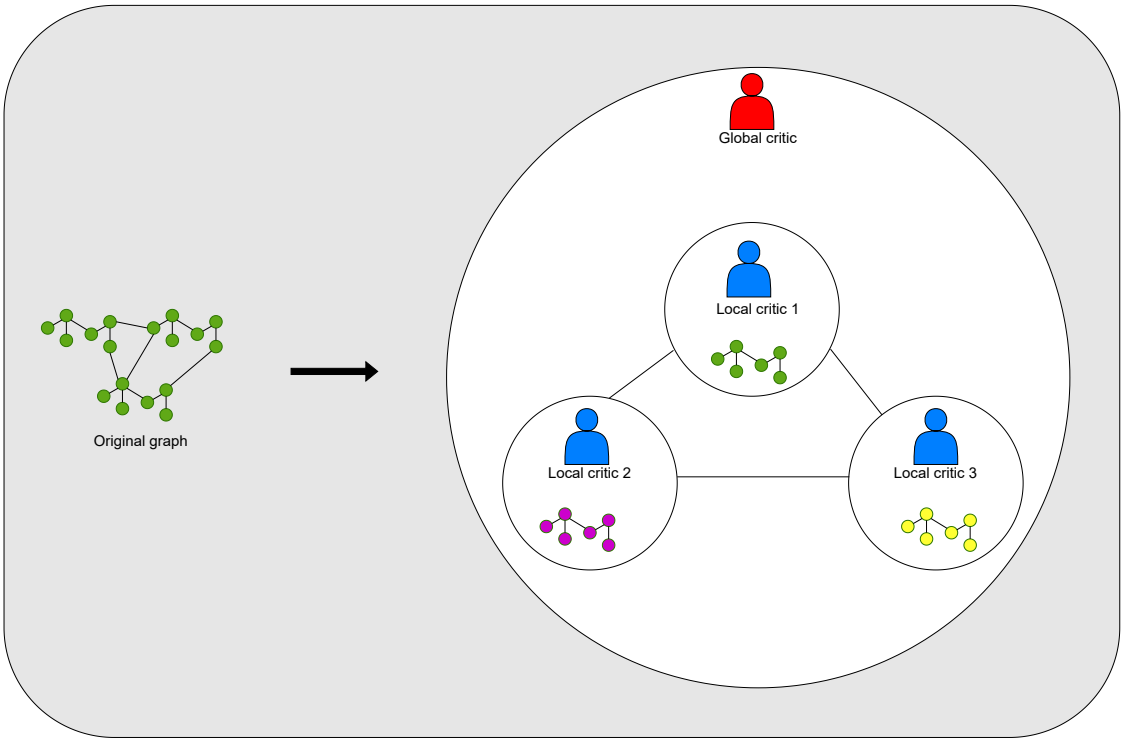


Figure 20. Partition of the original graph to tiered setup.

Despite these advantages, traditional MARL approaches often struggle to scale in dynamic environments where agent roles and interactions change frequently. To address these challenges, recent research has begun exploring hierarchical methods to improve agent coordination by explicitly capturing dependencies at multiple levels of abstraction [129, 64, 152]. However, integrating hierarchical critics

effectively with graph-based representations remains an open problem, particularly in environments characterized by dynamic agent interactions and specialized roles.

This paper presents a novel hierarchical critic framework, LearnHRL, that leverages tiered GNNs to improve scalability and coordination in graph-based MARL environments. Existing work, such as Multi-Agent Hierarchical Graph Attention Critic (MAHGAC) [73], has demonstrated that hierarchical methods can improve cooperation strategies in dynamic agent environments. However, these methods typically assume a fixed agent allocation to tiers, which limits their generalizability and introduces inductive bias. Our key insight is that the optimal allocation of agents to tiers should also be learned during training. To achieve this, we train our approach across various agent-tier combinations and employ a feedforward neural (FFN) classifier to predict the best allocation, ensuring adaptability and improved performance across different environments. The contributions of this work are summarized as follows:

- Multi-agent interactions are effectively modeled as graphs, where agents represent nodes, and their interactions form edges, enabling structured representation and efficient learning.
- A novel algorithm, LearnHRL, is introduced, employing hierarchical Graph Neural Networks (GNNs) with a learned tier-allocation mechanism.
- The efficacy and generalization of the proposed framework are validated through comprehensive experiments on the Google Research Football (GRF)[66] and Multi-Agent Particle Environment (MPE) benchmarks.

The remainder of this chapter is structured as follows: Section 5.2 discusses related works, Section 5.4 details the methodology, Section 5.6 presents the experimental setup and results.

5.2 Related Works

5.2.1 Multi-Agent Reinforcement Learning (MARL). Multi-Agent Reinforcement Learning (MARL) has gained significant attention for solving complex decision-making problems involving multiple interacting agents. Traditional MARL methods such as Independent Q-Learning [142] treat each agent as an independent learner, often leading to non-stationarity and convergence issues. Centralized learning approaches like MADDPG [78] and QMIX [115] attempt to address these challenges by leveraging shared information while maintaining decentralized execution. However, scalability remains a concern as the number of agents increases. Recent advancements focus on communication-based MARL methods [33] and value decomposition techniques [139] to improve coordination and reward allocation among agents.

5.2.2 Hierarchical Reinforcement Learning (HRL) Methods. Hierarchical Reinforcement Learning (HRL) introduces temporal abstraction by structuring policies into different levels of decision-making. Early HRL approaches such as the Options Framework [141] and MAXQ [23] decompose tasks into subtasks for better exploration efficiency. More recent works integrate HRL into MARL settings, such as Feudal Networks [146], which employ manager-worker hierarchies for decision-making. Hierarchical MARL approaches, including HSD [154] and HAMA [148], have demonstrated improved performance in large-scale multi-agent systems by structuring decision layers and reducing policy complexity.

5.2.3 Graph Neural Networks (GNNs) in RL.

Graph Neural Networks (GNNs) have emerged as powerful tools for modeling relational structures in multi-agent environments. Unlike conventional deep learning methods, GNNs effectively capture spatial and temporal dependencies among agents. Notable MARL applications leveraging GNNs include DGN [54], which enhances agent cooperation through graph convolution, and G2ANet [76], which uses graph attention for improved message passing. These methods enable more scalable and generalizable learning in graph-structured environments by dynamically encoding agent interactions.

5.2.4 Hierarchical GNN Methods.

Hierarchical Graph Neural Networks (HGNNs) extend standard GNNs by introducing multi-scale learning representations. This is particularly useful in MARL scenarios where agent interactions occur at varying levels of abstraction. Existing methods such as MAGNet [149] and MAHGAC [73] utilize hierarchical attention mechanisms to learn structured dependencies in multi-agent coordination. These approaches enhance message aggregation and improve policy learning efficiency in complex, high-dimensional environments. Our work builds upon these advances by introducing a hierarchical critic framework with learned tier allocations, optimizing multi-agent interactions dynamically.

These contributions collectively highlight the evolving landscape of MARL, HRL, and GNN-based learning, demonstrating the potential of hierarchical graph-based reinforcement learning in advancing multi-agent decision-making.

5.3 Preliminaries and Notation

In this section, we introduce the fundamental concepts and notations used in our hierarchical critic framework for graph-based multi-agent reinforcement learning (MARL).

5.3.1 Multi-Agent Reinforcement Learning (MARL) systems.

MARL systems are typically formulated as a Markov Game [**?**], defined by the tuple $\mathcal{M} = (\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \mathcal{O}, \gamma, N)$. where :

- $\mathcal{S}$ represents the state space, describing the global configuration of all agents in the environment.
- N is the number of agents indexed by $i \in \{1, 2, \ldots, N\}$.
- $\mathcal{A} = \{\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_N\}$ is the joint action space, where $\mathcal{A}_i$ denotes the action space of agent i.
- $\mathcal{P} : \mathcal{S} \times \mathcal{A} \to \mathcal{S}$ defines the transition probability function, mapping state-action pairs to the next state distribution.
- $\mathcal{R} = \{R_1, R_2, \ldots, R_N\}$ represents the reward function for each agent.
- $\mathcal{O} = \{O_1, O_2, \ldots, O_N\}$ is the observation function where each agent receives partial observations of the state $O_i : \mathcal{S} \to \mathcal{O}_i$.
- $\gamma \in [0, 1]$ is the discount factor that determines the importance of future rewards.

Each agent aims to maximize its expected cumulative reward $J_i = \mathbb{E}[\sum_{t=0}^{\infty} \gamma^t R_i^t]$ by learning an optimal policy $\pi_i(a_i|o_i)$, which maps observations to actions.

5.3.2 Graph-Based Multi-Agent Representation. We model the multi-agent system as a graph $G = (V, E)$, where:

- $V = \{v_1, v_2, \ldots, v_N\}$ represents the agents as nodes in the graph.
- $E \subseteq V \times V$ represents the edges, which define interactions or communication links between agents.
- Each node v_i has a feature vector h_i representing its local state or observation.
- Edge weights w_{ij} capture the interaction strength between connected agents.

A graph-based policy or value function can be expressed as:

$$Q(G, A) = f_\theta(h_1, h_2, ..., h_N, E), \tag{5.1}$$

where f_θ is a learnable function parameterized by θ using Graph Neural Networks (GNNs) to propagate information across agents.

5.3.3 Replay Buffer and Training Setup. We employ an experience replay buffer $\mathcal{D}$ to store agent transitions (s, a, r, s') and enable off-policy learning. The training process consists of the following steps:

- **Experience Collection**: Agents interact with the environment following their policies and store transitions in $\mathcal{D}$.
- **Mini-Batch Sampling**: A mini-batch of transitions is sampled from $\mathcal{D}$.
- **Graph-Based Value Computation**: The critic network processes the graph-structured state representation to evaluate the joint action-value function.

– **Policy Update**: The policy network updates using gradients computed from the critic.

– **Target Network Soft Updates**: A target critic network is updated periodically to stabilize training.

This setup allows efficient credit assignment and information sharing among agents while leveraging the structured representation of multi-agent interactions. The next section introduces our hierarchical critic framework, LearnHRL, which builds upon this foundation.

5.4 LearnHRL Architecture

This section breaks down the LearnHRL architecture starting with workflow and the following subsections formally define the mathematical formulation and present the algorithmic implementation.

5.4.1 Workflow and Overview. The LearnHRL framework consists of a structured workflow that integrates hierarchical critics, Graph Neural Networks (GNNs), and reinforcement learning-based tier allocation. The goal is to enable scalable and adaptive policy learning in multi-agent environments.

First, the environment is represented as a graph, where agents are nodes, and their interactions define the edges. At each timestep, agents observe their local state and pass it through a GNN to obtain structured embeddings. These embeddings are then used in two key processes: hierarchical tier assignment and policy execution.

