FastGNAS: Accelerating and Scaling Graph Neural Architecture Search on Multi-GPUs via Ring-Based Model Migration

ZHEN SONG*, Shandong University, China
HAO LI*, Northeastern University, China
TIANYI LI, Aalborg University, Denmark
YU GU†, Northeastern University, China
YUSHUAI LI, Aalborg University, Denmark
YANFENG ZHANG, Northeastern University, China
CHRISTIAN S. JENSEN, Aalborg University, Denmark
LIZHEN CUI†, Shandong University, China
GE YU, Northeastern University, China

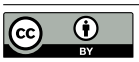
Graph Neural Architecture Search (GNAS), a subdomain of Automated Machine Learning (AutoML), aims to automate the design of graph neural network (GNN) architectures. However, GNAS is an inherently data-intensive process, as it calls for the training and evaluation of numerous candidate GNN architectures, leading to massive data access redundancy. Existing multi-GPU GNAS frameworks typically use data parallelism, which incurs high synchronization costs, or architecture parallelism, which struggles to scale to large datasets. We propose FastGNAS, a fast and scalable multi-GPU framework designed to tackle the data management challenges underlying GNAS.First, FastGNAS introduces a hybrid parallel framework that combines data and architecture parallelism, enabled by a novel ring-based model migration. Specifically, it offers an efficient data flow that exploits scalable data parallelism and isolated architecture parallelism to enable synchronization-free processing. Next, to address data redundancy, FastGNAS offers a specialized caching layer for intermediate data products, implemented through advanced batch management strategies including incremental storage and a probabilistic reuse policy. Finally, FastGNAS employs fine-grained load balancing and scheduling via exploratory task generation and predictive workload estimation, enabling purposeful resource allocation across candidate architectures. Extensive experiments show that FastGNAS can accelerate state-of-the-art baselines by an average of 3.14× (up to 6.18×) on diverse benchmarks, while achieving competitive accuracy.

CCS Concepts: • **Computing methodologies** → **Distributed computing methodologies**; • **Information systems** → **Data management systems**.

Additional Key Words and Phrases: graph neural architecture search, multi-GPU training, task decomposition, synchronization-free framework

*Both authors contributed equally to this research.
†Corresponding authors.

Authors' Contact Information: Zhen Song, Shandong University, Shandong, China, songzhen@sdu.edu.cn; Hao Li, Northeastern University, Shenyang, China, lihaoneu@stumail.neu.edu.cn; Tianyi Li, Aalborg University, Aalborg, Denmark, tianyi@cs.aau.dk; Yu Gu, Northeastern University, Shenyang, China, guyu@mail.neu.edu.cn; Yushuai Li, Aalborg University, Aalborg, Denmark, yusli@cs.aau.dk; Yanfeng Zhang, Northeastern University, Shenyang, China, Zhangyf@mail.neu.edu.cn; Christian S. Jensen, Aalborg University, Aalborg, Denmark, csj@cs.aau.dk; Lizhen Cui, Shandong University, Shandong, China, clz@sdu.edu.cn; Ge Yu, Northeastern University, Shenyang, China, yuge@mail.neu.edu.cn.



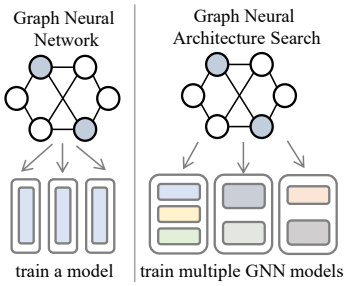
ACM 2836-6573/2026/6-ART182
https://doi.org/10.1145/3802059

ACM Reference Format:
Zhen Song, Hao Li, Tianyi Li, Yu Gu, Yushuai Li, Yanfeng Zhang, Christian S. Jensen, Lizhen Cui, and Ge Yu. 2026. FastGNAS: Accelerating and Scaling Graph Neural Architecture Search on Multi-GPUs via Ring-Based Model Migration. *Proc. ACM Manag. Data* 4, 3 (SIGMOD), Article 182 (June 2026), 27 pages. https://doi.org/10.1145/3802059

1 Introduction

Graph Neural Networks (GNNs) excel at graph data mining, but their performance is sensitive to intricate hyperparameter settings and architectural choices, making optimal model configuration critical and challenging. Graph Neural Architecture Search (GNAS) automates the selection of GNN architectures, reducing the dependence on expert knowledge and manual design while often achieving models that outperform handcrafted models. This positions GNAS as a critical enabler for advances in, e.g., social network [36, 40, 51] and knowledge graph [4, 13] analysis, drug discovery [43, 52, 55], blockchain transaction analysis [44], and traffic prediction [11, 17, 18, 22, 23].

GNAS aims to find optimal GNN architectures in potentially very large search spaces, incurring potentially much higher computational and storage costs than in the case of single-architecture GNN training. Fig. 1 highlights key differences between parallel GNN training and parallel GNAS training. The former focuses on a single model, whereas GNAS simultaneously trains a set of models, typically ranging from thousands to hundreds of thousands of models. In addition to conventional data parallelism, GNAS can leverage architecture parallelism to accommodate heterogeneous model structures. During training, GNAS must manage multiple models with diverse architectures and must incorporate procedures for model evaluation, search strategy update, and architecture generation as part of the search process.



	GNN	GNAS
Training Object	Single GNN model on a single graph	Multiple GNN models on a single graph
Parallelism Type	Data parallelism	Data parallelism and architecture parallelism
Partition Target	Graph data	Graph data and GNN architectures
Task Execution	Training, inference	Training, inference, evaluation and architecture generation

Fig. 1. Traditional GNN tasks vs. GNAS tasks.

While parallel GNN training systems improve single-architecture training efficiency through scheduling [30, 58], quantization [31, 33, 38], caching [27, 32, 57], memory optimization [15, 35, 54], and operator fusion [7, 21, 49], they are not designed to support the inter-architecture parallelism and interactions that can benefit GNAS. Further, existing parallel GNAS systems primarily adopt either data parallelism [34, 56] or architecture parallelism [2, 6, 28]. The former necessitates frequent model synchronization among parallel workers since a single candidate architecture is updated collectively. The latter requires each worker to store and train on the full dataset, causing scalability issues. Additionally, candidate architectures often exhibit heterogeneous computational workloads. As a result, prevailing frameworks are prone to high synchronization overheads, limited scalability, and severe load imbalance. Overall, key challenges in parallel GNAS have been addressed insufficiently. We summarize the main design challenges in achieving parallel GNAS systems as follows.

Challenge I: Limited parallel training and data-management-induced synchronization and scalability issues. Existing parallel GNAS frameworks typically rely on either data parallelism or

architecture parallelism. The former incurs high synchronization costs as multiple GPUs repeatedly exchange intermediate model states for a single architecture, while the latter suffers from poor scalability due to redundant dataset storage across GPUs. These data movement and replication bottlenecks call for a parallel training paradigm that jointly mitigates synchronization overheads and data management costs, thereby improving scalability.

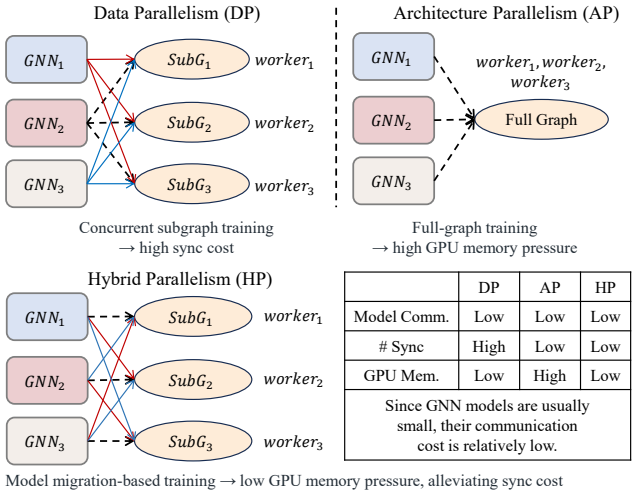
Challenge II: Inefficient, data-intensive mini-batch generation and redundant batch construction across GNN architectures. Mini-batch construction is often a major performance bottleneck during training, at times surpassing the computational cost of model optimization itself [37, 39]. In GNAS, repeatedly generating mini-batches for multiple candidate architectures amplifies data access, graph sampling, and encoding overheads, causing severe redundancy and reduced training throughput. This calls for data management mechanisms for caching and reusing mini-batches across architectures, thereby minimizing duplicate data processing.

Challenge III: Data- and architecture-induced workload variability causing system-level load imbalance. GNN architectures exhibit substantial variation in training workloads due to differences in depth, hidden dimensions, and aggregation mechanisms, which in turn drive heterogeneous data access, sampling, and communication patterns. This heterogeneity leads to imbalanced GPU and I/O utilization, where lightweight architectures finish much earlier and complex ones become stragglers, causing device idling. This highlights the need for cost-based, workload-aware scheduling and resource management that account for data movement and access costs to mitigate load imbalance.

To address these challenges, we propose a synchronization-free, scalable, and parallel framework for GNAS on multi-GPU systems, with explicit attention to data movement, data reuse, and cost-based scheduling. To address ***Challenge I***, we propose a hybrid parallelism paradigm that integrates data and architecture parallelism, thereby reducing synchronization overhead, mitigating redundant data replication, and enhancing scalability. To address ***Challenge II***, we propose an incremental batch storage and reuse strategy that maintains a shared mini-batch repository, substantially reducing redundant mini-batch generation and graph sampling across architectures. To address ***Challenge III***, we propose a fine-grained, cost-based task partitioning mechanism, informed by workload prediction, to achieve balanced GPU utilization under heterogeneous data access patterns. The three core data management contributions—**parallel data processing and migration**, **shared data caching and reuse**, and **imbalanced load management**—are summarized as follows.

- We propose FastGNAS[1], a parallel GNAS framework that combines data- and architecture-parallel execution to improve efficiency and scalability. It uses a novel ring-based model migration scheme and an efficient data flow that exploits scalable data parallelism and isolated architecture parallelism for synchronization-free processing.
- We propose incremental batch storage and a probabilistic batch reuse model over a shared mini-batch repository, forming a specialized caching layer for intermediate data products that enables efficient cross-architecture reuse and markedly reduces redundant batch construction overheads.
- We propose fine-grained task partitioning that uses an Estimator to predict architecture cost; by splitting large tasks into cost-similar subtasks, it achieves balanced loads and efficient resource allocation in parallel multi-GNN training.
- We evaluate FastGNAS on four publicly available datasets in two real-world GPU-cluster environments. Experimental results show that FastGNAS achieves an average speedup of 3.14× (up to 6.18×) over state-of-the-art methods on diverse benchmarks, while achieving models with competitive accuracy.

[1]https://github.com/lghosth/FastGNAS



	DP	AP	HP
Model Comm.	Low	Low	Low
# Sync	High	Low	Low
GPU Mem.	Low	High	Low

Since GNN models are usually small, their communication cost is relatively low.

Fig. 2. Parallel framework comparison.

The remainder of the paper is organized as follows. Section 2 surveys GNAS algorithms, parallel GNN processing, and parallel GNAS systems. Section 3 covers preliminaries. Section 4 presents the system framework and batch management techniques. Section 5 details the fine-grained task partitioning strategy. Section 6 provides a theoretical analysis, while Section 7 discusses experimental results. Finally, Section 8 summarizes the contributions.

2 Related Work

2.1 GNAS Algorithms

GNAS algorithms primarily aim to improve the quality of discovered architectures, using reinforcement learning [5, 19], entropy-driven sampling and weight sharing [64], and probabilistic gradient-based methods [62] to explore both micro-level operators and macro-level design choices. Differentiable NAS approaches support joint optimization of models and architectures in unified supernetworks. These methods differ in search space granularity—from low-level operations to high-level motifs—and employ diverse strategies including reinforcement learning [3, 5, 19, 29, 60], differentiable one-shot [12, 20, 25, 47, 59, 61], and evolutionary algorithms [8, 9, 24]. However, most GNAS methods run on a single device, limiting scalability on large graphs or multi-GPU systems. Thus, despite progress in GNN architecture search, parallel data management and scalable execution remain open challenges that this work addresses.

2.2 Parallel GNAS Frameworks

Most parallel GNAS frameworks use either architecture or data parallelism. Table 1 compares single- and multi-GPU GNAS systems, and Fig. 2 shows three representative parallel execution frameworks. In the figure, dashed lines mark tasks being executed, and lines of the same color indicate tasks scheduled in parallel.