For hierarchical tier assignment, a Feedforward Neural Network (FFN) dynamically allocates agents to different tiers based on their current observations. The tiers determine the granularity of decision-making for each agent. The policy

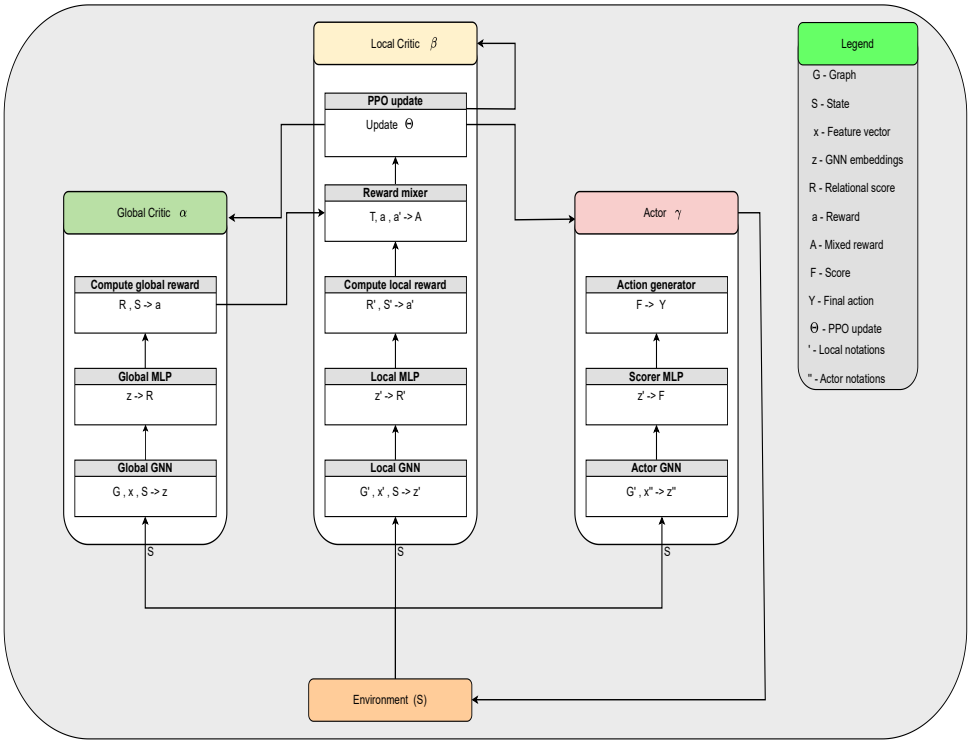


Figure 21. Workflow of LearnHRF architecture.

network then uses the GNN-processed embeddings and tier assignments to select actions.

During training, agent experiences are stored in a replay buffer, and the hierarchical critic network evaluates the Q-values of actions based on structured graph features. The critic is updated by minimizing the difference between predicted and target Q-values, while the actor network updates its policy through policy gradients. Target networks are softly updated to ensure stability. This workflow allows LearnHRL to dynamically adjust agent roles and optimize coordination in complex environments.

5.4.2 Mathematical Formulation. The equations used in the LearnHRL framework are fundamental to optimizing both hierarchical tier assignments and policy learning in a multi-agent reinforcement learning setting.

The **target Q-value equation** is based on the Bellman equation, which recursively estimates the value of a state-action pair by considering the immediate reward and the discounted future reward:

$$y_i = r_i + \gamma \mathbb{E} a' \sim \pi\theta'[Q^i_{\psi'}(G', A', T') - \alpha \log \pi_{\theta'}(a'|o')] \tag{5.2}$$

where r_i is the reward, γ is the discount factor, and $Q_{\psi'}$ is the target Q-value estimated by the hierarchical critic. The entropy term $\alpha \log \pi_{\theta'}(a'|o')$ encourages exploration by preventing premature convergence to a deterministic policy.

The **critic loss function** is defined as:

$$\mathcal{L}(\psi) = \sum_{i=1}^{N} \mathbb{E}(o, a, T, r, o') \sim \mathcal{D}[(Q\psi^i(o, a, T) - y_i)^2] \tag{5.3}$$

which minimizes the difference between the predicted Q-values and target Q-values, ensuring accurate value estimation.

The **policy gradient update** is computed as:

$$\nabla_{\theta_i} J(\pi_{\theta_i}) = \mathbb{E} o \sim \mathcal{D}, a \sim \pi[\nabla \theta_i \log \pi_{\theta_i}(a_i|o_i, T_i)(-\alpha \log \pi_{\theta_i}(a_i|o_i, T_i) + Q^i_\psi(o, a, T))] \tag{5.4}$$

where the actor network is updated based on the critic's evaluation of the expected return, ensuring policy improvement.

To stabilize training, we apply **soft target network updates**:

$$\psi' \leftarrow \tau\psi + (1 - \tau)\psi' \tag{5.5}$$

$$\theta' \leftarrow \tau\theta + (1 - \tau)\theta' \tag{5.6}$$

where τ is a small factor that prevents abrupt changes in the target networks, leading to more stable learning.

The **hierarchical reward function** ensures that agent tier assignments contribute positively to learning:

$$R_{\text{hierarchy}} = R_{\text{joint}} - \lambda \cdot \text{Penalty}(T) \tag{5.7}$$

where R_{joint} is the total multi-agent reward, and the penalty term discourages suboptimal tier assignments.

Finally, the **FFN classifier update** for hierarchical tier allocation is computed as:

$$\nabla_\phi J(\mathcal{H}) = \mathbb{E}[R_{\text{hierarchy}} \cdot \nabla_\phi \log \mathcal{H}_\phi(o_i)] \tag{5.8}$$

which ensures that the classifier learns to assign agents to tiers in a way that maximizes system-wide performance.

5.4.3 Algorithmic Implementation. Algorithm 3, defines the complete training process for the hierarchical reinforcement learning framework. It begins by initializing the actor networks, hierarchical critic networks, target networks, and a replay buffer. Each training episode starts with resetting the environment, where agents receive their initial observations. At each timestep, hierarchical tier allocations for the agents are determined using Algorithm 4. Given their allocated tiers, agents select actions based on their policies and interact with the environment. The system then observes the next states and rewards, and these transitions are stored in the replay buffer.

Algorithm 3 Training Procedure for LearnHRL

Initialize actor networks θ, hierarchical critic networks ψ, and Graph Neural Network (GNN) parameters η
Initialize target networks ψ', θ', and replay buffer $\mathcal{D}$
for each episode = 1 to M **do**
    Reset environment and initialize observations o_i^t for each agent i
    for each time step $t = 1$ to T **do**
        Compute hierarchical allocations T_i using Algorithm 4
        Compute graph embeddings $h = \text{GNN}(o)$
        Select actions $a_i \sim \pi_{\theta_i}(a_i|h_i, T_i)$
        Execute actions and observe next states o_i^{t+1}, rewards r_i^t
        Store transition $(o_i^t, a_i^t, T_i, r_i^t, o_i^{t+1})$ in replay buffer $\mathcal{D}$
    end for
    for each training step in batch size B **do**
        Sample minibatch from replay buffer $\mathcal{D}$
        Compute graph embeddings $h = \text{GNN}(o)$
        Compute Q-values $Q_\psi^i(G, A, T) = f_\psi(h, A, T)$
        Compute target Q-values
        Update hierarchical critic and actor using Equations (2) and (3)
        Soft-update target networks using Equations (4) and (5)
    end for
end for

During training, a mini-batch of transitions is sampled from the replay buffer, and Q-values are computed using the hierarchical critic network. The target Q-values are calculated using the Bellman equation, helping stabilize the learning process. The critic network is updated by minimizing the loss between predicted and target Q-values, while the actor policies are refined using policy gradients. The target networks for both the critic and actor are then softly updated to ensure stability during training. This algorithm effectively integrates hierarchical critics with reinforcement learning to improve scalability and adaptability in multi-agent environments.

Algorithm 4 is responsible for dynamically allocating agents to different hierarchical tiers. It starts by initializing the feedforward neural network (FFN)

Algorithm 4 Hierarchical Tier Assignment using Reinforcement Learning

Input: Observations o_i for each agent, FFN classifier $\mathcal{H}_\phi$, joint reward signal R_{joint}, penalty coefficient λ.
Output: Tier assignments T_i for each agent.
Initialize FFN classifier $\mathcal{H}_\phi$ with parameters ϕ
for each episode = 1 to M **do**
    Observe agent states o_i
    Compute tier probabilities: $P(T_i|o_i) = \mathcal{H}_\phi(o_i)$
    Sample tier assignment $T_i \sim P(T_i|o_i)$
    Compute hierarchical reward signal
    Update FFN classifier using policy gradient
end for

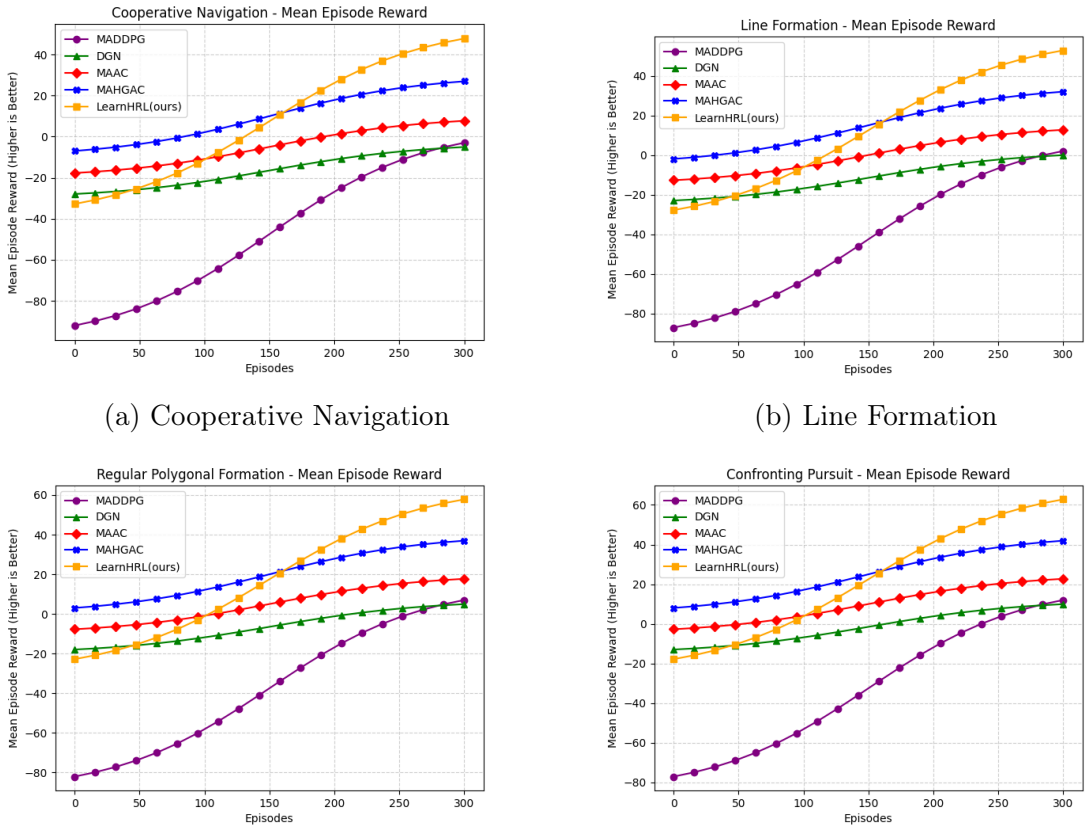
classifier, which predicts the most suitable tier for each agent based on its local observations. During each episode, agents observe their states and pass them through the FFN to obtain tier probabilities. A tier is then sampled based on these probabilities. The hierarchical reward signal is computed using the joint MARL reward while incorporating a penalty for suboptimal tier assignments to encourage diverse and effective agent allocations. The FFN classifier is updated using policy gradient methods, ensuring that it learns to assign agents to tiers in a way that maximizes overall system performance. By integrating this reinforcement learning-based tier assignment, the framework allows for adaptive and dynamic hierarchical coordination, eliminating the need for manually predefined agent allocations.

5.5 Experimental Evaluation

5.5.1 Benchmark Environments. To evaluate the performance of LearnHRL, we conduct experiments on two widely used multi-agent benchmarks: the Multi-Agent Particle Environment (MPE) and Google Research Football (GRF). These environments provide diverse challenges in cooperative and competitive multi-agent learning, making them suitable for assessing hierarchical critics in graph-based MARL.

5.6 Experiments on Multi-Agent Particle Environment (MPE)

5.6.1 Experimental Setup. MPE is a lightweight simulation platform featuring multiple cooperative and adversarial tasks. We utilize scenarios such as cooperative navigation and predator-prey to assess how LearnHRL enables agents to coordinate efficiently under dynamic interactions. Each agent is modeled as a node in a dynamic interaction graph, with edges representing communication or influence links. The experiments are run for a total of $1M$ environment steps with training performed using Adam optimizer.



(a) Cooperative Navigation

(b) Line Formation

(c) Regular Polygonal Formation

(d) Confronting Pursuit

Figure 22. Comparison of mean episode rewards for different multi-agent environments.