Data parallelism vs. architecture parallelism. In architecture-parallel GNAS [2, 6, 28], a centralized controller assigns candidate architectures to workers for independent training and evaluation, but each worker must store the full dataset, creating a memory bottleneck on large graphs. Data-parallel GNAS [34, 56] partitions the dataset across workers, but incurs heavy synchronization overhead from frequent parameter updates. Thus, existing parallel GNAS frameworks face high

Table 1. Comparison of single-GPU and multi-GPU GNAS systems. **Notation:** scalability and load balance: Good/Medium/Poor; synchronization overhead: High/Low.

System	Part.	Framework	Scal.	Sync.	Bala.	Drawbacks
GraphNAS [5]	Full	Single GPU	Poor	N/A	N/A	Poor scalability; high search cost for large graphs due to limited resources.
AutoGNN [64]	Full	Single GPU	Poor	N/A	N/A	
PDNAS [62]	Full	Single GPU	Poor	N/A	N/A	
Policy-GNN [19]	Full	Single GPU	Poor	N/A	N/A	
AutoGraph [24]	Full	Single GPU	Poor	N/A	N/A	
DHGAS [59]	Full	Single GPU	Poor	N/A	N/A	
GraphNAS++ [6]	Full	Architecture Parallelism	Poor	Low	Poor	High synchronization overhead; low scalability; inefficient coordination and data sharding, especially for heterogeneous or dynamic workloads.
GraphNAP++ [34]	Subgraph	Data Parallelism	Good	High	Medium	
GraphPAS [2]	Full	Architecture Parallelism	Poor	Low	Poor	
PasCa [56]	Subgraph	Data Parallelism	Good	High	Medium	
MANAS [28]	Full	Architecture Parallelism	Poor	Low	Poor	
FastGNAS (Ours)	Subgraph	Hybrid Parallelism	Good	Low	Good	Balanced, efficient, and scalable

synchronization costs and limited scalability, highlighting the need for more effective parallel solutions for large-scale GNAS.

Motivation for using hybrid parallelism. We propose a hybrid parallelism framework that integrates data parallelism and architecture parallelism, thereby combining the scalability benefits of data-parallel approaches with the reduced synchronization overhead inherent in architecture-parallel designs. Model migration is employed to enable training of candidate models across all subgraphs. Model migration is mainly motivated by the lightweight nature of GNN models, as communication overhead primarily stems from exchanging vertex embeddings rather than model parameters.

2.3 Other Parallel Frameworks

Parallel GNN training. Recent advances in parallel GNN training have centered on improving scalability, efficiency, and resource utilization through unified programming interfaces, communication reduction, and optimized resource management. Techniques such as mini-batch processing [63, 65], graph partitioning [41], offline sampling frameworks [53], pipelined execution [42, 46], and advanced caching [27] have been widely adopted to balance workloads and mitigate communication bottlenecks. Further optimizations—including message compression [38, 45], adaptive sampling [37], heterogeneous CPU/GPU computing [1, 26]—enhance sampling efficiency and throughput for large-scale and heterogeneous workloads. However, these systems are primarily optimized for GNN training, focusing on the development of a single GNN model. Consequently, they do not sufficiently address the complex scheduling mechanisms, data management strategies, and resource sharing challenges intrinsic to multi-architecture environments commonly encountered in GNAS tasks. It highlights the necessity for dedicated parallel system designs and optimization strategies to efficiently support large-scale, multi-architecture GNN training.

3 Preliminaries

First, we follow the literature by presenting the definition of an attributed graph in Definition 1. Definition 2 provides the formal definition of a graph neural network, while Definition 3 characterizes the task of graph neural architecture search. Example 1 is included to illustrate the main workflow of graph neural architecture search. Finally, Definition 4 clarifies the meanings of architectures, models, and frameworks as used in this paper.

Table 2. Search space of model configurations.

Component	Search Space
Encoders	{const, gcn, gat, sym-gat, cos, linear, gene-linear}
Aggregators	{sum, mean, max, mlp}
Activations	{sigmoid, tanh, relu, none, softplus, leaky relu, relu6, elu}
Attention heads	{1, 2, 4, 6, 8}
Hidden dimensions	{8, 16, 32, 64, 128, 256}

Definition 1 (**Attributed Graph**). *An attributed graph is defined as $G = (\mathcal{V}, \mathcal{E}, \mathcal{X})$, where $\mathcal{V}$ is a set of N vertices, $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$ is a set of edges, and $\mathcal{X} \in \mathbb{R}^{N \times F}$ is a vertex attribute matrix; and N is the number of vertices, and F is the feature dimensionality.*

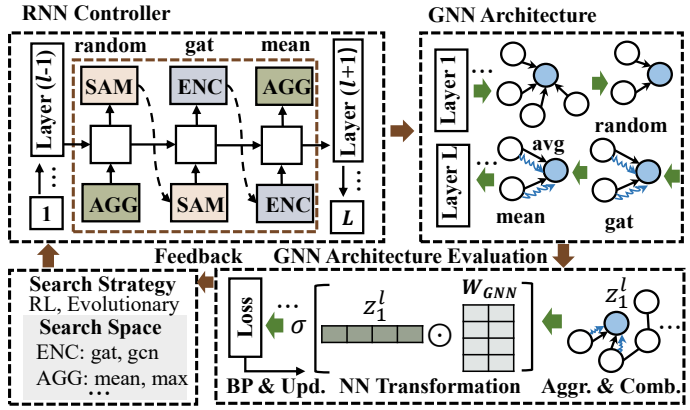


Fig. 3. GNAS workflow.

Definition 2 (**Graph Neural Networks**). *Graph Neural Networks (GNNs) are neural models designed to encode both structural (spatial) and attribute information from attributed graphs. They operate by iteratively aggregating and transforming node embeddings from local neighborhoods, guided by their graph topology. At each layer, a GNN updates a node's representation by combining its own previous embedding with aggregated information from the nodes in its neighborhood, followed by nonlinear transformation. Formally, for vertex v_i in layer l:*

$$\boldsymbol{z}^l(v_i) = \textsc{Comb}\left(\textsc{Agg}\left(\left\{\boldsymbol{h}^{l-1}(v_j) \mid v_j \in \mathcal{N}(v_i)\right\}\right), \boldsymbol{h}^{l-1}(v_i)\right) \tag{1}$$

$$\boldsymbol{h}^l(v_i) = \sigma\left(\boldsymbol{z}^l(v_i) \cdot \boldsymbol{W}_{GNN} + \boldsymbol{b}\right), \tag{2}$$

where $\mathcal{N}(v_i)$ denotes the set of neighbors of v_i, $\textsc{Agg}(\cdot)$ and $\textsc{Comb}(\cdot)$ are aggregation and combination functions, $\sigma(\cdot)$ is a nonlinear activation function, W_{GNN} denotes the learnable parameters, and $\boldsymbol{b}$ is a bias term. Stacking L layers allows a GNN to integrate information from up to L-hop neighborhoods.

Definition 3 (**Graph Neural Architecture Search**). *Graph Neural Architecture Search (GNAS) aims to automatically identify optimal graph neural network architectures. It generally includes three key components:*

- ***Search Space:*** *Defines the configurable components and their options in GNNs, such as aggregation and combination functions, attention mechanisms, activation functions, connection structures, and sampling methods. Table 2 shows the search space.*
- ***Search Strategy:*** *Specifies how architectures are explored and combined within the search space, commonly employing reinforcement learning, evolutionary algorithms, or differentiable search.*

- ***Evaluation Method:*** *Determines the quality of candidate architectures, typically via model training (full or partial) and often leveraging techniques like weight sharing or one-shot evaluation.*

Example 1. *Fig. 3 shows a typical GNAS workflow. A search strategy (e.g., an RNN controller in RL-based search) samples candidate architectures from the search space, each encoded as an operation sequence (e.g., ["random", "gat", "mean"] per layer). The corresponding GNNs are built, trained, and evaluated on a validation set; the resulting accuracy is used as a reward to update the controller. This process repeats to gradually find better architectures.*

FastGNAS is an underlying system framework designed to facilitate the development of diverse GNAS algorithms. It enables users to flexibly define customized search spaces, search strategies, and evaluation methods. All GNAS algorithms built using FastGNAS can exploit its parallel framework and advanced system-level optimizations. Additionally, we clarify three key concepts as follows.

Definition 4 (**Architecture**, **Model**, **Framework**). *The term **architecture** refers to the structure of a GNN, i.e., the specific design or configuration of the GNN. The **model** denotes an instantiation of a GNN architecture, that is, a concrete object with instantiated parameters given a particular architecture. The term **framework** refers to the parallel execution system that enables GNAS.*

4 Hybrid Parallelism and Batch Scheduling

4.1 System Overview

Fig. 4 shows FastGNAS, a decentralized parallel framework composed of multiple workers. Each worker includes five core modules: Architecture Generator, Architecture Partitioner, Data Manager, Probabilistic Batch Scheduler, and Incremental Batch Builder.

- **Architecture Generator** manages the generation of candidate architectures and updates search strategies, incorporating exploratory generation strategies for GNN architectures.
- **Architecture Partitioner** distributes architecture-related tasks among individual workers.
- **Workload Estimator** evaluates the computational demands of each GNN architecture.
- **Data Manager** distributes data to workers and monitors the per-architecture data workload for each model migration, enabling load-aware scheduling across workers.
- **Probabilistic Batch Scheduler** schedules mini-batches by dynamically selecting between newly generated and cached batches.
- **Incremental Batch Builder** manages batch storage and facilitates incremental batch construction.

The Data Manager loads the graph, statically partitions it, and assigns subgraphs to workers, which cache L-hop neighbor subgraphs in GPU memory. The Architecture Generator produces S candidate GNNs, and the load estimator evaluates their compute demands. The Load-aware Task Partitioning module then allocates training tasks by estimated load to achieve balanced, synchronization-free execution. During training, exploratory task generation increases model diversity while managing the trade-off between load balance and search quality. The probabilistic batch generator chooses between new and cached batches; if cached batches are insufficient for the current GNN depth, the Incremental Batch Builder constructs additional batches.

4.2 Hybrid Parallel Framework with Ring-Based Model Migration

To avoid synchronization overhead, each worker trains a different GNN architecture on its local subgraph, with the dataset partitioned across workers for scalability. However, training each architecture on only one subgraph risks convergence to local optima, so we migrate models across devices: each architecture is sequentially trained on all subgraphs, improving robustness and convergence.

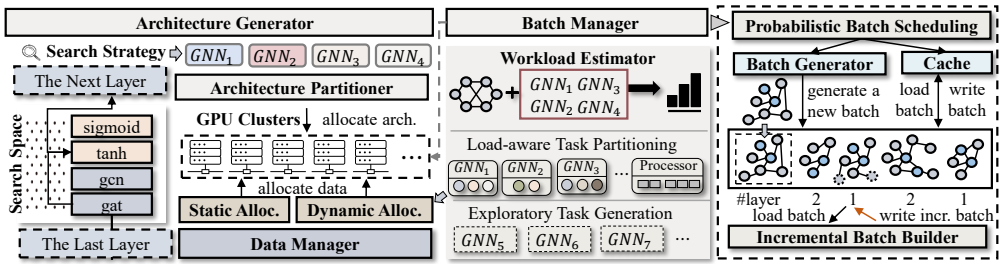


Fig. 4. Overview of FastGNAS.

Each worker first trains its assigned GNN architecture on its local subgraph (we simply say it trains the architecture). Since workers handle different architectures, no inter-worker parameter synchronization is needed. Model parameters are then passed to the next worker in a ring; with N_{wk} workers, after N_{wk} migrations, each architecture has been trained on all data partitions.

This framework is motivated by several considerations: (i) GNN models generally have a modest number of parameters, allowing for efficient transfer across GPU devices; (ii) after N_{wk} migrations, each model has been trained on the entire dataset, thereby achieving accuracy comparable to single-worker algorithms; (iii) since each model is updated exclusively by a single worker at any time, model synchronization overhead is avoided; and (iv) each worker maintains a distinct subgraph, which ensures scalability as the number of devices increases.

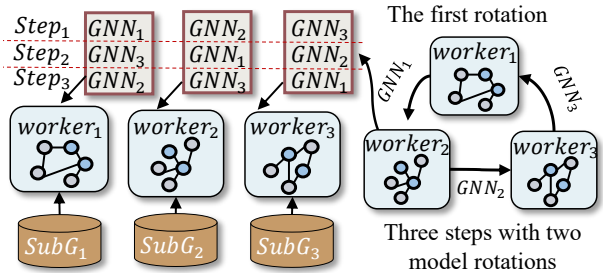


Fig. 5. Ring-based model migration.