Baselines: We compare LearnHRL against several state-of-the-art MARL algorithms:

- **MADDPG** [78]: A multi-agent extension of DDPG that incorporates a centralized critic for training and decentralized execution, enabling agents to learn cooperative and competitive behaviors in mixed environments.
- **MAAC** [49]: A multi-agent actor-critic approach that utilizes attention mechanisms to selectively focus on important interactions, improving coordination and scalability in multi-agent systems.
- **HAMA** [148]: A hierarchical attention-based MARL algorithm that models structured agent dependencies, improving credit assignment and coordination in complex environments.
- **DGN** [54]: A graph convolutional reinforcement learning framework that leverages relational structures among agents using graph neural networks (GNNs) to improve policy learning and generalization.

5.6.2 Evaluation Metrics.

Performance in MPE is evaluated using:

- **Mean Episode Reward (MER)**: A commonly used metric in Multi-Agent Reinforcement Learning (MARL) to evaluate the performance of trained policies over time. It represents the average cumulative reward obtained per episode, which reflects the effectiveness of the agent(s) in achieving the given task. It is computed as :

$$MER = \frac{1}{N} \sum_{i=1}^{N} R_i \tag{5.9}$$

- **Convergence Speed**: Evaluates how quickly agents learn an optimal policy.

– **Coordination Success Rate**: Assesses how effectively agents collaborate to complete tasks.

5.6.3 Results and Discussion. Our model initially exhibits a lower Mean Episode Reward (MER) compared to the baseline methods during the early episodes. This can be attributed to the exploratory phase of learning, where the agents are still refining their policies. However, as training progresses, our model demonstrates a significantly higher rate of improvement in MER compared to the baselines. This suggests that the hierarchical learning structure enables faster adaptation to complex multi-agent interactions.

As shown in Figure 22, our model outperforms the baselines in the later stages of training, achieving superior long-term performance. The steeper increase in MER highlights the effectiveness of our method in optimizing agent cooperation and policy efficiency over extended training periods. Unlike traditional baselines, which may plateau early, our model continues to refine its decision-making, resulting in better generalization across different multi-agent environments.

These results confirm that our approach provides a more adaptive and scalable solution for multi-agent reinforcement learning, particularly in dynamic and high-dimensional settings. The ability to achieve faster learning rates and improved final performance underscores the benefits of our hierarchical reinforcement learning framework.

Acknowledgments: This work is supported by NSF OAC-SPX grants 1725755, 1725566, and 1725585 for the collaborative SANDY project, and OAC-CSSI grants 2104076, 2104078, 2104115 for the collaborative CANDY project.

CHAPTER VI

ASYNCHRONOUS PARADIGM FOR FORWARD PROPAGATION ON DISTRIBUTED GRAPHS

Graph Neural Networks (GNNs) have been widely adopted for modeling complex relationships in graph-structured data, with applications spanning various domains such as social networks, recommendation systems, and scientific computing [62, 41, 145]. However, traditional GNN architectures, including GraphSAGE and GAT, primarily rely on synchronous message passing, which can suffer from significant challenges such as **underreaching**—where nodes fail to aggregate sufficient information from distant neighbors—and **oversmoothing**—where node representations become indistinguishable due to excessive aggregation [72, 166, 120]. These challenges are further exacerbated in multi-node and distributed environments, where communication overhead can severely impact efficiency and scalability [14]. In this paper, we introduce **Graph Asynchronous Propagation (GAP)**, a novel asynchronous paradigm for message passing in GNNs designed to address these limitations. By leveraging **YGM**, an efficient distributed communication framework, we conduct multi-node graph experiments to evaluate the effectiveness of our approach. Our results show that GAP achieves improved accuracy across various distance metrics, outperforming synchronous baselines in distributed settings. The proposed method effectively mitigates oversmoothing and underreaching, leading to more robust node representations and faster convergence.

6.1 Introduction

Graph Neural Networks (GNNs) have emerged as powerful tools for learning representations from graph-structured data, enabling applications in

social networks, recommendation systems, and scientific computing [62, 41, 145]. Traditional GNN architectures, such as GraphSAGE and GAT, rely on synchronous message-passing mechanisms, where nodes aggregate information from their neighbors at fixed intervals. While effective, this approach introduces significant challenges, particularly in large-scale and distributed settings.

One major issue with synchronous message passing is **underreaching**, where nodes fail to incorporate information from distant neighbors due to the fixed number of propagation layers [72, 166]. Underreaching limits the receptive field of each node, making it difficult to capture long-range dependencies within the graph. This is particularly problematic in sparse graphs, where distant nodes may contain crucial contextual information that is inaccessible within a few hops. As a result, GNN models may struggle to generalize and produce suboptimal embeddings, affecting downstream tasks such as classification and link prediction.

Another critical challenge is **oversmoothing**, where repeated aggregation causes node embeddings to converge to similar values, leading to indistinguishable representations [120, 13]. This phenomenon arises when information is repeatedly mixed across layers, eventually making all node embeddings nearly identical. Oversmoothing reduces the expressiveness of GNNs, particularly in deep architectures, and has been shown to degrade performance on node classification tasks [171]. Mitigation strategies such as residual connections and layer normalization have been explored, but they often fail to address the root cause of excessive message mixing.

These problems are further exacerbated in multi-node distributed GNN training, where communication overhead and synchronization delays negatively impact performance [14]. The strict synchronization requirements of traditional

GNNs force all nodes to wait for messages from their neighbors before proceeding to the next step, creating bottlenecks in large-scale systems.

To address these limitations, we propose **Graph Asynchronous Propagation (GAP)**, a novel asynchronous message-passing paradigm for GNNs. Instead of synchronously aggregating neighbor information at each layer, GAP allows nodes to update their embeddings asynchronously, reducing communication bottlenecks and improving convergence speed. We conduct multi-node experiments using the **YGM** distributed communication framework [113] to evaluate the effectiveness of GAP. Our results show that GAP significantly improves accuracy across various distance metrics, mitigates oversmoothing, and enhances model robustness compared to synchronous approaches.

Our key contributions are as follows:

- We introduce **GAP**, an asynchronous message-passing paradigm for GNNs that mitigates underreaching and oversmoothing.
- We conduct multi-node experiments using **YGM** [113] to evaluate GAP's effectiveness in distributed environments, demonstrating improved performance over traditional synchronous approaches.

The remainder of this paper is organized as follows: Section **??** provides background and related work. Section **??** describes the GAP framework in detail. Section **??** presents experimental evaluations, followed by a discussion in Section **??**. Finally, Section **??** concludes the paper and outlines directions for future research.

6.2 Related Works

This section reviews prior research related to Graph Neural Networks (GNNs), message-passing mechanisms, and asynchronous computation. We

focus on three key areas: (1) advances in GNN architectures, (2) challenges of synchronous message passing, and (3) asynchronous and distributed GNN training.

6.2.1 Advancements in Graph Neural Networks. GNNs have become a fundamental tool for learning from graph-structured data, with early models such as Graph Convolutional Networks (GCN) [62], GraphSAGE [41], and Graph Attention Networks (GAT) [145] demonstrating the power of iterative message passing. These models aggregate information from neighboring nodes to learn node representations, enabling state-of-the-art performance in various domains such as social networks, molecular property prediction, and recommendation systems.

Despite these successes, traditional GNNs face limitations, particularly in deep architectures, where oversmoothing [120, 13] and underreaching [166] can hinder model expressiveness. Researchers have proposed techniques such as residual connections, normalization layers, and attention mechanisms to improve information flow, but these approaches often require careful hyperparameter tuning and do not fully eliminate the fundamental issues caused by synchronous message passing.

6.2.2 Challenges of Synchronous Message Passing. Most existing GNN architectures rely on synchronous message passing, where nodes aggregate information from their neighbors at fixed intervals. While effective in small-scale graphs, this approach suffers from two key challenges:

1) Oversmoothing: As layers increase, node representations tend to converge, reducing the network's ability to distinguish between different nodes [13, 166]. This phenomenon limits the depth of GNNs, forcing practitioners to rely on shallow architectures that may not capture long-range dependencies.

2) Underreaching: When GNNs have too few layers, nodes fail to aggregate meaningful information from distant neighbors [166]. This problem is particularly relevant in real-world applications where global structural information is essential for accurate predictions.

3) Scalability Issues in Distributed Environments: In large-scale and multi-node settings, synchronous message passing introduces significant communication overhead [14]. Since all nodes must wait for their neighbors before updating, synchronization delays become a major bottleneck, limiting the efficiency of distributed GNN training.

6.2.3 Asynchronous and Distributed GNN Training.

To overcome these limitations, researchers have explored asynchronous message-passing mechanisms, where node updates occur independently rather than in a synchronized manner. Asynchronous processing has been widely studied in distributed optimization and deep learning [14], but its application to GNNs remains an emerging area.

Recent work has explored decentralized and asynchronous GNN training methods to enhance scalability. Asynchronous training strategies reduce synchronization overhead and allow models to propagate information dynamically. For example, [14] proposed an asynchronous training framework to mitigate communication bottlenecks in large-scale graphs. Other studies have investigated methods such as DropEdge [120] to sparsify connectivity patterns and reduce oversmoothing effects. However, existing approaches often focus on modifications to synchronous training rather than fully leveraging an asynchronous communication model.

In this work, we propose Graph Asynchronous Propagation (GAP), a novel asynchronous message-passing paradigm for GNNs. GAP dynamically updates node embeddings, mitigating oversmoothing while improving efficiency in distributed environments. Our approach builds upon recent advancements in asynchronous training and distributed computing frameworks such as **YGM** [113], enabling large-scale multi-node experiments with improved scalability.

The next section introduces our methodology and provides a formal description of GAP.

6.3 Graph Asynchronous Propagation

6.3.1 Initialization Phase. Each node in the graph starts with its initial feature vector as its embedding and sets its pulse count ($\mathcal{P}_v$) to zero. The pulse count ensures that nodes operate within the correct propagation layer and do not process messages prematurely. Additionally, an aggregator buffer ($\mathcal{A}_v$) is initialized to temporarily store embeddings received from neighboring nodes. Since the GAP algorithm is asynchronous, all nodes begin execution simultaneously and proceed at their own pace.

6.3.2 Seek Phase – Requesting Neighbor Information. At each layer l, a node increments its pulse count ($\mathcal{P}_v = \mathcal{P}_v + 1$) to indicate that it is transitioning to the next layer of computation. It then sends a seek message containing this updated pulse count to all its one-hop neighbors ($u \in \mathcal{N}(v)$). This seek message serves as a request for updated embeddings from its neighbors. Since the algorithm operates asynchronously, multiple nodes can initiate seek messages at different times, depending on when they complete their previous layer's computation.

6.3.3 Send Phase – Responding to Requests. When a node u receives a seek message from its neighbor v, it checks its own pulse count ($\mathcal{P}_u$). If u is already at the same pulse count as v, it sends back a send message containing its latest embedding ($\mathbf{H}_u^{(l-1)}$) and its own pulse count ($\mathcal{P}_u$). If u has not yet reached this layer (i.e., its pulse count is lower than v's), it delays its response until it has processed all prior layers. This ensures that nodes only communicate embeddings from completed layers, preventing inconsistencies in the message-passing process.

6.3.4 Aggregation and Update Phase. As a node v receives send messages from its neighbors, it stores them in its aggregator buffer ($\mathcal{A}_v$). Since aggregation is permutation-invariant, it does not matter in which order the messages arrive. The node waits until it has received messages from all its neighbors before proceeding to update its own embedding. If some neighbors have not responded yet, the node remains idle until it collects all required messages. Once the messages are received, the node applies an aggregation function (e.g., sum, mean, or attention-based aggregation) to combine its neighbors' embeddings and updates its own embedding for the current layer ($\mathbf{H}_v^{(l)}$).