Example 2. *As shown in Fig. 5, suppose there are $N_{wk} = 3$ workers, each holding a local subgraph $\mathcal{G}_1, \mathcal{G}_2, \mathcal{G}_3$ and an initial GNN model $\mathcal{M}_1^{(1)}, \mathcal{M}_2^{(1)}, \mathcal{M}_3^{(1)}$, respectively. Here, $\mathcal{M}_i^{(k)}$ denotes the i^{th} model at the k^{th} rotation. In the k^{th} rotation, each model $\mathcal{M}_1^{(i)}$ is trained for $|\boldsymbol{B}|$ mini-batches to obtain the updated model for the $(k+1)^{th}$ rotation. The resulting model is then transferred to the worker indexed by $i_{rot} = ((k + i - 2) \bmod N_{wk}) + 1$, i.e.,*

$$\mathcal{M}_i^{(k+1)} = Train(\mathcal{M}_i^{(k)}, \mathcal{G}_{i_{rot}}), \quad i = 1, 2, \ldots, N_{wk} \tag{3}$$

After each rotation, the models are transferred to the next worker in a ring topology; for example, $\mathcal{M}_1$ is passed to worker_2, *$\mathcal{M}_2$ to* worker_3, *and $\mathcal{M}_3$ to* worker_1. *This process is repeated for $N_{wk} - 1$ rotations, ensuring that by the end, each model has been trained on all subgraphs. For example, $\mathcal{M}_1^{(4)}$ represents the first model after one complete pass over the entire dataset.*

$$\mathcal{M}_1^{(4)} = Train(Train(Train(\mathcal{M}_1^{(1)}, \mathcal{G}_1), \mathcal{G}_2), \mathcal{G}_3). \tag{4}$$

`FastGNAS` performs a single pass over each subgraph, then moves to the next worker for the next subgraph, yielding results identical to those of single-worker training. Example 3 shows this process.

Example 3. *Suppose a 100-vertex graph is split into 10 mini-batches. In the single-worker case, mini-batches 0–9 are processed sequentially for each of T epochs. In the two-worker case, mini-batches 0–4 are assigned to worker 1 and 5–9 to worker 2.* FastGNAS *runs 0–4 on worker 1, migrates the model to worker 2 for 5–9 to complete an epoch, then repeats for T epochs. The training outputs are identical to the single-worker case. Thus, for m mini-batches and N_{wk} workers, assigning the mini-batches in a range-based manner to consecutive workers can reproduce single-machine SGD exactly.*

4.3 Probabilistic Batch Sharing Optimization

During GNAS, different GNNs sample many overlapping mini-batches from the same graph, causing redundancy. Efficient storage and reuse of mini-batches are thus critical. We address this with a probabilistic batch-sharing scheme.

Batch Storage Structure. We first define the mini-batch storage object to formally describe the storage structure. Then, we illustrate its practical operation with a concrete example.

Definition 5 (**Mini-Batch Cache Object**). *For the mini-batch with the target vertex set $\boldsymbol{B}_i$, we define the corresponding mini-batch storage object MCO_i as follows:*

$$MCO_i = (\boldsymbol{B}_i, \mathcal{E}(\boldsymbol{B}_i)), \tag{5}$$

where $\mathcal{E}(\boldsymbol{B}_i)$ is the set of edge-related objects associated with $\boldsymbol{B}_i$. More specifically, $\mathcal{E}(\boldsymbol{B}_i)$ consists of edge objects of multiple layers:

$$\mathcal{E}(\boldsymbol{B}_i) = \left\{\mathcal{E}^1(\boldsymbol{B}_i), \mathcal{E}^2(\boldsymbol{B}_i), \ldots, \mathcal{E}^{\mathcal{L}}(\boldsymbol{B}_i)\right\}, \tag{6}$$

where $\mathcal{L}$ denotes the maximum number of message-passing layers in the GNN. For any layer l, the edge-list unit ELU_i^l is defined as

$$ELU_i^l = \left\{\mathcal{E}^l(\boldsymbol{B}_i), Map\left(\mathcal{E}^l(\boldsymbol{B}_i)\right)\right\}, \tag{7}$$

where $\mathcal{E}^l(\boldsymbol{B}_i)$ denotes the edge list for the target vertices $\boldsymbol{B}_i$ at the l^{th} layer, and $Map(\mathcal{E}^l(\boldsymbol{B}_i))$ represents the mapping of global vertex indices to locally re-indexed integers (starting from 0) within mini-batch $\boldsymbol{B}_i$ at layer l.

We present an example to elucidate the proposed mini-batch storage structure. In this design, only structural information is stored within the mini-batch, while attribute data are excluded to optimize GPU memory usage. Features are dynamically accessed from the feature matrix during training as required. Since features reside in GPU memory, accessing them via indexing incurs negligible overhead. We adopt this strategy for two main reasons: (i) feature data is stored on the GPU, so retrieving it through indexing is inexpensive; and (ii) caching feature data separately for every mini-batch would introduce substantial redundancy and reduce GPU memory utilization considerably.

Example 4. *Consider a graph with vertices $\{0, 1, 2, 3, 4\}$ and edges $\{(0, 1), (1, 2), (2, 3), (3, 4)\}$. Suppose the target vertex set for the i^{th} mini-batch is $\boldsymbol{B}_i = \{2, 3\}$ and focus on the first layer. The corresponding edge set $\mathcal{E}^1(\boldsymbol{B}_i)$ is $\{(1, 2), (2, 3), (3, 4)\}$, involving vertices $\{1, 2, 3, 4\}$. For efficient mini-batch computation, these vertices are locally re-indexed as $1 \mapsto 0$, $2 \mapsto 1$, $3 \mapsto 2$, and $4 \mapsto 3$, resulting in the re-indexed edge list $\{(0, 1), (1, 2), (2, 3)\}$. Thus, the storage object for this mini-batch at layer $l = 1$ comprises the edge set $\mathcal{E}^1(\boldsymbol{B}_i)$ and the mapping $Map(\mathcal{E}^1(\boldsymbol{B}_i))$ as described above.*

Algorithm 1 outlines distributed GNAS training with model migration. Training runs for T epochs (Line 1). Each epoch has N_{wk} rotations; in rotation k, the model on worker wid is $((k + wid - 2) \bmod N_{\text{wk}}) + 1$, ensuring all models visit all workers and subgraphs (Lines 2–8). For each mini-batch in the local subgraph, the designated model is updated via mini-batch training (Lines

Algorithm 1 Model Migration Training

Input: GNN Architecture List ***gnn_list***; Number of Training Epochs T; Number of Workers N_{wk}
Output: Best Model ***best_model***
for all worker *wid* in $[1, 2, \ldots, N_{\text{wk}}]$ **in parallel**

```
1:  for t = 1 to T do
2:      for k = 1 to N_wk do
3:          model_id ← ((k + wid − 2) mod N_wk) + 1
4:          for b = 1 to |B| do
5:              train(gnn_list[model_id], G_wid[b])
6:          end for
7:          migrate_model(gnn_list[model_id], wid + 1)
8:      end for
9:  end for
10: reward ← eval(gnn_list)
11: update_grt_strategy(gnn_list, reward)
12: topk_model ← update_topk_model(gnn_list, reward)
13: best_model ← test(topk_model)
14: return best_model
```

4–6). After each rotation, the trained model is migrated to the next worker in the ring (Line 7). Upon completion of all training epochs, all models are evaluated, the search strategy is updated, and the top-performing models are selected for final testing to determine the best architecture (Lines 10–14).

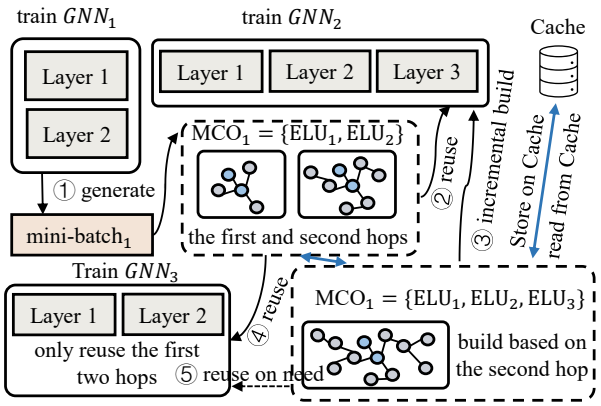


Fig. 6. Cache structure and construction.

Batch Incremental Construction. Variations in the number of GNN layers require sampling at different depths, necessitating batch-wise incremental construction methods tailored to each sampling depth. For instance, consider a training scenario involving two 2-layer GNNs (GNN_1 and GNN_3) and one 3-layer GNN (GNN_2). The naive approach constructs separate 2-layer and 3-layer mini-batches for GNN_1, GNN_3, and GNN_2, respectively, with GNN_1 reusing the batch created for GNN_3. However, this results in suboptimal data reuse; for example, there is no sharing between GNN_2 and GNN_1 or GNN_3. To address this, we leverage the proposed batch storage structure to support incremental batch construction within a unified storage architecture.

When the batch manager creates a new mini-batch, FastGNAS stores it in the appropriate edge list unit (ELU) within the cache. For an L-layer GNN architecture, each mini-batch includes L ELUs, with each ELU holding the edge list and mapping information for one layer. The mini-batch initializes a cache object (MCO), which instantiates L ELUs to accommodate all layers. Cached mini-batches are selected for reuse in a probabilistic manner. If a reused MCO contains fewer than L ELUs, the batch manager incrementally generates $L - |ELU|$ additional ELUs to expand the MCO; otherwise, the first L ELUs are reused directly.

Example 5. *In Fig. 6, suppose the GNN models to be trained are GNN_1, GNN_2, and GNN_3, with 2, 3, and 2 layers, respectively. When training GNN_1 for the first time, the mini-batch cache object (MCO_1) are created, containing 2 ELUs. For instance, $MCO_1 = \{ELU_1, ELU_2\}$, where ELU_1 and ELU_2 store the first- and second-hop neighbors of mini-batch$_1$, respectively (Step ①). When training GNN_2 and reusing MCO_1, ELU_3 is generated based on ELU_2, expanding MCO_1 to $MCO_1 = \{ELU_1, ELU_2, ELU_3\}$ for reuse (Steps ② and ③). Subsequently, when training GNN_3 and reusing MCO_1, only the first two ELUs $\{ELU_1, ELU_2\}$ are required (Step ⑤).*

Probabilistic batch sharing. Cache reuse often encounters the cold start problem, where initial cache occupancy is low and the likelihood of reuse is minimal. As the batch manager continually generates new mini-batches, the number of MCOs present in the cache increases, thereby raising the probability of cache reuse. This trend supports the validity of the reuse strategy: a larger and more diverse cache reduces the risk of information loss. Moreover, increasing the sampling fan-out (N_{fo}) enriches each MCO with information from more vertices, further mitigating information loss. Consequently, the cache reuse probability should be proportional to $|MCO| \cdot N_{fo}$, where $|MCO|$ denotes the number of MCOs and N_{fo} is the sampling fan-out. To normalize this probability when the dataset is fully covered, we define

$$p = \frac{(|MCO| \cdot S_{mb}) \cdot N_{fo}}{N \cdot \bar{d}}, \tag{8}$$

where S_{mb} is the mini-batch size, N is the number of vertices, and $\bar{d}$ is the average graph degree.

5 Fine-Grained Partitioning-Based Load Balancing Strategy

Due to the varying workloads associated with different GNNs, simply allocating identical data loads and an equal number of GNNs to each worker does not guarantee balanced execution-time workloads. Based on this observation, we propose a *Fine-Grained Partitioning-Based Load Balancing Strategy*. Specifically, we first employ a workload estimator to assess the workload of a given GNN architecture. Then, guided by workload estimations, we logically assign smaller data subsets to architectures with higher computational demands, thereby maintaining balanced task workloads throughout each model migration. Finally, we introduce an exploratory GNN architecture generation mechanism, which adaptively generates GNN architecture that can be distributed in a balanced manner.

5.1 Workload Estimation for GNN Architectures

During GNAS execution, computational workloads vary considerably across different GNN architectures. Therefore, accurate workload estimation is crucial for balanced task partitioning.