6.3.5 Progression to the Next Layer. After computing its updated embedding, the node sends a send message containing its newly computed embedding and updated pulse count to its neighbors. This process repeats for the next propagation layer. If a node receives a send message with a pulse count higher than its current pulse count, it buffers the message for future processing. This buffering mechanism ensures that nodes always process updates in the correct order.

6.3.6 Asynchronous Execution and Parallelism. Since different nodes may have different degrees (i.e., different numbers of neighbors), some nodes

complete their updates faster than others. This degree variance allows nodes from different subgraphs to overlap their computations, reducing overall idle time in distributed settings. For example, a node with fewer neighbors might finish a layer earlier and start processing the next layer while other nodes are still completing the previous layer. Unlike traditional synchronous GNN training, where all nodes must wait for the slowest node to finish before proceeding, GAP allows independent subgraph execution, significantly improving computational efficiency.

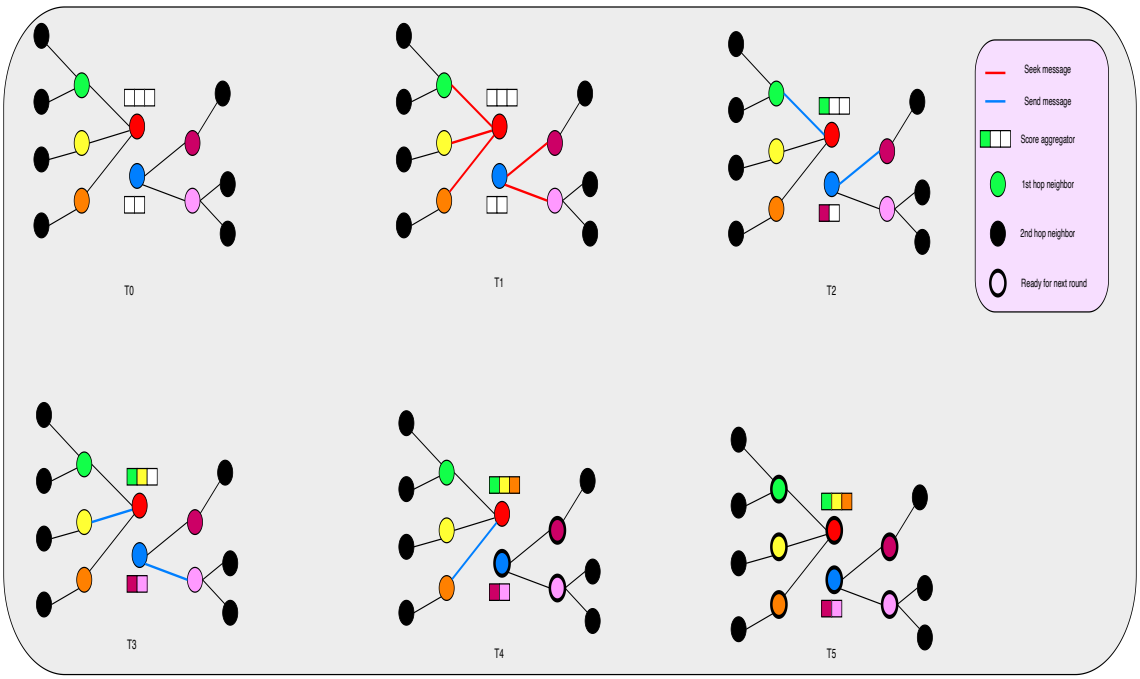


Figure 23. Workflow of async GNN communication from the red and blue nodes perspective. Both initiate seek messages to their neighbors in T1. Blue gets back all the neighbor messages at T4 and is ready for the next round before Red gets its messages at T5.

6.3.7 Illustration of GAP Workflow. Figure 23 presents a step-by-step visualization of the GAP algorithm in action. The figure demonstrates how two independent subgraphs, centered around the red (R) and blue (B) vertices, perform asynchronous message passing using the seek-send synchronization

mechanism across six timesteps (T_0 to T_5). Each colored vertex represents a node in the graph, while the black vertices denote two-hop neighbors that do not actively participate in the first layer of message passing.

At the beginning (T_0), both R and B have empty aggregators, meaning they have no collected embeddings from their neighbors. This signals that they need to initiate a seek message to request updated embeddings from their immediate one-hop neighbors.

At T_1, R sends seek messages to its one-hop neighbors: green (G), yellow (Y), and orange (O). Similarly, B sends seek messages to pink (P) and maroon (M). Each seek message contains the pulse count of the sender (which is 1 at this step), ensuring that all messages exchanged belong to the same propagation layer. Importantly, since R and B belong to **separate subgraphs**, their execution is entirely independent, meaning they can process updates asynchronously without waiting for each other.

At T_2, some of the neighbors start responding. G sends a send message back to R, and M sends one back to B. Each send message includes the neighbor's updated embedding along with its pulse count. However, at this point, R is still waiting for responses from Y and O, and B is still waiting for a response from P.

At T_3, additional neighbors respond: Y sends a send message to R, and P sends one to B. At this moment, B has received all the embeddings it requested from its neighbors, so it is now ready to compute the next-layer embedding for itself. However, R is still waiting for a response from O, meaning it cannot update its embedding just yet.

At T_4, B completes its layer 1 embedding update and can now proceed to the next layer. Meanwhile, R finally receives a send message from O, completing its

layer 1 aggregation. Since the GAP framework allows **independent execution**, B starts computing layer 2 embeddings while R is still finalizing layer 1. This **asynchronous overlapping of forward propagation** is a key benefit of GAP, allowing different subgraphs to progress at different speeds based on their neighborhood structures.

At T_5, both R and B have finished processing layer 1 and are ready to propagate messages for the next layer. Since B finished earlier, it already initiated its seek messages for layer 2 before R even completed its layer 1. This illustrates how GAP can significantly **reduce idle waiting times** by enabling different regions of the graph to progress asynchronously.

Key insights from Figure 23:

- **Asynchronous Execution:** Since R and B belong to separate subgraphs, they perform message passing at their own pace without waiting for each other.
- **Pulse Count Synchronization:** Each node only processes embeddings from the correct propagation layer, preventing premature updates.
- **Independent Forward Propagation:** Nodes with fewer neighbors (e.g., B) can complete their computations earlier and start the next layer before more connected nodes (e.g., R) finish.
- **Reduced Computation Bottlenecks:** In large-scale graphs, nodes with high degree variance (some having thousands of neighbors, others having just a few) can overlap computations effectively, leading to significant speedups.

Algorithm 5 GAP: Asynchronous Seek-Send Propagation

Input: Graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, node features $\mathbf{X}$, max layers L
Output: Node embeddings $\mathbf{H}$

Initialize embeddings $\mathbf{H}_v^{(0)} = \mathbf{X}_v$ for all $v \in \mathcal{V}$
Initialize pulse count $\mathcal{P}_v = 0$ for all $v \in \mathcal{V}$
for each node $v \in \mathcal{V}$ in parallel **do**
    Initialize empty aggregator buffer $\mathcal{A}_v$
    for $l = 1$ to L **do**
        $\mathcal{P}_v \leftarrow \mathcal{P}_v + 1$ ▷ Increment pulse count for layer
        Send **seek** message $(\mathcal{P}_v)$ to all $u \in \mathcal{N}(v)$
        Wait for **send** messages from neighbors
        while messages arrive from neighbors **do**
            Extract $(\mathcal{P}_u, \mathbf{H}_u^{(l-1)})$ from received send message
            if $\mathcal{P}_u = \mathcal{P}_v$ **then**
                $\mathcal{A}_v \leftarrow \mathcal{A}_v \cup \mathbf{H}_u^{(l-1)}$ ▷ Aggregate received embedding
            else
                Buffer message for later processing
            end if
        end while
        if received messages from all neighbors **then**
            $\mathbf{H}_v^{(l)} \leftarrow \text{AGGREGATE}(\mathcal{A}_v)$
            Send **send** message $(\mathcal{P}_v, \mathbf{H}_v^{(l)})$ to all $u \in \mathcal{N}(v)$
        else
            Wait until all expected messages arrive
        end if
    end for
end for

The GAP algorithm, as presented in Algorithm 5, operates through a seek-send synchronization mechanism that enables asynchronous message passing. This allows nodes to update embeddings independently while ensuring safe and correct message exchanges.

The algorithm begins by initializing node embeddings and pulse counts. Each node v is assigned an initial embedding $\mathbf{H}_v^{(0)}$, which is taken from the input features of the graph. A pulse count $\mathcal{P}_v$ is also initialized to zero, ensuring that each node tracks its current propagation layer. Additionally, an aggregator buffer

$\mathcal{A}_v$ is initialized as empty for each node. This buffer temporarily stores embeddings received from neighboring nodes before aggregation.

The main execution loop iterates over each node in parallel. Each node independently processes updates without waiting for other nodes. The outer loop iterates through the propagation layers from $l = 1$ to L, ensuring that messages are exchanged in a controlled layer-wise manner. At the beginning of each layer, the pulse count of the node is incremented by one ($\mathcal{P}_v \leftarrow \mathcal{P}_v + 1$), signaling that the node is progressing to a new computation step. Each node then broadcasts a seek message containing its pulse count to all its neighbors $u \in \mathcal{N}(v)$. This message serves as a request for updated embeddings from the neighbors.

After sending seek messages, the node must wait for send messages from its neighbors. The algorithm enters a waiting phase where it continuously listens for incoming messages. Whenever a send message arrives, the node extracts the pulse count $\mathcal{P}_u$ and the embedding $\mathbf{H}_u^{(l-1)}$ from the received message. Before processing the embedding, the node verifies whether the received pulse count matches its own. If $\mathcal{P}_u = \mathcal{P}_v$, it means that the sender is operating at the same layer, so the embedding is added to the aggregator buffer $\mathcal{A}_v$. If the pulse count does not match, the message is buffered for later processing to maintain correct layer synchronization.

A node can only proceed to update its embedding once it has received send messages from all of its neighbors. If messages are still missing, the node remains idle and waits. Once all required messages have arrived, the node computes its new embedding by applying an aggregation function over the collected embeddings. This function could be a simple sum, mean, or an attention mechanism, depending on the specific implementation of the GNN.

After computing the new embedding for the current layer, the node immediately sends a send message containing its updated embedding and pulse count to all of its neighbors. This ensures that its neighbors have the most recent information available for their own computations. The process then repeats for the next propagation layer until all L layers are completed.

This mechanism enables different nodes to operate asynchronously, meaning that nodes in different subgraphs can progress independently. Some nodes with lower degrees will finish their layers earlier, while higher-degree nodes may take longer to aggregate their embeddings before proceeding. This asynchrony allows for overlapping computations between different regions of the graph, reducing idle time and improving computational efficiency in large-scale GNN training.

6.4 Experimental Evaluation

In this section, we present our experimental evaluation of the proposed GAP algorithm. We assess its accuracy and runtime performance on the shortest path parity prediction task across various graph sizes and datasets. The experiments compare GAP against synchronous methods such as GIN [160] and GAT [145], which leverage batching for parallelism, and an asynchronous baseline, GWAC-iter [28]. This evaluation highlights GAP's unique ability to increase parallelism through multiple pivots, enabling efficient processing, particularly for larger graphs.

6.4.1 Experimental Setup and Datasets.

In our experiments, we evaluate the performance of the proposed GAP algorithm using the shortest path parity prediction task. In this task, each node in a graph predicts whether its shortest path distance from a designated source node is even or odd. This task tests the ability of a model to propagate information over various distances within the graph.

Four datasets were used in our evaluation. The first dataset is a Citation Network, where nodes represent scholarly articles and edges denote citation relationships. The second dataset, Proteins, is derived from protein–protein interaction networks, testing the model's capability to capture complex biological interactions. In addition, two synthetic datasets were generated using the RMAT model, namely RMAT25 and RMAT26. These synthetic graphs are designed with controlled structural properties such as degree distribution and average path length, allowing us to analyze the scalability and performance of GAP under varied conditions.