Individual operators have different computational costs; for example, GCN performs one spectral aggregation per layer, whereas GAT applies attention at every layer. Two-layer GNNs are much cheaper than three-layer ones, and workload scales non-linearly with depth. It also depends on graph sparsity and hardware, e.g., $T_{\text{GAT}}/T_{\text{GCN}}$ on $\mathcal{G}_1$ can exceed that on $\mathcal{G}_2$ when $\mathcal{G}_1$ has higher density than $\mathcal{G}_2$. Given these factors, an accurate analytic cost model is infeasible, so we use an

offline-trained neural estimator M_{load} to predict workload ratios across graphs, architectures, and hardware.

First, the input dimensions for $\mathcal{M}_{\text{load}}$ must be specified. The computational workload of a GNN architecture is influenced by three primary factors: the GNN architecture type (C_{arc}), graph characteristics (C_{gra}), and hardware specifications (C_{dev}). Let $L_{\max}$ denote the maximum number of GNN layers; the input to $\mathcal{M}_{\text{load}}$ thus consists of L sets of layer-specific attributes. For each layer, the relevant components include activation type, attention mechanism, aggregation method, number of attention heads, and hidden units.

Let the input feature vector be $\mathbf{x} = [\, \mathbf{x}_{arc} \,\|\, \mathbf{x}_{gra} \,\|\, \mathbf{x}_{dev} \,]$,

where $\mathbf{x}_{arc}$, $\mathbf{x}_{gra}$, and $\mathbf{x}_{dev}$ represent architecture, graph, and device features, respectively. Categorical components are encoded using one-hot representations based on the number of possible choices. For example, if aggregation type includes "sum", "mean", "max", and "mlp", it is encoded as a four-dimensional one-hot vector, where each value corresponds to $[1, 0, 0, 0]$, $[0, 1, 0, 0]$, $[0, 0, 1, 0]$, or $[0, 0, 0, 1]$, respectively. Numerical components are normalized prior to inclusion as input features. Graph characteristics (C_{gra}) encompass the number of vertices, number of edges, average node degree, and graph diameter. Device specifications (C_{dev}) include GPU core count, core frequency, and floating-point performance. These attributes are processed analogously: numerical features are normalized, and categorical features are one-hot encoded. For model training, the mean runtime measured over 10 repeated executions serves as the label. Formally, given input $\mathbf{x}$, the load prediction model M_{load} is trained to predict the mean runtime, $y = \mathcal{M}_{\text{load}}(\mathbf{x})$.

In addition, when users run architecture search with `FastGNAS`, the system automatically logs data. and retrains the Estimator on demand, continuously improving its accuracy.

5.2 Load-Aware Task Scheduling

Due to heterogeneous workloads across GNN architectures, balancing load among workers is critical for efficiency. We jointly optimize task generation and scheduling to build a multi-GNN training framework with balanced computation.

Fine-Grained Task Partitioning and Scheduling. In our parallel framework, each model is trained sequentially on subgraphs assigned to all workers. Let N_{wk} be the number of workers and N_{search} the number of architectures per search. If each worker trains one model per rotation, the number of migrations is $N_{\text{rot}} = \lceil N_{\text{search}}/N_{wk} \rceil$, and the total parallel training time is:

$$\sum_{i=1}^{N_{\text{rot}}} \max_{j=1,\ldots,N_{wk}} \mathcal{M}_{\text{load}}((\text{GNN}_{(i-1)\cdot N_{wk})+j}), \tag{9}$$

where $\lceil \cdot \rceil$ represents the ceiling operations. However, this naive scheduling strategy is prone to bottlenecks, as the duration of each rotation is determined by the GNN model with the highest load, resulting in load imbalance and reduced parallel efficiency.In addition, when the number of epochs is limited, such coarse-grained rotation may lead to imbalanced training data, causing the model to overfit to the subgraph currently being trained.

To address this, we adopt fine-grained task partitioning: each GNN is assigned a logical data share inversely proportional to its predicted workload. For example, if GNN_1 is twice as expensive as GNN_2, it receives half the data of GNN_2, approximately equalizing computation per migration.

Additionally, tasks are scheduled in descending order of computational load to prevent high-load models from becoming stale. The partitioning is performed at the logical level and does not require physical data movement, resulting in negligible overhead. This approach not only balances workloads and improves training efficiency, but also—by promoting more randomized

data distributions for each model—enhances the generalization capability of the trained GNN architectures.

5.3 Exploratory Task Generation

Even with fine-grained task partitioning, simply allocating identical equal numbers of GNNs to each worker fails to achieve balanced workloads. To address this, we introduce an exploratory architecture generation mechanism that adaptively produces training tasks tailored for balanced execution among workers.

Rather than predetermining the number of GNN architectures to be generated, FastGNAS dynamically adjusts this number in response to the estimated computational loads. Users specify a minimum number of architectures ($NG_{\min}$) to be trained, and the system initially generates $NG_{\min}$ candidate architectures in a batch. The workload estimator then evaluates the computational requirements of each candidate. A task balance evaluation module assesses whether the aggregated workload meets the predefined load balancing criterion. If the criterion is not satisfied, additional architectures are incrementally produced via the GNAS search strategy, with the balance condition re-evaluated after each addition. This process continues until the balance criterion is met, at which point architecture generation concludes. Alternatively, the process terminates when K_{expand} expansions are reached.

Algorithm 2 Exploratory Architecture Generation

Input: GNN Architecture List ***gnn_list***; Worker Number N_{wk}
Output: Optimized GNN Architecture List ***bucket_gnn***

1: $bucket\_wl \leftarrow [0 \textbf{ for } i \in [1, 2, \ldots, N_{\text{wk}}]]$ ▷ Initialize workloads
2: $bucket\_gnn \leftarrow [\,[\,] \textbf{ for } i \in [1, 2, \ldots, N_{\text{wk}}]]$
3: $gnn\_wl\_list \leftarrow \mathcal{M}_{\text{load}}(\boldsymbol{gnn\_list})$ ▷ Evaluate workloads
4: **for** $gid = 1$ **to** $N_{\text{wk}} + K_{\text{expand}}$ **do**
5: $wl \leftarrow gnn\_wl\_list[gid]$
6: $bid \leftarrow$ **getMinWorkloadBucket**$(bucket\_wl)$
7: $bucket\_gnn[bid]$.**append**(gid)
8: **if** $gid \leq N_{\text{wk}}$ **then**
9: **continue** ▷ Skip for initial assignments
10: **end if**
11: $bid\_max \leftarrow$ **getMaxWorkloadBucket**$(bucket\_wl)$
12: $bid\_min \leftarrow$ **getMinWorkloadBucket**$(bucket\_wl)$
13: **if** $\dfrac{bid\_max - bid\_min}{bid\_max} \leq \alpha$ **then**
14: **break** ▷ Stop if load is balanced
15: **else**
16: $new\_gnn \leftarrow$ **generateGNNByController()**
17: $new\_wl \leftarrow \mathcal{M}_{\text{load}}(new\_gnn)$
18: ***gnn_list***.**append**(new_gnn)
19: ***gnn_wl_list***.**append**(new_wl)
20: **end if**
21: **end for**
22: **return** ***bucket_gnn***

Algorithm 2 shows the process of exploratory architecture generation. We construct N_{wk} buckets representing the available workers (Lines 1–2). Candidate GNN architectures are generated and

their computational workloads are estimated (Line 3). These architectures are then iteratively assigned to the bucket with the lowest current load, reducing the risk of latency bottlenecks caused by heavily loaded architectures (Lines 5–7). For the initial assignments ($gid \leq N_{\text{wk}}$), FastGNAS ensures that each worker is allocated at least one task (Lines 8–10). After the initial allocation, the system evaluates the load balance by comparing the maximum and minimum bucket workloads. If the resulting imbalance is within the tolerance α, architecture generation terminates (Lines 13–15). Otherwise, additional GNN architectures are synthesized, their workloads estimated, and both lists updated accordingly (Lines 16–19). This process is repeated until either the balance criterion is met or the maximum number of architectures is reached.

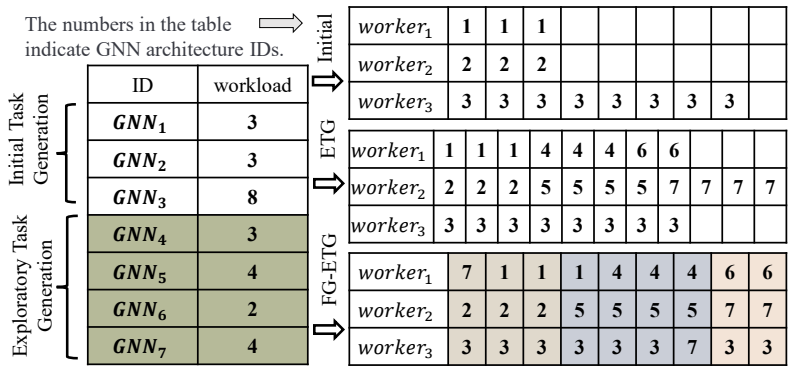


Fig. 7. Exploratory generation-based task scheduling.

Example 6. *In Fig. 7, consider three workers and three GNN architectures with computational loads of 3, 3, and 8 time units, respectively. The naive partitioning strategy allows only three architectures to be processed in eight time units, leaving* $worker_1$ *and* $worker_2$ *largely idle. Exploratory task generation increases the number of processed architectures to seven in eleven time units, but idle time still exists. In contrast, fine-grained partitioning based on exploratory generation enables seven architectures to be processed in just nine time units.*

Proposition 1. *Each bucket i is assigned a GNN training task set, formally defined as $\mathcal{T}_i = \{GNN_{i,1}, GNN_{i,2}, \ldots, GNN_{i,s_i}\}$. If the task allocation is balanced across buckets, i.e.,* $\text{var}(\mathcal{T}) < \beta$, *where* $\text{var}(\mathcal{T})$ *denotes the variance of task distribution, the ring-based migration strategy ensures load balancing throughout the training process.*

Intuition. *Under the ring-based migration scheme, each worker's workload is cyclically transferred to the next worker, while simultaneously receiving workload from the previous worker. Since the partitioning of tasks $\mathcal{T}$ remains constant throughout execution, the variance in incoming and outgoing workloads for each worker is always bounded by β. Therefore, the computational load across all workers remains balanced during the entire ring-based migration and training process.*

Proposition 2. *Let $\mathcal{T}^{(i,j)}$ denote the j^{th} subtask of the i^{th} GNN architecture.* FastGNAS *partitions each GNN task into subtasks of identical computational cost, i.e., for any i and j, $\mathcal{T}^{(i,j)} = C$, where C is a constant. If the coarse-grained partitioning is balanced, i.e.,* $\text{var}(\mathcal{T}) < \beta$, *then the resulting fine-grained partitioning is also guaranteed to be balanced, that is,* $\text{var}(\mathcal{T}_{\text{fg}}) \leq \text{var}(\mathcal{T}) < \beta$.

Intuition. *Suppose GNN architecture training tasks are first partitioned into different buckets using a coarse-grained strategy. Subsequent fine-grained partitioning of each GNN task results in load balance identical to that of the coarse-grained partition, i.e.,* $\text{var}(\mathcal{T}) = \text{var}(\mathcal{T}_{\text{fg}}) < \beta$. *If high-load GNN subtasks are then migrated to workers with lower load and inserted at the bottom of their buckets, load balance can be further enhanced, that is,* $\text{var}(\mathcal{T}_{\text{fg}}) \leq \text{var}(\mathcal{T}) < \beta$.

6 Theoretical Analysis

Since all workers behave identically, we analyze communication from a single worker's perspective. In data-parallel training, the dataset is divided among workers, and inter-worker communication is required at each iteration to synchronize vertex embeddings and gradients. The communication overhead per iteration is $O(\bar{E} \cdot \bar{L} \cdot \bar{K})$, where $\bar{E}$ represents the average number of edges, $\bar{L}$ the average number of GNN layers, and $\bar{K}$ the number of candidate architectures per search. Over T iterations, the total communication complexity is $O(\bar{E} \cdot \bar{L} \cdot \bar{K} \cdot T)$. By defining $\omega = \bar{L} \cdot \bar{K} \cdot T$, this overhead can be succinctly written as $O(\bar{E} \cdot \omega)$.

Furthermore, data-parallel frameworks require model synchronization at every iteration. When AllReduce is employed for parameter synchronization, the communication volume per iteration is $2 \cdot S \cdot \frac{N_{wk}-1}{N_{wk}} \cdot \bar{K} \cdot T$, where S is the model size and N_{wk} is the number of workers. The per-iteration synchronization complexity is therefore $O(\omega)$. As each worker processes only a subgraph in data-parallel frameworks, the memory space complexity per worker is $O(D/N_{wk})$, where D denotes the total dataset size.

In architecture-parallel frameworks, each GNN architecture is trained independently by a single worker, eliminating the need for model synchronization. Consequently, both vertex and model communication, as well as synchronization overhead, are zero. However, every worker must process the complete dataset, resulting in a memory complexity of $O(D)$ per device. As a result, increasing the number of devices does not enhance scalability for large-scale graphs, thereby limiting the applicability of architecture-parallel approaches to extensive graph datasets.

In the hybrid parallel framework based on model migration proposed in this paper, caching mechanisms ensure that vertex communication overhead is eliminated. Model migration during training introduces a model communication cost of $O(S \cdot (N_{wk} - 1))$, where S is typically small because GNN models share parameters across all vertices and model size does not scale with the graph size. Since different workers do not train the same model concurrently, synchronization overhead is avoided. The memory complexity per worker, given caching, is $O(((1 - d^{L+1}) \cdot D)/((1 - d) \cdot N_{wk}))$, where d refers to the node degree.

Table 3. Comparison of complexity.

Para. Type	Data Para.	Arch. Para.	Hybrid Para.
VComm.	$O(\bar{E} \cdot \omega)$	N/A	N/A
MComm.	$O(S \cdot \frac{N_{wk}-1}{N_{wk}} \cdot \bar{K} \cdot T)$	N/A	$O(S \cdot (N_{wk} - 1))$
#Sync.	$O(\omega)$	N/A	N/A
Memo.	$O(D/N_{wk})$	$O(D)$	$O\left(\frac{(1-d^{L+1}) \cdot D}{(1-d) \cdot N_{wk}}\right)$

In summary, the hybrid parallel framework uses L-hop neighborhood caching, which introduces modest storage redundancy but maintains acceptable overall storage overhead. Scalability to large-scale graphs is achieved by increasing the number of devices, thereby overcoming the constraints of architectural parallelism in processing extensive graphs. Moreover, the framework effectively reduces both synchronization and inter-vertex communication overhead, substantially enhancing computational efficiency. Although model transfer between devices is necessary, its cost remains low since the model typically consists of a fixed-size matrix—for example, a layer with 128 input features and 256 hidden units corresponds to a 128×256 matrix. Consequently, communication cost associated with the model does not scale with graph size.

Scalability analysis. Existing GNN models are typically small, having parameter counts that are negligible compared with intermediate embeddings [14, 48]. In this case, the model migration cost is almost negligible. Even with complex models and large datasets, FastGNAS preserves its performance gains. Compared with data-parallel training, which repeatedly synchronizes a shared

model, FastGNAS avoids concurrent updates and requires far fewer transfers, so its benefits remain at scale. Compared with architecture-parallel training, where each worker must store and process the full graph, FastGNAS also scales better because per-device resource limits quickly become a bottleneck.

7 Experiments

7.1 Experimental Setup

GNAS Models. FastGNAS provides a unified programming interface for search algorithms, enabling users to implement custom designs atop the system. In contrast, most existing parallel GNAS execution frameworks are restricted to supporting only a single pre-defined algorithm. To enable a fair comparison with state-of-the-art parallel systems, we reimplement existing GNAS algorithms in FastGNAS and report experimental results using two representative methods: a Shapley value-based algorithm [34] and a reinforcement learning-based algorithm [6].

Datasets. Four commonly used public datasets are employed for experimental evaluation in Table 4. PubMed is a citation network dataset for node classification in medical papers; Flickr is a node classification dataset based on user relationships and interest groups in a social network; OGBN-Arxiv (Arxiv) is a large-scale academic paper citation network for subject area classification; and OGBN-Products (Product) is a large e-commerce product network with millions of nodes and co-purchase relationships for large-scale node classification. These datasets are commonly used in distributed or multi-GPU parallel experiments for GNAS. Since GNAS tasks require simultaneous training of multiple GNN architectures, both memory consumption and computational cost are substantially higher than in standard GNN training. Typically, OGBN-Products is the largest dataset employed in existing distributed and parallel GNAS systems [2, 6, 34, 56].

Table 4. Statistics of datasets.

Dataset	#Node	#Edge	#Feat.	#Class	$d_{\mathbf{avg}}$
PubMed	19,717	44,338	500	3	4.5
Flickr	89,250	899,756	500	7	10.1
Arxiv	169,343	1,166,243	128	40	6.9
Product	2,449,029	61,859,140	100	47	25.3

Baselines. We primarily compare our approach with four state-of-the-art parallel GNAS system baselines. As GNAS-based search methods have been shown to outperform existing GNN algorithms in terms of accuracy, we omit direct accuracy comparisons with conventional GNN approaches for brevity. Existing multi-GPU GNN engines optimize a single architecture across multiple GPUs, so their natural extension to GNAS is one-by-one data-parallel training of candidate architectures. This essentially uses the same framework as NAP-CO/CA. To rule out poor implementation as the cause of NAP-CO/CA's inefficiency, we extend a well-optimized multi-GPU GNN engine, PipeGCN [42], to support GNAS.

- **NAP-CO** is a vertex communication variant of GraphNAP++ [34], requiring GPU-to-GPU communication for non-local neighbors and adopting data parallelism.
- **NAP-CA** is a data-parallel system based on GraphNAP++ [34], enhanced with L-hop neighbor caching. During training, each GPU processes its local data independently, eliminating the need for inter-vertex communication.
- **PAS-PL** employs architecture parallelism [2, 6], assigning each GPU a distinct GNN architecture. Each GPU preloads the entire dataset, avoiding CPU-GPU data transfer.

- **PAS-OD** employs an architecture-parallel framework [2, 6] with on-demand loading to support large-scale datasets, but incurs batch-wise data transfers between the CPU and GPUs.
- **PipeGCN** adopts a data-parallel framework, extended from the PipeGCN engine [42] with its original caching and asynchronization optimizations.

Both NAP-CO and NAP-CA adopt a data-parallel framework in which all workers update a single GNN model jointly, requiring full model synchronization at every training iteration via inter-GPU communication. In addition, NAP-CO employs *vertex-level embedding communication*: after each GNN layer, workers exchange vertex embeddings to obtain the features needed for the next layer. In contrast, NAP-CA caches each vertex's L-hop neighborhood locally on each worker, thereby eliminating cross-worker embedding exchanges. Both are GNAS systems extended from GNN engines.

Experimental environments. All experiments are conducted separately on two multi-GPU servers. **E1:** The first environment consists of eight NVIDIA RTX A6000 GPUs (each with 48 GB memory) and Intel® Xeon® Gold 6326 CPUs at 2.90 GHz. **E2:** The second environment is equipped with two Intel Xeon CPUs, 256 GB RAM, and two NVIDIA Tesla P100 GPUs (each with 16 GB memory). The first environment (**E1**) serves as the primary testbed for our experiments; unless otherwise noted, all reported results are obtained on this system. The second environment (**E2**) is utilized primarily to assess the applicability of FastGNAS, as discussed in the scalability evaluation section. The experiment demonstrates that FastGNAS consistently delivers substantial system performance improvements, even on cost-effective multi-GPU platforms.

Table 5. Comparison of parallel GNAS systems. ***Bold*** marks the best result; *underline* marks the second best. Metrics: Single Architecture Time (ST), Best Architecture Iteration (#BI), Total Training Time (TT), Best Accuracy (BA).

System	Strategy	PubMed				Flickr				Arxiv				Product			
		ST	#BI	TT	BA	ST	#BI	TT	BA	ST	#BI	TT	BA	ST	#BI	TT	BA
NAP-CO	RL	3.95	14	632.02	0.83	31.46	16	5,034.35	0.46	88.11	8	14,097.65	0.72	9,888.29	9	1,582,126.21	0.84
NAP-CA	RL	4.19	17	671.67	0.84	6.13	9	981.49	0.44	9.41	17	1,505.78	0.70	427.26	12	68,361.93	**0.86**
PAS-OD	RL	0.26	11	41.67	0.85	0.71	17	113.11	0.44	1.17	9	186.64	0.71	53.83	14	8,612.15	0.85
FastGNAS	RL	0.20	12	**31.80**	**0.86**	0.30	8	**48.71**	**0.48**	0.67	10	**107.17**	**0.72**	11.32	8	**1,811.20**	0.85
NAP-CO	Shapley	3.99	3	637.76	0.83	31.25	9	4,999.20	0.46	93.79	1	1,500.56	0.72	9,956.12	9	1,652,879.20	0.84
NAP-CA	Shapley	3.72	7	594.93	0.83	7.35	8	1,175.34	0.42	9.36	8	1,497.76	0.71	443.06	9	70,889.84	**0.85**
PAS-OD	Shapley	0.41	6	66.08	0.85	0.89	19	141.60	0.43	1.38	11	221.12	0.72	83.69	12	13,391.04	0.81
FastGNAS	Shapley	0.18	2	**29.15**	**0.87**	0.33	7	**52.56**	**0.48**	0.36	2	**57.56**	**0.72**	13.54	7	**2,166.40**	0.84

Hyperparameter settings. We specify several key hyperparameter settings for GNN architecture training as follows. We set the learning rate to 0.01 and dropout to 0.6, following [10, 16]. The mini-batch size is 1024 (PubMed), 512 (Flickr, Arxiv), and 256 (Product). Load balance is considered acceptable when the normalized load variance is below 0.1 [50]. During search, we evaluate 160 candidate architectures, generating up to 32 per round. For PubMed, we use a 0.8/0.1/0.1 train/validation/test split; for Flickr, Arxiv, and Products, we use the official splits. We apply the same offline degree-based graph partitioning scheme to FastGNAS and all baselines to ensure an initially balanced distribution of data and computation. Additionally, the number of training epochs for each architecture is set to five. After training each candidate for five epochs, candidate selection is performed to identify the optimal architectures.

7.2 Overall Comparison of Architecture Search

Comparison of search results. Table 5 presents comprehensive results for the architecture search across different systems, including search time (ST), the epoch at which the best architecture

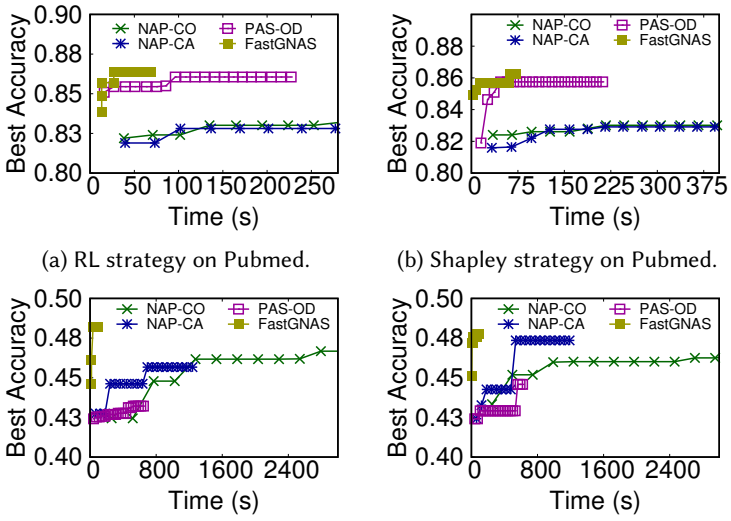
emerges (#BI), total training time (TT), and the highest achieved accuracy(BA). We evaluate and compare four systems utilizing both reinforcement learning (RL) and Shapley value-based search strategies on four datasets. Since PAS-PL requires preloading the entire dataset into GPU memory, its applicability is restricted to small-scale datasets. Furthermore, the data processing capacity of PAS-PL does not scale with the number of GPUs, making it unsuitable for large-scale scenarios. As a result, PAS-PL is excluded from this comparison. A more detailed analysis of PAS-PL, focusing on training time and GPU memory utilization relative to `FastGNAS`, will be provided in a subsequent section.