For each dataset, experiments were conducted over a range of graph sizes (100, 1000, 2000, 5000, and 10,000 nodes). This setup enables a comprehensive assessment of model accuracy and runtime performance as the complexity of the graph increases.

In the distributed setup for GAP, we explore parallelism by increasing the number of pivots. Pivots act as anchor points that partition the graph into smaller, independent subgraphs, which can then be processed in parallel. By increasing the number of pivots, the workload is more evenly distributed across multiple nodes, allowing for overlapping forward pass computations and reducing overall runtime. Our experiments in this distributed environment demonstrate that increasing the number of pivots enhances scalability, leading to improved efficiency in processing larger graphs.

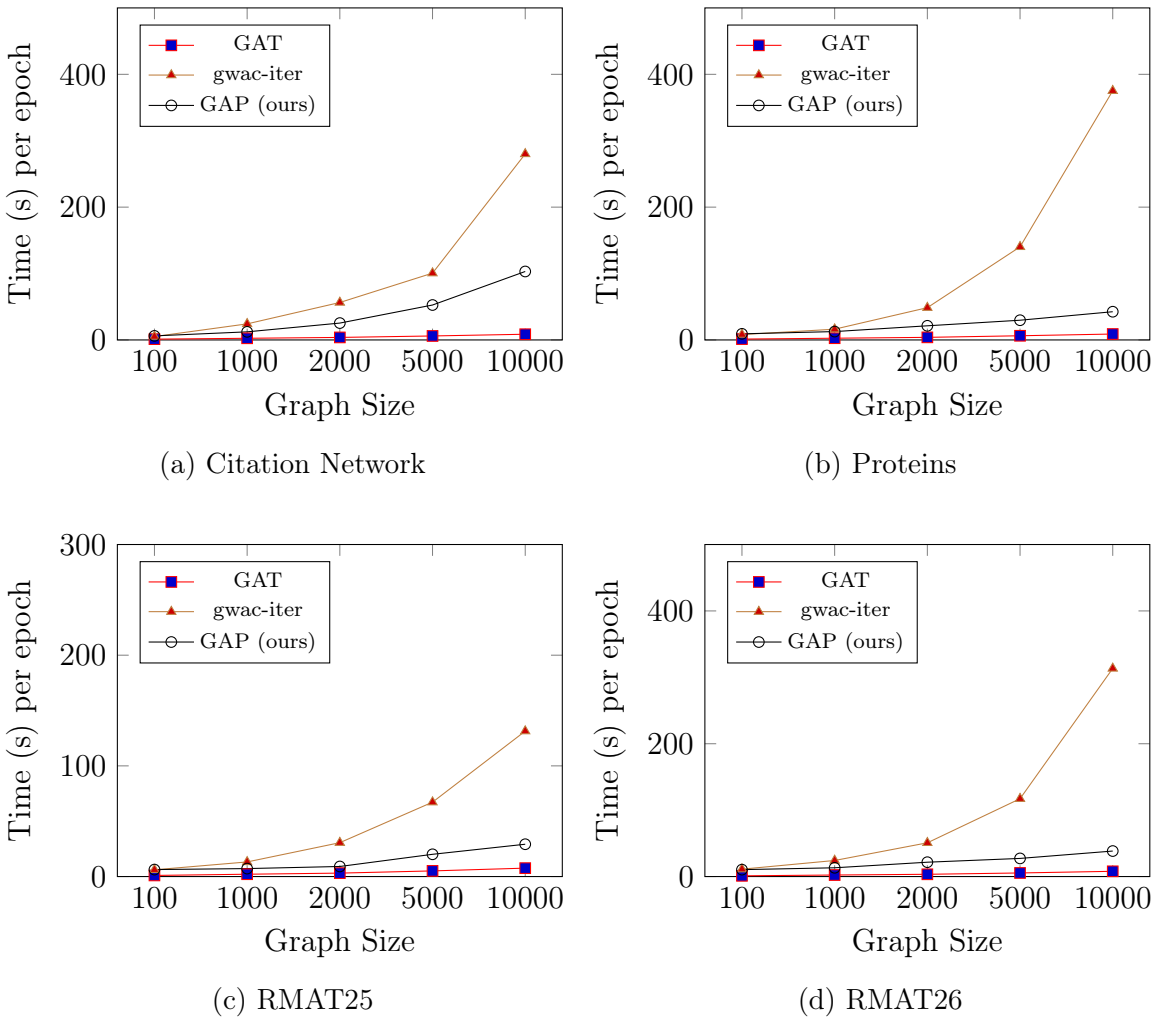
6.4.2 Accuracy metrics. Table 7 presents accuracy values for four methods—GIN, GAT, gwac-iter, and GAP (ours)—across graph sizes of 100, 1000, 2000, 5000, and 10,000 nodes. Each entry in the table indicates how well a model predicts whether the distance from a designated source node is even or odd, with

standard deviations reflecting variability across multiple runs. GIN and GAT, which employ synchronous message passing, achieve moderate performance on smaller graphs but experience noticeable declines as graph size increases. This drop suggests that synchronous models may encounter difficulties in propagating information over larger or more complex networks. In contrast, gwac-iter, an asynchronous baseline, maintains higher accuracy than GIN and GAT on both smaller and medium-scale graphs, indicating that asynchronous communication can mitigate some of the synchronization overhead and oversmoothing associated with traditional GNNs. GAP (ours) builds on the asynchronous paradigm and frequently attains higher accuracy than gwac-iter, particularly at the 1000- and 2000-node levels, while remaining competitive or superior at larger scales. These results imply that GAP's asynchronous message-passing mechanism effectively balances long-range information flow and local representation fidelity, preserving node-level distinctions even in larger graphs.

Table 8. Accuracy on the Shortest Path Parity Problem for Different Graph Sizes

Baseline	100 nodes	1000 nodes	2000 nodes	5000 nodes	10000 nodes
GIN	0.58 ± 0.01	0.50 ± 0.00	0.48 ± 0.01	0.43 ± 0.01	0.42 ± 0.02
GAT	0.57 ± 0.01	0.49 ± 0.00	0.49 ± 0.02	0.42 ± 0.02	0.41 ± 0.01
gwac-iter	0.96 ± 0.01	0.67 ± 0.01	0.55 ± 0.01	0.51 ± 0.01	0.50 ± 0.02
GAP (ours)	0.96 ± 0.02	0.88 ± 0.01	0.67 ± 0.01	0.56 ± 0.01	0.51 ± 0.02

6.4.3 Performance Evaluation. Figure 24 compares the runtime (in seconds per epoch) of four baselines—GAT, GIN, gwac-iter, and GAP (ours)—across graph sizes of 100, 1000, 2000, 5000, and 10,000 nodes. Subfigure 24a corresponds to the Citation Network dataset, while Subfigures 24b, 24c, and 24d represent Proteins, RMAT25, and RMAT26, respectively.



(a) Citation Network

(b) Proteins

(c) RMAT25

(d) RMAT26

Figure 24. Runtime (s) per epoch for four baselines (GAT, gwac-iter, and GAP) across equidistant x-axis labels for four datasets.

GIN and GAT employ synchronous message passing, which allows them to achieve parallelism by processing data in larger batches. This batching can lead to fast execution times on smaller or moderate graph sizes. However, as graph size increases, synchronous approaches often experience growing overhead from global synchronization, causing runtimes to escalate sharply at the higher node counts.

Gwac-iter and GAP, on the other hand, use asynchronous message passing and, therefore, cannot be batched in the same manner as synchronous methods.

Nevertheless, GAP incorporates an additional layer of parallelism through the concept of multiple pivots. By increasing the number of pivots, the graph is partitioned into more subgraphs that can be processed in parallel, thus mitigating the communication bottlenecks commonly associated with asynchronous execution. This design helps GAP maintain a more modest rise in runtime, even on large graphs. Consequently, these results suggest that while synchronous methods may excel on smaller datasets due to efficient batching, GAP's ability to exploit parallel pivots enables it to handle large-scale graph analytics more effectively.

CHAPTER VII

CONCLUSION AND FUTURE DIRECTIONS

This dissertation has explored the challenges and advancements in large-scale graph analytics, particularly at the intersection of high-performance computing (HPC) and machine learning. By focusing on novel algorithms and paradigms for distributed and asynchronous GNN-based solutions, this work has addressed critical limitations in scaling graph analytics to larger datasets and more complex domains. Below is a concise overview of each chapter's contributions:

Chapter I introduced the overarching motivation and objectives of the dissertation. It highlighted the increasing need for scalable graph processing techniques due to the rapid growth of data and posed key research questions on how HPC and machine learning could jointly address limitations such as oversmoothing, underreaching, and high communication overhead.

Chapter II reviewed the theoretical background and computational frameworks crucial for large-scale graph analytics. It examined state-of-the-art solutions for tasks like strongly connected components (SCC) detection, emphasizing the trade-offs between performance, memory usage, and communication overhead in distributed settings.

Chapter III presented an optimized algorithm for incremental SCC detection tailored for distributed environments. By storing intermediate results in a novel meta-graph structure, this approach reduced recomputation overhead when new edges or vertices were introduced. Experimental evaluations on both real and synthetic datasets demonstrated up to 2.8× performance improvement over existing methods.

Chapter IV proposed a model-based selection framework for choosing parallel graph processing packages. This framework used machine-learning techniques to predict execution time or classify packages based on their suitability for specific graph sizes and hardware configurations. The approach significantly reduced the manual trial-and-error process, enabling researchers to make more informed decisions with less computational overhead.

Chapter V explored hierarchical GNN architectures in the context of multi-agent reinforcement learning (MARL). A tiered critic model was introduced, leveraging hierarchical GNNs to capture both fine-grained and coarse-grained dependencies among agents. Experimental results on multi-agent environments showed faster convergence, better coordination, and stronger generalization than baseline MARL algorithms.

Chapter VI introduced an asynchronous GNN communication paradigm for multi-node systems. Building on concepts of oversmoothing, underreaching, and communication overhead, this chapter proposed the Graph Asynchronous Propagation (GAP) algorithm, which employs multiple pivots to enhance parallelism and mitigate synchronization delays. Through experiments on both real-world and synthetic datasets (e.g., Citation Network, Proteins, RMAT25, RMAT26), GAP demonstrated competitive accuracy and lower runtime compared to synchronous methods such as GIN [160] and GAT [145], and an asynchronous baseline, GWAC-iter [28].

Overall, this dissertation has shown that combining HPC strategies with advanced machine learning methods can significantly improve the scalability and efficiency of graph analytics. The asynchronous approaches and hierarchical designs

presented here are particularly promising for domains where graphs grow large and dynamically evolve.

Future Directions. Several avenues remain open for exploration. First, integrating dynamic load balancing techniques could further reduce communication bottlenecks in high-degree subgraphs. Second, extending the asynchronous GNN paradigm to handle fully streaming graph data, where edges and nodes arrive continuously, would enable real-time inference in rapidly changing environments. Third, applying the hierarchical GNN approach to more complex multi-agent tasks, such as autonomous driving or large-scale robotics, could reveal new insights into cooperative decision-making. By pursuing these directions, researchers and practitioners can continue to push the boundaries of high-performance graph analytics, delivering faster, more accurate, and more adaptable solutions to an ever-growing range of real-world challenges.

REFERENCES CITED

[1] Graph algorithm building blocks workshop. In David A. Bader, Aydın Buluç, John Gilbert, and Jeremy Kepner, editors, *30th IEEE International Parallel and Distributed Processing Symposium*, GABB '16, 2016.

[2] Stefano Allesina, Antonio Bodini, and Cristina Bondavalli. Ecological subsystems via graph theory: the role of strongly connected components. *Oikos*, 110(1):164–176, 2005.

[3] J Chris Anderson, Jan Lehnardt, and Noah Slater. *CouchDB: the definitive guide: time to relax*. " O'Reilly Media, Inc.", 2010.