Experimental results demonstrate that `FastGNAS` consistently identifies optimal architectures more rapidly than competing methods. This improvement is primarily attributable to its exploratory architecture generation strategy, which constructs a larger pool of candidate architectures in each search round, thereby increasing the statistical likelihood of discovering superior architectures earlier. For instance, on the PubMed dataset, `FastGNAS` discovers the optimal architecture in the second round, in contrast to NAP-CO, NAP-CA, and PAS-OD, which identify the best architecture in the third, seventh, and sixth rounds, respectively. Furthermore, `FastGNAS` achieves the lowest single-architecture search time and total training time across all evaluated datasets and search algorithms. This efficiency arises from its hybrid parallel architecture, which alleviates model synchronization overhead during each training iteration. Additionally, its scalability effectively reduces the frequency of CPU-to-GPU data transfers, thereby alleviating related bottlenecks. For example, on the Product dataset, `FastGNAS` requires only 13.54 seconds to train a single architecture, while NAP-CO, NAP-CA, and PAS-OD require 9,956.12 s, 443.06 s, and 83.69 s, respectively. In terms of total search time, `FastGNAS` completes in 2166.40 s, compared to 1,652,879.20 s, 70,889.84 s, and 13,391.04 s for NAP-CO, NAP-CA, and PAS-OD, respectively.

NAP-CO exhibits the lowest efficiency due to frequent inter-GPU communication and embedding synchronization at every layer and iteration during architecture training, causing substantial overhead due to PCIe transmission latency. This overhead exceeds that of model synchronization because: (i) synchronization frequency scales with the number of GNN layers; (ii) the transferred data scales with the number of vertices; and (iii) inter-GPU transfer time dominates GPU computation, so frequent transfers become a major bottleneck for NAP-CO. While NAP-CA avoids layer-wise embedding synchronization, it still incurs overhead through iteration-wise synchronization. In contrast, PAS-OD mitigates frequent synchronization, but its performance is constrained by repeated CPU-to-GPU mini-batch transfers in each training iteration.

Finally, the optimal accuracies achieved by all parallel systems are comparable across different search algorithms and datasets, indicating the correctness of the underlying GNAS algorithm is maintained. Minor accuracy differences among systems stem primarily from variations in data partitioning, mini-batch construction, and the inherent randomness of the architecture search process.

Comparison of search processes. Fig. 8 shows the process of architecture search, with the x-axis indicating elapsed execution time and the y-axis representing the accuracy of the best architecture identified at each time point. The results show that `FastGNAS` consistently discovers the optimal GNN architecture in the shortest time. For instance, `FastGNAS` finds the optimal architecture within 30 seconds, whereas NAP-CO, NAP-CA, and PAS-OD require 2,250.00, 441.36, and 96.56 seconds, respectively. Notably, even when search is terminated upon identifying the optimal architecture, rather than completing evaluation of all candidates, `FastGNAS` remains the most efficient system, achieving the greatest speedup. For example, employing the RL strategy on Flickr, `FastGNAS` completes the full candidate search in 48.71 seconds, compared to 5,034.35, 981.49, and 113.11 seconds for NAP-CO, NAP-CA, and PAS-OD, yielding speedups of 103.35×, 20.15×, and 2.32×, respectively.



(a) RL strategy on Pubmed.

(b) Shapley strategy on Pubmed.

(c) RL strategy on Flickr.

(d) Shapley strategy on Flickr.

Fig. 8. Comparison of architecture search processes.

These results stem from two main factors. First, FastGNAS employs a hybrid architecture that eliminates iteration synchronization overhead and utilizes batch caching to accelerate batch construction. Second, its adaptive exploration reduces the number of search iterations needed to identify the optimal architecture.

In addition, FastGNAS shows no clear runtime dominance between the RL and Shapley settings; their relative speed depends on the training load of the architectures explored, which is influenced by search randomness.

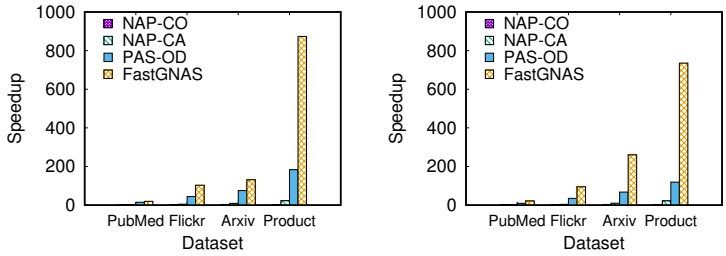
7.3 Speedup Comparison and Time Breakdown

Speedup analysis. Fig. 9 shows the speedup achieved by FastGNAS relative to existing parallel GNAS systems. The slowest baseline, NAP-CO, is used as the reference, and speedup is computed for all other systems with respect to NAP-CO. The results indicate that FastGNAS achieves the highest speedup—up to 874×—as well as the highest average speedup of 282.16× compared to NAP-CO. Moreover, FastGNAS outperforms NAP-CA and PAS-OD, achieving speedups ranging from 14.05× to 32.72× (average: 24.33×) and from 1.31× to 6.18× (average: 3.14×), respectively.

The reduced efficiency of NAP-CO primarily results from frequent synchronization and communication overhead, particularly in transmitting embeddings and gradients over the PCIe bus. NAP-CA is typically slower than PAS-OD, as its data-parallel design requires all workers to simultaneously update a shared model, incurring substantial synchronization costs. While PAS-OD overcomes the scalability limitations of conventional architecture-parallel approaches through on-demand mini-batch loading, this also introduces notable batch transfer overhead between the CPU and GPUs.

Comparison to PipeGCN [42]. Table 6 compares FastGNAS and PipeGCN. The notation is consistent with Table 5. FastGNAS clearly outperforms PipeGCN in both per-epoch time and total training time, while also achieving higher accuracy. For example, on Arxiv, FastGNAS vs. PipeGCN: 1064.54 s vs. 107.17 s in time, and 0.72 vs. 0.71 in accuracy. Specifically, FastGNAS achieves a 3.73–38.73× speedup over PipeGCN, with an average speedup of 15.44×.

This is mainly because PipeGCN is optimized for parallel training of a *single* GNN architecture and cannot exploit GNAS-specific opportunities where multiple architectures are trained concurrently.



(a) RL strategy. (b) Shapley strategy.

Fig. 9. Speedup comparisons over existing systems.

Table 6. Comparison with PipeGCN.

Dataset	System	ST	#BI	TT	BA
PubMed	PipeGCN	0.74 s	13	118.60 s	0.86
	FastGNAS	0.20 s	12	31.80 s	0.86
Flickr	PipeGCN	2.85 s	14	456.70 s	0.44
	FastGNAS	0.30 s	8	48.71 s	0.48
Arxiv	PipeGCN	6.65 s	9	1064.54 s	0.71
	FastGNAS	0.67 s	10	107.17 s	0.72
Product	PipeGCN	438.42 s	12	70144.32 s	0.84
	FastGNAS	11.32 s	8	1811.20 s	0.85

In contrast, FastGNAS is designed for GNAS, with a hybrid parallel framework for multi-architecture training, cross-architecture batch caching and reuse, and fine-grained task scheduling. Consequently, FastGNAS has lower synchronization, batch construction, and transfer overheads, and it achieves better load balancing. Moreover, PipeGCN relies on stale data during training, which can degrade accuracy, so FastGNAS typically achieves higher accuracy.

Time breakdowns. To analyze system bottlenecks and highlight FastGNAS advantages, Table 7 reports a time breakdown. The breakdown includes: remote neighbor communication time (RCT), iteration synchronization time (ITS), model migration time (MMT), batch construction and loading time (BCL), computation time (CTT), and GNN model update time (MUT).

Due to its caching-based framework, FastGNAS eliminates remote neighbor communication time (RCT). Furthermore, its hybrid architecture employs model migration to avoid iteration synchronization (ITS) following batch training. In contrast, NAP-CO requires neighbor embedding communication and synchronization at each GNN layer, which becomes a primary bottleneck. For instance, on PubMed, NAP-CO incurs 0.28 s of remote neighbor communication and 2.91 s for synchronization, whereas the total execution time of FastGNAS is only 0.20 s. Notably, among all evaluated systems, only FastGNAS utilizes a model migration strategy. As a result, migration time is absent in other systems, and the migration overhead in our framework is negligible due to the compact size of GNN models, enabling rapid and efficient migration.

FastGNAS has much lower batch construction and loading time (BCL) than other systems. On Product, FastGNAS takes 11.28 s, versus 36.16 s (NAP-CO), 33.29 s (NAP-CA), and 53.59 s (PAS-OD), mainly due to subgraph preloading on GPU and batch reuse. PAS-OD suffers high BCL from on-demand CPU–GPU transfers. Since no system applies compute optimizations, computation

time (CTT) is similar across methods. Model rotation in FastGNAS slightly increases the number of updates, but their low cost has negligible impact on overall runtime.

Table 7. Average runtime breakdowns (seconds).

System	Dataset	Communication				Computation	
		RCT	*ITS*	*MMT*	*BCL*	*CTT*	*MUT*
NAP-CO	PubMed	0.28	2.91	N/A	0.66	0.11	0.01
	Product	26.54	9,771.80	N/A	36.16	0.40	0.02
NAP-CA	PubMed	N/A	2.60	N/A	0.72	0.13	0.02
	Product	N/A	382.22	N/A	33.29	0.37	0.01
PAS-OD	PubMed	N/A	N/A	N/A	0.19	0.07	0.01
	Product	N/A	N/A	N/A	53.59	0.34	0.01
FastGNAS	PubMed	N/A	N/A	0.02	0.02	0.14	0.02
	Product	N/A	N/A	0.50	11.28	0.30	0.02

7.4 Memory Utilization and Search Time

Table 8 compares execution time and GPU memory usage for FastGNAS, NAP-CO, NAP-CA, PAS-OD, and PAS-PL, emphasizing the contrast between FastGNAS and the preloading-based PAS-PL. FastGNAS attains comparable total time to PAS-PL: on PubMed, 31.83 s vs. 31.09 s; on Product, 1811.2 s vs. 1564.8 s.

However, FastGNAS uses much less GPU memory than PAS-PL: on PubMed: 3.68 GB vs. 4.88 GB and on Product: 35.08 GB vs. 53.82 GB. This reduction comes from storing only one subgraph per GPU, whereas PAS-PL stores the full graph on every GPU.

Although FastGNAS incurs model rotation overhead, the models are small and the cost is minor, and batch reuse further reduces reconstruction overhead. Thus, its execution time is comparable to PAS-PL, which has no model migration. More importantly, FastGNAS scales better in memory and graph size: with more GPUs it can handle larger graphs, whereas PAS-PL's supported graph size is fixed regardless of GPU count.

Table 8. Performance comparison based on total training time (**TT**) and average GPU memory consumption ($\mathbf{M}_{avg}$).

System	PubMed		Product	
	TT (Sec)	M_{avg}(GB)	*TT* (Sec)	M_{avg}(GB)
NAP-CO	632.02	5.98	15,821,26.21	14.05
NAP-CA	671.67	3.23	68,361.93	20.16
PAS-OD	41.67	3.38	8,612.15	5.57
PAS-PL	31.09	4.88	1,564.80	53.82
FastGNAS	31.83	3.68	1,811.20	35.08

7.5 Ablation Study and Scalability Test

Ablation study. Fig. 10 presents the results of the ablation study. In these experiments, *Base* denotes the data-parallel architecture with caching enabled; *+HP* indicates the incorporation of a hybrid parallel architecture with the model migration; *+CACHE* refers to the adoption of batch caching and reuse techniques; and *+BALANCE* represents the use of a task scheduling strategy driven by exploratory task generation.

The results demonstrate that each optimization technique contributes positively to system performance. Notably, the hybrid parallel architecture with model migration yields the most substantial improvement. For example, on the Product dataset, the introduction of hybrid parallelism reduces the search time from 419.63 s to 50.75 s. Subsequent application of cache reuse further

decreases the time to 23.22 s, and integrating the load-balanced scheduling strategy results in a further reduction to 11.32 s.

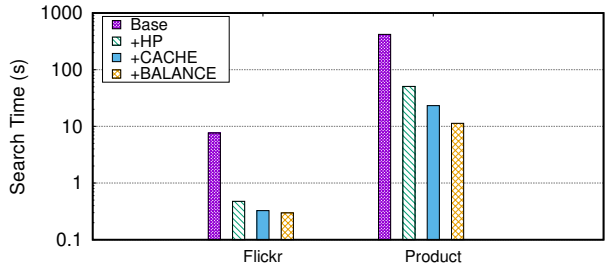


Fig. 10. Ablation study.

Hyperparameter sensitivity analysis. Fig. 11 investigates the sensitivity of FastGNAS to the hyperparameter *epoch*, which specifies the number of training epochs allotted to each GNN architecture prior to evaluation. Other key hyperparameters—including mini-batch size, learning rate, load-balancing coefficient, dataset split ratios, and the number of architectures sampled—are configured according to standard practices in recent literature. The *epoch* value directly influences both the accuracy of architecture evaluation and the overall computational time. As shown in Fig. 11, FastGNAS consistently maintains, and occasionally improves, its acceleration ratio across different epoch settings, indicating robust performance with respect to this hyperparameter.