[4] Muhammed Fatih Balin, Kaan Sancak, and Umit V Catalyurek. Mg-gcn: A scalable multi-gpu gcn training framework. In *Proceedings of the 51st International Conference on Parallel Processing*, pages 1–11, 2022.

[5] Scott Beamer, Krste Asanović, and David Patterson. Direction-optimizing breadth-first search. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 12:1–12:10, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[6] Scott Beamer, Krste Asanovic, and David A. Patterson. The GAP benchmark suite. *CoRR*, abs/1508.03619, 2015.

[7] Tal Ben-Nun, Michael Sutton, Sreepathi Pai, and Keshav Pingali. Groute: An asynchronous multi-gpu programming model for irregular computations. *ACM SIGPLAN Notices*, 52(8):235–248, 2017.

[8] Abraham Bernstein and Jürgen Cito. Incremental graph processing: A survey. *ACM Computing Surveys*, 54(2):1–36, 2021.

[9] Maciej Besta, Marc Fischer, Vasiliki Kalavri, Michael Kapralov, and Torsten Hoefler. Practice of streaming processing of dynamic graphs: Concepts, models, and systems. *IEEE Transactions on Parallel and Distributed Systems*, 2021.

[10] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci. A portable programming interface for performance evaluation on modern processors. *Int. J. High Perform. Comput. Appl.*, 14(3):189–204, August 2000.

[11] Aydın Buluç and John R Gilbert. The combinatorial blas: design, implementation, and applications. *The International Journal of High Performance Computing Applications*, 25(4):496–509, 2011.

[12] Mihai Capotă, Tim Hegeman, Alexandru Iosup, Arnau Prat-Pérez, Orri Erling, and Peter Boncz. Graphalytics: A big data benchmark for graph-processing platforms. In *Proceedings of the Graph Data Management Experiences and Systems*, GRADES '15, pages 7:1–7:6, New York, NY, USA, 2015. ACM.

[13] Dongkuan Chen, Yankai Lin, Wei Li, Ming Ding, Peng Liu, Jie Wang, and Jing Zhou. Measuring and relieving the over-smoothing problem for graph neural networks from the topological view. *Proceedings of the AAAI Conference on Artificial Intelligence*, 34(4):3438–3445, 2020.

[14] Jianfei Chen, Xiaolin Pan, Yichen Sun, Jie Tang, and Qiaozhu Mei. Asynchronous decentralized training for large-scale graphs. In *Advances in Neural Information Processing Systems*, 2020.

[15] Qun Chen, Song Bai, Zhanhuai Li, Zhiying Gou, Bo Suo, and Wei Pan. Graphhp: A hybrid platform for iterative graph processing. *arXiv preprint arXiv:1706.07221*, 2017.

[16] Raymond Cheng, Ji Hong, Aapo Kyrola, Youshan Miao, Xuetian Weng, Ming Wu, Fan Yang, Lidong Zhou, Feng Zhao, and Enhong Chen. Kineograph: Taking the pulse of a fast-changing and connected world. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, page 85–98, New York, NY, USA, 2012. Association for Computing Machinery.

[17] Unnikrishnan Cheramangalath, Rupesh Nasre, and YN Srikant. Falcon: A graph manipulation language for heterogeneous systems. *ACM Transactions on Architecture and Code Optimization (TACO)*, 12(4):1–27, 2015.

[18] Wei-Lin Chiang, Xuanqing Liu, Si Si, Yang Li, Samy Bengio, and Cho-Jui 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*, pages 257–266, 2019.

[19] YGM Contributors. Ygm: A high-performance communication library for c++, 2021.

[20] Chao Ma Da Zheng, Minjie Wang, and Jinjing 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.

[21] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[22] Laxman Dhulipala, Guy E Blelloch, and Julian Shun. Low-latency graph streaming using compressed purely-functional trees. In *Proceedings of the 40th ACM SIGPLAN conference on programming language design and implementation*, pages 918–934, 2019.

[23] Thomas G Dietterich. Hierarchical reinforcement learning with the maxq value function decomposition. In *International Conference on Machine Learning*, 2000.

[24] Niels Doekemeijer and Ana Lucia Varbanescu. A survey of parallel graph processing frameworks. Technical report, Delft University of Technology, 2014.

[25] David Ediger, Rob McColl, Jason Riedy, and David A Bader. Stinger: High performance data structure for streaming graphs. In *2012 IEEE Conference on High Performance Extreme Computing*, pages 1–5. IEEE, 2012.

[26] David Ediger, Rob McColl, Jason Riedy, and David A. Bader. Stinger: High performance data structure for streaming graphs. In *2012 IEEE Conference on High Performance Extreme Computing*, pages 1–5, 2012.

[27] Erich Elsen and Vishal Vaidyanathan. Vertexapi2–a vertex-program api for large graph computations on the gpu. 2014.

[28] Michael Faber, Alexander Gottschalk, Thomas Müller, and Stefan Schultze. Gwac: Accelerating graph neural networks with global weighted asynchronous communication. In *Proceedings of the 40th International Conference on Machine Learning*, volume 231, pages 123–132. PMLR, 2023.

[29] Wenfei Fan and Chao Tian. Incremental graph computations: Doable and undoable. *ACM Transactions on Database Systems (TODS)*, 47(2):1–44, 2022.

[30] Guoyao Feng, Xiao Meng, and Khaled Ammar. Distinger: A distributed graph data structure for massive dynamic graph processing. In *2015 IEEE International Conference on Big Data (Big Data)*, pages 1814–1822. IEEE, 2015.

[31] Matthias Fey and Jan Eric Lenssen. Fast graph representation learning with pytorch geometric. *arXiv preprint arXiv:1903.02428*, 2019.

[32] Lisa K Fleischer, Bruce Hendrickson, and Ali Pınar. On identifying strongly connected components in parallel. In *International Parallel and Distributed Processing Symposium*, pages 505–511. Springer, 2000.

[33] Jakob N Foerster, Yannis M Assael, Nando de Freitas, and Shimon Whiteson. Learning to communicate with deep multi-agent reinforcement learning. In *Advances in Neural Information Processing Systems*, 2016.

[34] Denis Foley and John Danskin. Ultra-performance pascal gpu and nvlink interconnect. *IEEE Micro*, 37(2):7–17, 2017.

[35] Joseph Gonzalez, Yucheng Low, Arthur Gretton, and Carlos Guestrin. Parallel gibbs sampling: From colored fields to thin junction trees. In *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, pages 324–332. JMLR Workshop and Conference Proceedings, 2011.

[36] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. {PowerGraph}: Distributed {Graph-Parallel} computation on natural graphs. In *10th USENIX symposium on operating systems design and implementation (OSDI 12)*, pages 17–30, 2012.

[37] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, ODSI '12, pages 17–30, Hollywood, CA, 2012. USENIX.

[38] Oded Green and David A Bader. custinger: Supporting dynamic graph algorithms for gpus. In *2016 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–6. IEEE, 2016.

[39] D. Gregor, N. Edmonds, B. Barrett, and A. Lumsdaine. The parallel boost graph library. `http://www.osl.iu.edu/research/pbgl`, 2005.

[40] Yong Guo and Alexandru Iosup. The game trace archive. In *Proceedings of the 11th Annual Workshop on Network and Systems Support for Games*, NetGames '12, pages 4:1–4:6, Piscataway, NJ, USA, 2012. IEEE Press.

[41] William L. Hamilton, Rex Ying, and Jure Leskovec. Inductive representation learning on large graphs. In *Proceedings of the 31st International Conference on Neural Information Processing Systems (NIPS)*, pages 1025–1035. Curran Associates Inc., 2017.

[42] Wei Han, Daniel Mawhirter, Bo Wu, and Matthew Buland. Graphie: Large-scale asynchronous graph traversals on just a gpu. In *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 233–245. IEEE, 2017.

[43] Muhammad Amber Hassaan, Martin Burtscher, and Keshav Pingali. Ordered vs. unordered: a comparison of parallelism and work-efficiency in irregular algorithms. *Acm Sigplan Notices*, 46(8):3–12, 2011.

[44] Bruce Hendrickson and Robert Leland. A multilevel algorithm for partitioning graphs. *Proceedings of the ACM/IEEE Conference on Supercomputing*, page 28, 1995.

[45] Changwan Hong, Aravind Sukumaran-Rajam, Jinsung Kim, and P Sadayappan. Multigraph: Efficient graph processing on gpus. In *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 27–40. IEEE, 2017.

[46] Sungpack Hong, Sang Kyun Kim, Tayo Oguntebi, and Kunle Olukotun. Accelerating cuda graph algorithms at maximum warp. *Acm Sigplan Notices*, 46(8):267–276, 2011.

[47] Sungpack Hong, Nicole C Rodia, and Kunle Olukotun. On fast parallel detection of strongly connected components (scc) in small-world graphs. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–11, 2013.

[48] Imranul Hoque and Indranil Gupta. Lfgraph: Simple and fast distributed graph analytics. In *Proceedings of the First ACM SIGOPS Conference on Timely Results in Operating Systems*, pages 1–17, 2013.

[49] Shariq Iqbal and Fei Sha. Actor-attention-critic for multi-agent reinforcement learning. In *International Conference on Machine Learning (ICML)*, 2019.

[50] Anand Padmanabha Iyer, Li Erran Li, Tathagata Das, and Ion Stoica. Time-evolving graph processing at scale. In *Proceedings of the fourth international workshop on graph data management experiences and systems*, pages 1–6, 2016.

[51] Anand Padmanabha Iyer, Qifan Pu, Kishan Patel, Joseph E Gonzalez, and Ion Stoica. {TEGRA}: Efficient {Ad-Hoc} analytics on evolving graphs. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 337–355, 2021.

[52] Yuede Ji, Hang Liu, and H. Howie Huang. ISpan: Parallel identification of strongly connected components with spanning trees. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*. IEEE Press, 2018.

[53] Zhihao Jia, Sina Lin, Mingyu Gao, Matei Zaharia, and Alex Aiken. Improving the accuracy, scalability, and performance of graph neural networks with roc. *Proceedings of Machine Learning and Systems*, 2:187–198, 2020.

[54] Jiechuan Jiang, Chen Dun, Tiejun Huang, and Zongqing Lu. Graph convolutional reinforcement learning. In *International Conference on Learning Representations (ICLR)*, 2020.

[55] Tim Kaler, Nickolas Stathas, Anne Ouyang, Alexandros-Stavros Iliopoulos, Tao Schardl, Charles E Leiserson, and Jie Chen. Accelerating training and inference of graph neural networks with fast sampling and pipelining. *Proceedings of Machine Learning and Systems*, 4:172–189, 2022.

[56] U Kang, Brendan Meeder, and Christos Faloutsos. Spectral analysis for billion-scale graphs: Discoveries and implementation. In *Advances in Knowledge Discovery and Data Mining*, volume 6635 of *Lecture Notes in Computer Science*, Berlin, 2011. Springer.

[57] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing*, 20(1):359–392, 1999.

[58] Arindam Khanda, Sanjukta Bhowmick, Xin Liang, and Sajal K. Das. Parallel vertex color update on large dynamic networks. In *2022 IEEE 29th International Conference on High Performance Computing, Data, and Analytics (HiPC)*, pages 115–124, 2022.

[59] Arindam Khanda, Sriram Srinivasan, Sanjukta Bhowmick, Boyana Norris, and Sajal K. Das. A parallel algorithm template for updating single-source shortest paths in large-scale dynamic networks. *IEEE Transactions on Parallel and Distributed Systems*, 33(4):929–940, 2022.

[60] Zuhair Khayyat, Karim Awara, Amani Alonazi, Hani Jamjoom, Dan Williams, and Panos Kalnis. Mizan: a system for dynamic load balancing in large-scale graph processing. In *Proceedings of the 8th ACM European conference on computer systems*, pages 169–182, 2013.

[61] Farzad Khorasani, Keval Vora, Rajiv Gupta, and Laxmi N Bhuyan. Cusha: vertex-centric graph processing on gpus. In *Proceedings of the 23rd international symposium on High-performance parallel and distributed computing*, pages 239–252, 2014.