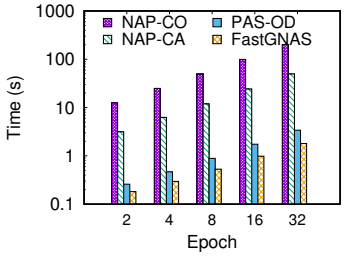


Fig. 11. HP test.

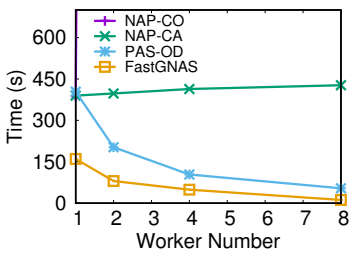


Fig. 12. Scalability.

Scalability test. Fig. 12 presents a comparison of the execution times of different systems on the Product dataset under varying numbers of workers, highlighting the scalability of each systems. As illustrated, FastGNAS achieves an almost linear speedup as the number of workers increases. This result can be attributed to the minimal synchronization overhead required during training, as well as the amortized cost of batch construction through batch reuse. PAS-OD also exhibits decent scalability due to its reduced need for synchronization; however, its execution time is higher than that of FastGNAS, primarily because it performs on-demand batch reading. In contrast, both NAS-CA and NAS-CO suffer from substantial synchronization overhead, resulting in limited scalability. Notably, their execution times can even increase as the number of workers grows. This issue is particularly pronounced for NAS-CO: in addition to excessive synchronization costs, scaling training across multiple GPUs incurs significant communication overhead.

Test on environment E2. Table 9 presents the results on **E2**, demonstrating that FastGNAS consistently achieves superior system performance even on small-scale, cost-effective multi-GPU platforms. As shown in Table 9, under the environment, FastGNAS achieves a speedup of 1.47–2.39× over PAS-OD, 3.14–6.03× over NAP-CA, and 26.01–75.45× over NAP-CO, respectively, demonstrating substantial improvements compared to existing systems.

Table 9. Overall comparison on the **E2** environment.

System	Stra.	PubMed				Flickr			
		ST	#BI	TT	BA	ST	#BI	TT	BA
NAP-CO	RL	11.66	11	1,866.07	0.84	146.50	11	23,440.00	0.47
NAP-CA	RL	14.85	18	432.13	0.83	6.64	10	1,061.69	0.46
PAS-OD	RL	1.07	6	171.20	0.86	3.06	9	490.23	0.46
FastGNAS	RL	0.45	8	**71.74**	**0.86**	2.09	9	**334.10**	**0.47**
NAP-CO	SP	11.40	5	1,824.79	0.84	147.23	13	23,556.80	0.45
NAP-CA	SP	15.15	7	343.58	0.83	6.13	9	980.01	0.48
PAS-OD	SP	0.92	3	146.43	0.86	2.97	11	475.84	0.46
FastGNAS	SP	0.42	3	**67.74**	**0.86**	1.95	7	**312.31**	**0.48**

Profiling results. Table 10 reports profiling metrics that highlight FastGNAS's kernel-level advantages. It achieves the highest GPU/CPU time fraction (about 16% higher than baselines), indicating better GPU utilization via improved load balancing and less idle time. On the CPU side, it has the highest computation fraction (36.67% higher on average), indicating lower synchronization and communication overheads. Batch caching reduces sampling (batch construction) cost, while NAP-CO's low sampling fraction is masked by its high communication and synchronization time. On GPUs, FastGNAS achieves both the highest computation fraction and the lowest data-transfer fraction, exploiting GPU compute better and alleviating data-transfer bottlenecks.

Table 10. Other profiling metrics: GPU/CPU time ratio (P_{gpu}/P_{cpu}), CPU compute (P_{comp}), CPU sampling (P_{csamp}), GPU compute (P_{gcomp}), and GPU transfer (P_{gtrans}).

Metric	$P_{gpu/cpu}$	P_{ccomp}	P_{csamp}	P_{gtrans}	P_{gcomp}
NAP-CO	8.7%	32.8%	7.1%	16.9%	38.9%
NAP-CA	21.1%	33.1%	30.4%	31.8%	24.4%
PAS-OD	24.9%	32.9%	47.4%	17.2%	66.0%
FastGNAS	34.2%	69.6%	7.4%	60.5%	11.0%

Accuracy results under different settings. Table 11 reports the accuracy of FastGNAS with hash-based, metis-based, and degree-based partitioning and with 2, 4, and 8 workers, while comparing with the single-worker (sequential) baseline. The results show that FastGNAS achieves accuracy comparable to the single-worker setting across all configurations. For example, under different configurations, FastGNAS achieves accuracies of 0.719–0.728 compared to 0.721 for the single-worker setting, i.e., only a slight variation of −0.002 to +0.007. In some cases, FastGNAS even outperforms the single-worker accuracy, mainly because exploratory task generation helps discover GNN architectures with higher accuracy. This confirms empirically that FastGNAS is logically consistent with the single-worker algorithm and that varying the partitioning strategy or the number of workers does not affect its accuracy.

7.6 Analysis on Load Estimator

We evaluate the impact of adding different levels of noise to the predicted results on batch processing time (T_{Bat}), time variance (σ), end-to-end time (T_{E2E}), processing time increase ratio ($P_{\Delta T}$), GPU utilization (U_{gpu}), and the imbalance factor ($\mathcal{I}$).

Table 12 reports the impact of injecting noise of ±20%, ±40%, ±80%, and ±160% into the predicted load. As the prediction error increases, the performance of FastGNAS degrades steadily; with ±160%

Table 11. Performance of FastGNAS with different settings.

Dataset	Partition Stra.	N_{wk}=1	N_{wk}=2	N_{wk}=4	N_{wk}=8
PubMed	hash	0.834	0.848	0.852	0.846
	metis		0.853	0.855	0.850
	degree		0.852	0.858	0.854
Arxiv	hash	0.721	0.721	0.724	0.719
	metis		0.725	0.727	0.722
	degree		0.728	0.725	0.723

noise, the performance drops by 45.3%, which confirms the effectiveness of the Load Estimator. Using the predicted load vs. ±20% noise yields 107.17 s vs. 125.66 s (a 17.3% drop), indicating that FastGNAS is insensitive to small noise because scheduling depends mainly on relative load ratios. Larger errors increase the imbalance and reduce GPU utilization: from 0.157 and 91.0% (accurate) to 1.023 and 64.0% (±160% noise).

Table 12. Impact of different noise levels on performance.

Metric	T_{Bat}	σ	T_{E2E}	$P_{\Delta T}$	U_{gpu}	Imbalance
0%	0.075 s	0.011 s	107.17 s	–	91.0%	0.157
20%	0.083 s	0.012 s	125.66 s	+ 10.4%	82.9%	0.381
40%	0.094 s	0.021 s	134.80 s	+ 24.7%	74.1%	0.641
80%	0.101 s	0.025 s	148.27 s	+ 34.7%	68.1%	0.838
160%	0.109 s	0.025 s	155.76 s	+ 45.2%	64.0%	1.023
T_{est}	2.57 s on a single NVIDIA P100 GPU					

In addition, because the collected data has low dimensionality and the Estimator is shallow, the data collection and training overhead is low. On a single NVIDIA P100 GPU, the total collection and training time is thus 2.57 s.

8 Conclusion

To address data management challenges in parallel GNAS on multi-GPU platforms, we propose a parallel GNAS data processing framework with built-in data management optimizations. Specifically, we present FastGNAS, a hybrid framework that integrates data-parallel and architecture-parallel execution to reduce synchronization overhead and enable scalable training on large graphs. Second, we develop data caching and reuse techniques based on incremental batch storage and probabilistic batch reuse to manage redundant mini-batch data. Third, to handle load imbalance, FastGNAS uses fine-grained load scheduling via a Load Estimator and exploratory task decomposition for balanced multi-architecture training. Experiments show that FastGNAS achieves up to 6.18× speedups over state-of-the-art systems without sacrificing search quality. It is of interest to extend FastGNAS to heterogeneous distributed environments for improved scalability.

Acknowledgments

This work is supported by the Shandong Provincial Natural Science Foundation of China (ZR2025QC643), and the National Natural Science Foundation of China (62572108).

References

[1] Chunyu Cao, Xin Ai, Qiange Wang, Yanfeng Zhang, Zhenbo Fu, Hao Yuan, Mingyi Cao, Chaoyi Chen, Yingyou Wen, Yu Gu, et al. 2025. NeutronHeter: Optimizing Distributed Graph Neural Network Training for Heterogeneous Clusters. *Proceedings of the ACM on Management of Data* 3, 4 (2025), 1–29.

[2] Jiamin Chen, Jianliang Gao, Yibo Chen, Moctard Babatounde Oloulade, Tengfei Lyu, and Zhao Li. 2021. Graphpas: Parallel architecture search for graph neural networks. In *Proceedings of the 44th International ACM SIGIR Conference on Research and Development in Information Retrieval*. 2182–2186.
[3] Guosheng Feng, Hongzhi Wang, and Chunnan Wang. 2023. Search for deep graph neural networks. *Information Sciences* 649 (2023), 119617.
[4] Yunjun Gao, Xiaoze Liu, Junyang Wu, Tianyi Li, Pengfei Wang, and Lu Chen. 2022. ClusterEA: Scalable Entity Alignment with Stochastic Training and Normalized Mini-batch Similarities. In *KDD*. 421–431.
[5] Yang Gao, Hong Yang, Peng Zhang, Chuan Zhou, and Yue Hu. 2019. Graphnas: Graph neural architecture search with reinforcement learning. *arXiv:1904.09981* (2019).
[6] Yang Gao, Peng Zhang, Hong Yang, Chuan Zhou, Yue Hu, Zhihong Tian, Zhao Li, and Jingren Zhou. 2022. GraphNAS++: Distributed architecture search for graph neural networks. *IEEE Transactions on Knowledge and Data Engineering* 35, 7 (2022), 6973–6987.
[7] Marcos Paulo Silva Gôlo, Marcelo Isaias De Moraes, Rudinei Goularte, and Ricardo Marcondes Marcacini. 2023. On the use of early fusion operators on heterogeneous graph neural networks for one-class learning. In *Proceedings of the 29th Brazilian Symposium on Multimedia and the Web*. 128–136.
[8] Chaoyu Guan, Xin Wang, Hong Chen, Ziwei Zhang, and Wenwu Zhu. 2022. Large-scale graph neural architecture search. In *International Conference on Machine Learning*. 7968–7981.
[9] Chaoyu Guan, Xin Wang, and Wenwu Zhu. 2021. Autoattend: Automated attention representation search. In *ICML*. 3864–3874.
[10] Will Hamilton, Zhitao Ying, and Jure Leskovec. 2017. Inductive representation learning on large graphs. *Advances in neural information processing systems* 30 (2017).
[11] Jindong Han, Weijia Zhang, Hao Liu, Tao Tao, Naiqiang Tan, and Hui Xiong. 2024. Bigst: Linear complexity spatio-temporal graph neural network for traffic forecasting on large-scale road networks. *Proceedings of the VLDB Endowment* 17, 5 (2024), 1081–1090.
[12] Zhao Huan, Yao Quanming, and TU Weiwei. 2021. Search to aggregate neighborhood for graph neural network. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. 552–563.
[13] Zijian Huang, Meng-Fen Chiang, and Wang-Chien Lee. 2022. LinE: Logical Query Reasoning over Hierarchical Knowledge Graphs. In *KDD*. 615–625.
[14] Zhihao Jia, Sina Lin, and Mingyu et al. Gao. 2020. Improving the accuracy, scalability, and performance of graph neural networks with roc. *Proceedings of Machine Learning and Systems* 2 (2020), 187–198.
[15] Qisheng Jiang, Lei Jia, and Chundong Wang. 2024. Reducing memory contention and i/o congestion for disk-based gnn training. *arXiv:2406.13984* (2024).
[16] TN Kipf. 2016. Semi-supervised classification with graph convolutional networks. *arXiv:1609.02907* (2016).
[17] Lingbai Kong, Hanchen Yang, Wengen Li, Yichao Zhang, Jihong Guan, and Shuigeng Zhou. 2024. Traffexplainer: A Framework Toward GNN-Based Interpretable Traffic Prediction. *IEEE Transactions on Artificial Intelligence* 6, 3 (2024), 559–573.
[18] Lingbai Kong, Hanchen Yang, Wengen Li, Yichao Zhang, Jihong Guan, and Shuigeng Zhou. 2024. Traffexplainer: A Framework Toward GNN-Based Interpretable Traffic Prediction. *IEEE Transactions on Artificial Intelligence* 6, 3 (2024), 559–573.
[19] Kwei-Herng Lai, Daochen Zha, Kaixiong Zhou, and Xia Hu. 2020. Policy-gnn: Aggregation optimization for graph neural networks. In *Proceedings of the 26th ACM SIGKDD international conference on knowledge discovery & data mining*. 461–471.
[20] Chao Li, Hao Xu, and Kun He. 2023. Differentiable meta multigraph search with partial message propagation on heterogeneous information networks. In *Proceedings of the AAAI conference on artificial intelligence*, Vol. 37. 8518–8526.
[21] Mingyi Li, Junmin Xiao, Kewei Zhang, Zhiheng Lin, Chaoyang Shui, Ke Meng, Zehua Wang, Yunfei Pang, and Guangming Tan. 2024. A Coordinated Strategy for GNN Combining Computational Graph and Operator Optimizations. In *Proceedings of the 38th ACM International Conference on Supercomputing*. 460–472.
[22] Tianyi Li, Lu Chen, Christian S Jensen, and Torben Bach Pedersen. 2021. TRACE: Real-time compression of streaming trajectories in road networks. *Proceedings of the VLDB Endowment* 14, 7 (2021), 1175–1187.
[23] Tianyi Li, Ruikai Huang, Lu Chen, Christian S Jensen, and Torben Bach Pedersen. 2020. Compression of Uncertain Trajectories in Road Networks. *Proc. VLDB Endow.* 13, 7 (2020), 1050–1063.
[24] Yaoman Li and Irwin King. 2020. Autograph: Automated graph neural network. In *International conference on neural information processing*. 189–201.
[25] Yanxi Li, Zean Wen, Yunhe Wang, and Chang Xu. 2021. One-shot graph neural architecture search with dynamic search space. In *Proceedings of the AAAI conference on artificial intelligence*, Vol. 35. 8510–8517.
[26] Yi-Chien Lin, Gangda Deng, and Viktor Prasanna. 2024. A unified cpu-gpu protocol for gnn training. In *Proceedings of the 21st ACM International Conference on Computing Frontiers*. 155–163.

[27] Zhiqi Lin, Cheng Li, Youshan Miao, Yunxin Liu, and Yinlong Xu. 2020. Pagraph: Scaling gnn training on large graphs via computation-aware caching. In *Proceedings of the 11th ACM Symposium on Cloud Computing*. 401–415.
[28] Vasco Lopes, Fabio Maria Carlucci, Pedro M Esperança, Marco Singh, Antoine Yang, Victor Gabillon, Hang Xu, Zewei Chen, and Jun Wang. 2024. MANAS: Multi-agent neural architecture search. *Machine Learning* 113, 1 (2024), 73–96.
[29] Qing Lu, Weiwen Jiang, Meng Jiang, Jingtong Hu, Sakyasingha Dasgupta, and Yiyu Shi. 2020. Fgnas: Fpga-aware graph neural architecture search. (2020).
[30] Ziyue Luo, Yixin Bao, and Chuan Wu. 2024. Optimizing Task Placement and Online Scheduling for Distributed GNN Training Acceleration in Heterogeneous Systems. *IEEE/ACM Transactions on Networking* 32, 5 (2024), 3715–3729.
[31] Yuxin Ma, Ping Gong, Tianming Wu, Jiawei Yi, Chengru Yang, Cheng Li, Qirong Peng, Guiming Xie, Yongcheng Bao, Haifeng Liu, et al. 2024. Eliminating Data Processing Bottlenecks in GNN Training over Large Graphs via Two-level Feature Compression. *Proceedings of the VLDB Endowment* 17, 11 (2024), 2854–2866.
[32] Sudipta Mondal, Susmita Dey Manasi, Kishor Kunal, Ramprasath S, and Sachin S Sapatnekar. 2022. GNNIE: GNN inference engine with load-balancing and graph-specific caching. In *Proceedings of the 59th ACM/IEEE Design Automation Conference*. 565–570.
[33] Samir Moustafa, Nils Kriege, and Wilfried N Gansterer. 2025. Efficient Mixed Precision Quantization in Graph Neural Networks. In *2025 IEEE 41st International Conference on Data Engineering (ICDE)*. 4038–4052.
[34] Babatounde Moctard Oloulade, Jianliang Gao, Jiamin Chen, Raeed Al-Sabri, Zhenpeng Wu, and Monir Abdullah. 2025. Shapley-guided pruning for efficient graph neural architecture prediction in distributed learning environments. *Information Sciences* 695 (2025), 121695.
[35] Jeongmin Brian Park, Kun Wu, Vikram Sharma Mailthody, Zaid Quresh, Scott Mahlke, and Wen-mei Hwu. 2024. Lsm-gnn: Large-scale storage-based multi-gpu gnn training by optimizing data transfer scheme. *arXiv:2407.15264* (2024).
[36] Shashank Sheshar Singh, Samya Muhuri, Shivansh Mishra, Divya Srivastava, Harish Kumar Shakya, and Neeraj Kumar. 2024. Social network analysis: A survey on process, tools, and application. *ACM computing surveys* 56, 8 (2024), 1–39.
[37] Zhen Song, Yu Gu, Tianyi Li, Qing Sun, Yanfeng Zhang, Christian S Jensen, and Ge Yu. 2023. ADGNN: towards scalable GNN training with aggregation-difference aware sampling. *Proceedings of the ACM on Management of Data* 1, 4 (2023), 1–26.
[38] Zhen Song, Yu Gu, Jianzhong Qi, Zhigang Wang, and Ge Yu. 2022. EC-Graph: A distributed graph neural network system with error-compensated compression. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*. 648–660.
[39] Zhen Song, Yu Gu, Qing Sun, Tianyi Li, Yanfeng Zhang, Yushuai Li, Christian S Jensen, and Ge Yu. 2024. DynaHB: A Communication-Avoiding Asynchronous Distributed Framework with Hybrid Batches for Dynamic GNN Training. *Proceedings of the VLDB Endowment* 17, 11 (2024), 3388–3401.
[40] Yongye Su, Yinqi Sun, Minjia Zhang, and Jianguo Wang. 2024. Vexless: A serverless vector data management system using cloud functions. *Proceedings of the ACM on Management of Data* 2, 3 (2024), 1–26.
[41] Jie Sun, Li Su, Zuocheng Shi, Wenting Shen, Zeke Wang, Lei Wang, Jie Zhang, Yong Li, Wenyuan Yu, Jingren Zhou, et al. 2023. Legion: Automatically pushing the envelope of Multi-GPU system for Billion-Scale GNN training. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*. 165–179.
[42] Cheng Wan, Youjie Li, Cameron R Wolfe, Anastasios Kyrillidis, Nam Sung Kim, and Yingyan Lin. 2022. Pipegcn: Efficient full-graph training of graph convolutional networks with pipelined feature communication. *arXiv:2203.10428* (2022).
[43] Conghao Wang, Gaurav Asok Kumar, and Jagath C Rajapakse. 2025. Drug discovery and mechanism prediction with explainable graph neural networks. *Scientific Reports* 15, 1 (2025), 179.
[44] Gang Wang, Yanfeng Zhang, Chenhao Ying, Xiaohua Lit, and Ge Yu. 2024. Hammer: A general blockchain evaluation framework. In *2024 IEEE 44th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 391–402.
[45] Jihe Wang, Ying Wu, and Danghui Wang. 2024. SC-GNN: A Communication-Efficient Semantic Compression for Distributed Training of GNNs. In *Proceedings of the 61st ACM/IEEE Design Automation Conference*. 1–6.
[46] Yuke Wang, Boyuan Feng, Zheng Wang, Tong Geng, Kevin Barker, Ang Li, and Yufei Ding. 2023. MGG: Accelerating graph neural networks with Fine-Grained Intra-Kernel Communication-Computation pipelining on Multi-GPU platforms. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. 779–795.
[47] Zhili Wang, Shimin Di, and Lei Chen. 2021. Autogel: An automated graph neural network with explicit link information. *Advances in Neural Information Processing Systems* 34 (2021), 24509–24522.
[48] Meng Wu, Mingyu Yan, and Wenming et al. Li. 2025. Survey on Characterizing and Understanding GNNs From a Computer Architecture Perspective. *IEEE Transactions on Parallel and Distributed Systems* (2025).
[49] Wenchao Wu, Xuanhua Shi, Ligang He, and Hai Jin. 2023. TurboMGNN: Improving concurrent GNN training tasks on GPU with fine-grained kernel fusion. *IEEE Transactions on Parallel and Distributed Systems* 34, 6 (2023), 1968–1981.
[50] Chenzhong Xu and Francis CM Lau. 2007. *Load balancing in parallel computers: theory and practice*. Vol. 381. Springer.

[51] Ahmad Zareie and Rizos Sakellariou. 2024. Fuzzy influence maximization in social networks. *ACM Transactions on the Web* 18, 3 (2024), 1–28.

[52] Xin Zeng, Peng-Kun Feng, Shu-Juan Li, Shuang-Qing Lv, Meng-Liang Wen, and Yi Li. 2024. GNN-DDAS: Drug discovery for identifying anti-schistosome small molecules based on graph neural network. *Journal of Computational Chemistry* 45, 32 (2024), 2825–2834.

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

[54] Hengrui Zhang, Zhongming Yu, Guohao Dai, Guyue Huang, Yufei Ding, Yuan Xie, and Yu Wang. 2022. Understanding gnn computational graph: A coordinated computation, io, and memory perspective. *Proceedings of Machine Learning and Systems* 4 (2022), 467–484.

[55] Odin Zhang, Haitao Lin, Xujun Zhang, Xiaorui Wang, Zhenxing Wu, Qing Ye, Weibo Zhao, Jike Wang, Kejun Ying, Yu Kang, et al. 2025. Graph Neural Networks in Modern AI-aided Drug Discovery. *Chemical Reviews* (2025).

[56] Wentao Zhang, Yu Shen, Zheyu Lin, Yang Li, Xiaosen Li, Wen Ouyang, Yangyu Tao, Zhi Yang, and Bin Cui. 2022. Pasca: A graph neural architecture search system under the scalable paradigm. In *Proceedings of the ACM Web Conference 2022*. 1817–1828.

[57] Zhe Zhang, Ziyue Luo, and Chuan Wu. 2023. Two-level graph caching for expediting distributed gnn training. In *IEEE INFOCOM 2023-IEEE Conference on Computer Communications*. 1–10.

[58] Zhen Zhang, Chen Xu, Kun Liu, Shaohua Xu, and Long Huang. 2024. A resource optimization scheduling model and algorithm for heterogeneous computing clusters based on GNN and RL. *Journal of Supercomputing* 80, 16 (2024).

[59] Zeyang Zhang, Ziwei Zhang, Xin Wang, Yijian Qin, Zhou Qin, and Wenwu Zhu. 2023. Dynamic heterogeneous graph attention neural architecture search. In *Proceedings of the AAAI conference on artificial intelligence*, Vol. 37. 11307–11315.

[60] Huan Zhao, Lanning Wei, and Quanming Yao. 2020. Simplifying architecture search for graph neural network. *arXiv:2008.11652* (2020).

[61] Yiren Zhao, Duo Wang, Daniel Bates, Robert Mullins, Mateja Jamnik, and Pietro Lio. 2020. Learned low precision graph neural networks. *arXiv:2009.09232* (2020).

[62] Yiren Zhao, Duo Wang, Xitong Gao, Robert Mullins, Pietro Lio, and Mateja Jamnik. 2020. Probabilistic dual network architecture search on graphs. *arXiv:2003.09676* (2020).

[63] Da Zheng, Chao Ma, Minjie Wang, Jinjing Zhou, Qidong Su, Xiang Song, Quan Gan, Zheng Zhang, and George Karypis. 2020. DistDGL: Distributed graph neural network training for billion-scale graphs. In *2020 IEEE/ACM 10th Workshop on Irregular Applications: Architectures and Algorithms (IA3)*. 36–44.

[64] Kaixiong Zhou, Xiao Huang, Qingquan Song, Rui Chen, and Xia Hu. 2022. Auto-gnn: Neural architecture search of graph neural networks. *Frontiers in big Data* 5 (2022), 1029307.

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

Received October 2025; revised January 2026; accepted February 2026