[62] Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2017.

[63] Sanjana Krishnan, Ankesh Garg, and Roberto Calandra. Hierarchical multi-agent reinforcement learning with dynamic goals. In *International Conference on Learning Representations (ICLR)*, 2020.

[64] Tejas D. Kulkarni, Ardavan Saeedi, Simanta Gautam, and Samuel J. Gershman. Hierarchical reinforcement learning with deep transition models. In *Advances in Neural Information Processing Systems (NeurIPS)*, pages 3675–3683, 2016.

[65] Jérôme Kunegis. KONECT: The koblenz network collection. In *Proceedings of the 22Nd International Conference on World Wide Web*, WWW '13 Companion, pages 1343–1350, New York, NY, USA, 2013. ACM.

[66] Karol Kurach, Marcin Andrychowicz, Raphael Marinier, Piotr Raichuk, and Olivier Bachem. Google research football: A novel reinforcement learning environment. *arXiv preprint arXiv:2003.13350*, 2020.

[67] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. {GraphChi}:{Large-Scale} graph computation on just a {PC}. In *10th USENIX symposium on operating systems design and implementation (OSDI 12)*, pages 31–46, 2012.

[68] Michael LeBeane, Shuang Song, Reena Panda, Jee Ho Ryoo, and Lizy K. John. Data partitioning strategies for graph workloads on heterogeneous clusters. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '15, pages 56:1–56:12, New York, NY, USA, 2015. ACM.

[69] Jure Leskovec, Deepayan Chakrabarti, Jon Kleinberg, Christos Faloutsos, and Zoubin Ghahramani. Kronecker graphs: An approach to modeling networks. *J. Mach. Learn. Res.*, 11:985–1042, March 2010.

[70] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. `http://snap.stanford.edu/data`, June 2014.

[71] Guohui Li, Zhe Zhu, Zhang Cong, and Fumin Yang. Efficient decomposition of strongly connected components on gpus. volume 60, pages 1–10. Elsevier, 2014.

[72] Qimai Li, Zhichao Han, and Xiao-Ming Wu. Deeper insights into graph convolutional networks for semi-supervised learning. *Proceedings of the AAAI Conference on Artificial Intelligence*, 32(1), 2018.

[73] Tongyue Li, Dianxi Shi, Songchang Jin, Zhen Wang, Huanhuan Yang, and Yang Chen. Multi-agent hierarchical graph attention actor-critic reinforcement learning. *Entropy*, 27(4), 2024.

[74] Zhiqi Lin, Cheng Li, Youshan Miao, Yunxin Liu, and Yinlong Xu. Pagraph: Scaling gnn training on large graphs via computation-aware caching. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, pages 401–415, 2020.

[75] Yongchao Liu, Houyi Li, Guowei Zhang, Xintan Zeng, Yongyong Li, Bin Huang, Peng Zhang, Zhao Li, Xiaowei Zhu, Changhua He, et al. Graphtheta: A distributed graph neural network learning system with flexible training strategy. *arXiv preprint arXiv:2104.10569*, 2021.

[76] Yunpeng Liu, Bo Li, Yaodong Yang, Jianye Wang, and Jianye Hao. Multi-agent graph-attention communication and team disentangled actor-critic. In *International Conference on Autonomous Agents and MultiAgent Systems*, 2020.

[77] Yucheng Low, Joseph E Gonzalez, Aapo Kyrola, Danny Bickson, Carlos E Guestrin, and Joseph Hellerstein. Graphlab: A new framework for parallel machine learning. *arXiv preprint arXiv:1408.2041*, 2014.

[78] Ryan Lowe, Yi Wu, Aviv Tamar, Jean Harb, Pieter Abbeel, and Igor Mordatch. Multi-agent actor-critic for mixed cooperative-competitive environments. In *Advances in neural information processing systems*, 2017.

[79] Wei Lu, Jialu Li, James Cheng, David Wai-Lok Cheung, and Baihua Zheng. An experimental evaluation of graph processing frameworks on GPU. *Proceedings of the VLDB Endowment*, 9(12):972–983, 2016.

[80] Yi Lu, James Cheng, Da Yan, and Huanhuan Wu. Large-scale distributed graph computing systems: An experimental evaluation. *Proc. VLDB Endow.*, 8(3):281–292, November 2014.

[81] Nancy A Lynch. *Distributed algorithms*. Elsevier, 1996.

[82] Lingxiao Ma, Zhi Yang, Han Chen, Jilong Xue, and Yafei Dai. Garaph: Efficient {GPU-accelerated} graph processing on a single machine with balanced replication. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 195–207, 2017.

[83] Lingxiao Ma, Zhi Yang, Youshan Miao, Jilong Xue, Ming Wu, Lidong Zhou, and Yafei Dai. {NeuGraph}: Parallel deep neural network computation on large graphs. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 443–458, 2019.

[84] Peter Macko, Virendra J Marathe, Daniel W Margo, and Margo I Seltzer. Llama: Efficient graph analytics using large multiversioned arrays. In *2015 IEEE 31st International Conference on Data Engineering*, pages 363–374. IEEE, 2015.

[85] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 135–146, 2010.

[86] Mugilan Mariappan, Joanna Che, and Keval Vora. Dzig: Sparsity-aware incremental processing of streaming graphs. In *Proceedings of the Sixteenth European Conference on Computer Systems*, pages 83–98, 2021.

[87] Mugilan Mariappan and Keval Vora. Graphbolt: Dependency-driven synchronous processing of streaming graphs. In *Proceedings of the Fourteenth EuroSys Conference 2019*, pages 1–16, 2019.

[88] Claudio Martella, Roman Shaposhnik, Dionysios Logothetis, and Steve Harenberg. *Practical graph analytics with apache giraph*, volume 1. Springer, 2015.

[89] T. Mattson, D. Bader, J. Berry, A. Buluç, J. Dongarra, C. Faloutsos, J. Feo, J. Gilbert, J. Gonzalez, B. Hendrickson, J. Kepner, C. Leiserson, A. Lumsdaine, D. Padua, S. Poole, S. Reinhardt, M. Stonebraker, S. Wallach, and A. Yoo. Standards for graph algorithm primitives. In *2013 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–2, Sept 2013.

[90] Rahul Mazumder and Trevor Hastie. Exact covariance thresholding into connected components for large-scale graphical lasso. *The Journal of Machine Learning Research*, 13(1):781–794, 2012.

[91] Robert McColl, Oded Green, and David A. Bader. A new parallel algorithm for connected components in dynamic graphs. In *20th Annual International Conference on High Performance Computing*, pages 246–255. IEEE, 2013.

[92] Frank D McSherry, Rebecca Isaacs, Michael A Isard, and Derek G Murray. Differential dataflow, October 20 2015. US Patent 9,165,035.

[93] Vasimuddin Md, Sanchit Misra, Guixiang Ma, Ramanarayan Mohanty, Evangelos Georganas, Alexander Heinecke, Dhiraj Kalamkar, Nesreen K Ahmed, and Sasikanth 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*, pages 1–14, 2021.

[94] S. Medya, L. Cherkasova, and A. Singh. Predictive modeling and scalability analysis for large graph analytics. In *2017 IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*, pages 63–71, May 2017.

[95] Duane Merrill, Michael Garland, and Andrew Grimshaw. Scalable gpu graph traversal. *ACM Sigplan Notices*, 47(8):117–128, 2012.

[96] Jayanta Mondal and Amol Deshpande. Managing large dynamic graphs efficiently. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 145–156, 2012.

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

[98] Richard C. Murphy, Kyle B. Wheeler, Brian W. Barrett, and James A. Arg. Introducing the graph500. Technical report, Cray User's Group, 2010.

[99] Lifeng Nai, Yinglong Xia, Ilie G. Tanase, and Hyesoon Kim. Exploring big graph computing — an empirical study from architectural perspective. *Journal of Parallel and Distributed Computing*, August 2016.

[100] Lifeng Nai, Yinglong Xia, Ilie G. Tanase, Hyesoon Kim, and Ching-Yung Lin. GraphBIG: Understanding graph computing in the context of industrial solutions. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '15, pages 69:1–69:12, New York, NY, USA, 2015. ACM.

[101] Rupesh Nasre, Martin Burtscher, and Keshav Pingali. Atomic-free irregular computations on gpus. In *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units*, pages 96–107, 2013.

[102] Rupesh Nasre, Martin Burtscher, and Keshav Pingali. Data-driven versus topology-driven irregular computations on gpus. In *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, pages 463–474. IEEE, 2013.

[103] Rupesh Nasre, Martin Burtscher, and Keshav Pingali. Morph algorithms on gpus. In *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 147–156, 2013.

[104] Wing Lung Ngai. Fine-grained performance evaluation of large-scale graph processing systems. Master's thesis, Delft University of Technology, 2015.

[105] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. A lightweight infrastructure for graph analytics. In *Proceedings of the twenty-fourth ACM symposium on operating systems principles*, pages 456–471, 2013.

[106] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. A lightweight infrastructure for graph analytics. In *Proceedings of ACM Symposium on Operating Systems Principles*, SOSP '13, pages 456–471, 2013.

[107] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bring order to the web. Technical report, Technical report, stanford University, 1998.

[108] Sreepathi Pai and Keshav 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*, pages 1–19, 2016.

[109] Yuechao Pan, Yangzihao Wang, Yuduo Wu, Carl Yang, and John D Owens. Multi-gpu graph analytics. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 479–490. IEEE, 2017.

[110] Georgios A Pavlopoulos, Maria Secrier, Charalampos N Moschopoulos, Theodoros G Soldatos, Sophia Kossida, Jan Aerts, Reinhard Schneider, and Pantelis G Bagos. Using graph theory to analyze biological networks. In *BioData Mining*, Bethesda, MD, 2011. PubMed Central.

[111] Keshav Pingali, Donald Nguyen, Milind Kulkarni, Martin Burtscher, M Amber Hassaan, Rashid Kaleem, Tsung-Hsien Lee, Andrew Lenharth, Roman Manevich, Mario Méndez-Lojo, et al. The tao of parallelism in algorithms. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, pages 12–25, 2011.

[112] Samuel D. Pollard, Sudharshan Srinivasan, and Boyana Norris. A performance and recommendation system for parallel graph processing implementations: Work-in-progress. In *Companion of the 2019 ACM/SPEC International Conference on Performance Engineering*, page 25–28. Association for Computing Machinery, 2019.

[113] Benjamin Priest, Trevor Steil, Geoffrey Sanders, and Roger Pearce. You've got mail (ygm): Building missing asynchronous communication primitives. In *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 221–230. IEEE, 2019.

[114] Morteza Ramezani, Weilin Cong, Mehrdad Mahdavi, Mahmut T Kandemir, and Anand Sivasubramaniam. Learn locally, correct globally: A distributed algorithm for training graph neural networks. *arXiv preprint arXiv:2111.08202*, 2021.

[115] Tabish Rashid, Mikayel Samvelyan, Christian Schroeder De Witt, Gregory Farquhar, Jakob Foerster, and Shimon Whiteson. Qmix: Monotonic value function factorisation for deep multi-agent reinforcement learning. In *International Conference on Machine Learning*, 2018.

[116] Mohammad Rashti, Gerald Sabin, and Boyana Norris. Power and energy analysis and modeling of high performance computing systems using WattProf. In *Proceedings of the 2015 IEEE National Aerospace and Electronics Conference (NAECON)*, July 2015.

[117] John H Reif. Depth-first search is inherently sequential. *Information Processing Letters*, 20(5):229–234, 1985.

[118] Chenghui Ren, Eric Lo, Ben Kao, Xinjie Zhu, and Reynold Cheng. On querying historical evolving graph sequences. *Proceedings of the VLDB Endowment*, 4(11):726–737, 2011.

[119] Marko A. Rodriguez. The gremlin graph traversal machine and language. *CoRR*, abs/1508.03843, 2015.

[120] Yu Rong, Wenbing Huang, Tingyang Xu, and Junzhou Huang. Dropedge: Towards deep graph convolutional networks on node classification. *International Conference on Learning Representations (ICLR)*, 2020.

[121] Ryan A Rossi, Brian Gallagher, Jennifer Neville, and Keith Henderson. Modeling dynamic behavior in large evolving graphs. In *Proceedings of the sixth ACM international conference on Web search and data mining*, pages 667–676, 2013.

[122] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. X-stream: Edge-centric graph processing using streaming partitions. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 472–488, 2013.

[123] Scott Sallinen, Roger Pearce, and Matei Ripeanu. Incremental graph processing for on-line analytics. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1007–1018, 2019.

[124] Nadathur Satish, Narayanan Sundaram, Md. Mostofa Ali Patwary, Jiwon Seo, Jongsoo Park, M. Amber Hassaan, Shubho Sengupta, Zhaoming Yin, and Pradeep Dubey. Navigating the maze of graph analytics frameworks using massive graph datasets. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 979–990, New York, NY, USA, 2014. ACM.

[125] Julian Shun and Guy 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*, pages 135–146, 2013.

[126] Julian Shun and Guy E. Blelloch. Ligra: A lightweight graph processing framework for shared memory. *SIGPLAN Not.*, 48(8):135–146, February 2013.

[127] George M Slota, Sivasankaran Rajamanickam, and Kamesh Madduri. Bfs and coloring-based parallel algorithms for strongly connected components and related problems. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, pages 550–559. IEEE, 2014.

[128] George M Slota, Sivasankaran Rajamanickam, and Kamesh Madduri. High-performance graph analytics on manycore processors. In *2015 IEEE International Parallel and Distributed Processing Symposium*, pages 17–27. IEEE, 2015.

[129] Kyunghwan Son, Daewoo Kim, Wan Ju Kang, David Earl Hostallero, and Yung Yi. Qtran: Learning to factorize with transformation for cooperative multi-agent reinforcement learning. In *International Conference on Machine Learning (ICML)*, pages 5887–5896, 2019.

[130] S. Song, M. Li, X. Zheng, M. LeBeane, J. H. Ryoo, R. Panda, A. Gerstlauer, and L. K. John. Proxy-guided load balancing of graph processing workloads on heterogeneous clusters. In *2016 45th International Conference on Parallel Processing (ICPP)*, pages 77–86, Aug 2016.

[131] S. Srinivasan, A. Khanda, S. Srinivasan, A. Pandey, S. K. Das, S. Bhowmick, and B. Norris. A distributed algorithm for identifying strongly connected components on incremental graphs. In *2023 IEEE 35th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pages 109–118, 2023.

[132] Sriram Srinivasan, Samuel D. Pollard, Boyana Norris, Sajal K. Das, and Sanjukta Bhowmick. A shared-memory algorithm for updating tree-based properties of large dynamic networks. *IEEE Transactions on Big Data*, 8(2):302–317, 2022.

[133] Sriram Srinivasan, Sara Riazi, Boyana Norris, Sajal K. Das, and Sanjukta Bhowmick. A shared-memory parallel algorithm for updating single-source shortest paths in large dynamic networks. In *2018 IEEE 25th International Conference on High Performance Computing (HiPC)*, pages 245–254, 2018.

[134] Sudharshan Srinivasan. *Model-Based Algorithm Selection Techniques*. PhD thesis, University of Oregon, 2019. Available from ProQuest Dissertations and Theses. Last updated: 2023-06-21.

[135] Sudharshan Srinivasan, Arindam Khanda, Sriram Srinivasan, Aashish Pandey, Sajal K. Das, Sanjukta Bhowmick, and Boyana Norris. A distributed algorithm for identifying strongly connected components on incremental graphs. In *2023 IEEE 35th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pages 109–118. IEEE, 2023.

[136] Trevor Steil, Tahsin Reza, Keita Iwabuchi, Benjamin W Priest, Geoffrey Sanders, and Roger Pearce. Tripoll: computing surveys of triangles in massive-scale temporal graphs with metadata. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12, 2021.

[137] Miroslav Stuhl. Computing strongly connected components with cuda. *Master's thesis, Masaryk University*, 2013.

[138] Narayanan Sundaram, Nadathur Satish, Md Mostofa Ali Patwary, Subramanya R. Dulloor, Michael J. Anderson, Satya Gautam Vadlamudi, Dipankar Das, and Pradeep Dubey. Graphmat: High performance graph analytics made productive. *The Proceedings of the VLDB Endowment*, 8(11):1214–1225, jul 2015.

[139] Peter Sunehag, Guy Lever, Nicolas Heess, Jean-Baptiste Reymond, Ben Coppin, Shimon Whiteson, et al. Value-decomposition networks for cooperative multi-agent learning. In *International Conference on Autonomous Agents and MultiAgent Systems*, 2018.

[140] Siddharth Suri and Sergei Vassilvitskii. Counting triangles and the curse of the last reducer. In *Proceedings of the 20th international conference on World wide web*, pages 607–614, 2011.

[141] Richard S Sutton, Doina Precup, and Satinder Singh. Between mdps and semi-mdps: A framework for temporal abstraction in reinforcement learning. *Artificial Intelligence*, 1999.

[142] Ming Tan. Multi-agent reinforcement learning: Independent vs. cooperative agents. In *Proceedings of the tenth international conference on machine learning*, 1993.

[143] Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM journal on computing*, 1(2):146–160, 1972.

[144] John Thorpe, Yifan Qiao, Jonathan Eyolfson, Shen Teng, Guanzhou Hu, Zhihao Jia, Jinliang Wei, Keval Vora, Ravi Netravali, Miryung 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)*, pages 495–514, 2021.

[145] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. Graph attention networks. In *International Conference on Learning Representations (ICLR)*, 2018.

[146] Alexander Sasha Vezhnevets, Simon Osindero, Tom Schaul, Nicolas Heess, Max Jaderberg, David Silver, and Koray Kavukcuoglu. Feudal networks for hierarchical reinforcement learning. In *International Conference on Machine Learning*, 2017.

[147] Guozhang Wang, Wenlei Xie, Alan J Demers, and Johannes Gehrke. Asynchronous large-scale graph processing made easy. In *CIDR*, volume 13, pages 3–6, 2013.

[148] Jianyu Wang, Li Jiang, Zhuoran Wang, and Yaodong Yang. Hama: Hierarchical attention for multi-agent reinforcement learning. *arXiv preprint arXiv:2106.13229*, 2021.

[149] Jun Wang, Tianyang Zhang, Zheng-Jun Zha, Jianye Wang, and Yang Yu. Magnet: Multi-agent graph attention network for multi-agent reinforcement learning. In *NeurIPS*, 2020.

[150] Lei Wang, Qiang Yin, Chao Tian, Jianbang Yang, Rong Chen, Wenyuan Yu, Zihang Yao, and Jingren Zhou. Flexgraph: a flexible and efficient distributed framework for gnn training. In *Proceedings of the Sixteenth European Conference on Computer Systems*, pages 67–82, 2021.

[151] Minjie Yu Wang. Deep graph library: Towards efficient and scalable deep learning on graphs. In *ICLR workshop on representation learning on graphs and manifolds*, 2019.

[152] Xiaoming Wang, Jianye Hao, Yanhua Li, Xuefeng Liu, and Dong Li. Hierarchical graph-based multi-agent reinforcement learning for scalable coordination. *Journal of Artificial Intelligence Research (JAIR)*, 71:217–243, 2021.

[153] Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John 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*, pages 1–12, 2016.

[154] Yuqing Wang, Jianye Wang, Feng Wang, Haihong Jin, and Jianye Hao. Hierarchical reinforcement learning for multi-agent moba game. In *AAAI Conference on Artificial Intelligence*, 2020.

[155] Martin Winter, Daniel Mlakar, Rhaleb Zayer, Hans-Peter Seidel, and Markus 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*, pages 754–766. IEEE, 2018.

[156] Martin Winter, Rhaleb Zayer, and Markus Steinberger. Autonomous, independent management of dynamic graphs on gpus. In *2017 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–7. IEEE, 2017.

[157] Chenning Xie, Rong Chen, Haibing Guan, Binyu Zang, and Haibo Chen. Sync or async: Time to fuse for distributed graph-parallel computation. *ACM SIGPLAN Notices*, 50(8):194–204, 2015.

[158] Reynold S Xin, Joseph E Gonzalez, Michael J Franklin, and Ion Stoica. Graphx: A resilient distributed graph system on spark. In *First international workshop on graph data management experiences and systems*, pages 1–6, 2013.

[159] Reynold S. Xin, Joseph E. Gonzalez, Michael J. Franklin, and Ion Stoica. GraphX: A resilient distributed graph system on spark. In *First International Workshop on Graph Data Management Experiences and Systems*, GRADES '13, pages 2:1–2:6, New York, NY, USA, 2013. ACM.

[160] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. How powerful are graph neural networks? In *International Conference on Learning Representations (ICLR)*, 2019.

[161] Yajun Yang, Jeffrey Xu Yu, Hong Gao, Jian Pei, and Jianzhong Li. Mining most frequently changing component in evolving graphs. *World Wide Web*, 17:351–376, 2014.

[162] Pingpeng Yuan, Wenya Zhang, Changfeng Xie, Hai Jin, Ling Liu, and Kisung 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*, pages 401–412. IEEE, 2014.

[163] Dalong Zhang, Xin Huang, Ziqi Liu, Zhiyang Hu, Xianzheng Song, Zhibang Ge, Zhiqiang Zhang, Lin Wang, Jun Zhou, Yang Shuang, et al. Agl: a scalable system for industrial-purpose graph machine learning. *arXiv preprint arXiv:2003.02454*, 2020.

[164] Kaiyuan Zhang, Rong Chen, and Haibo Chen. Numa-aware graph-structured analytics. In *Proceedings of the 20th ACM SIGPLAN symposium on principles and practice of parallel programming*, pages 183–193, 2015.

[165] Lixia Zhang and Jianliang Gao. Incremental graph pattern matching algorithm for big graph data. *Scientific Programming*, 2018, 2018.

[166] Mia Zhang, Shenghua He, Yizhou Tian, Yao Xu, Yu Qiao, Zhen Wang, and Meng Wang Yang. Understanding oversmoothing in graph neural networks via graph counterfactuals. *arXiv preprint arXiv:2106.05157*, 2021.

[167] Yanfeng Zhang, Qixin Gao, Lixin Gao, and Cuirong Wang. Priter: A distributed framework for prioritized iterative computations. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, pages 1–14, 2011.

[168] Jianlong Zhong and Bingsheng He. Medusa: Simplified graph processing on gpus. *IEEE Transactions on Parallel and Distributed Systems*, 25(6):1543–1552, 2013.

[169] Jianlong Zhong and Bingsheng He. Medusa: Simplified graph processing on gpus. *IEEE Trans. Parallel Distrib. Syst.*, 25(6):1543–1552, June 2014.

[170] Huanzhou Zhu, Ligang He, Matthew Leeke, and Rui Mao. Wolfgraph: The edge-centric graph processing on gpu. *Future Generation Computer Systems*, 111:552–569, 2020.

[171] Jiawei Zhu, Haoyue Xu, Yang Liu, Jian Zhang, Fei Wu, and Li Yu. Transferable graph neural networks for long-term and short-term node influence modeling. *IEEE Transactions on Neural Networks and Learning Systems*, 2021.

[172] Rong Zhu, Kun Zhao, Hongxia Yang, Wei Lin, Chang Zhou, Baole Ai, Yong Li, and Jingren Zhou. Aligraph: A comprehensive graph neural network platform. *arXiv preprint arXiv:1902.08730*, 2019.

[173] Xiaowei Zhu, Wenguang Chen, Weimin Zheng, and Xiaosong Ma. Gemini: A {Computation-Centric} distributed graph processing system. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 301–316, 2016.