

Multi-GPU Parallelization Strategies for Large-Scale Graph Learning

Kamila Ploskonka

Department of Computer Science, Colorado State University, Fort Collins, CO, USA.
kamilaploskonka408@colostate.edu

Dorotea Millan

Department of Computer Science, University of New Hampshire, Durham, NH, USA.
hellodorotea@unh.edu

Roksana Franc

School of Computing, Clemson University, Clemson, SC, USA.
rfranc@clemson.edu

Abstract

Large-scale graph learning has become a central computational workload in artificial intelligence because graphs can represent social networks, transportation systems, biological pathways, supply chains, financial transactions, and scientific simulations. As graph sizes and model depths increase, single-device training rapidly becomes impractical, making multi-GPU parallelism a necessary but structurally difficult design space. This paper examines multi-GPU parallelization strategies for large-scale graph learning from a systems perspective, emphasizing architectural trade-offs, data movement, memory hierarchy, scheduling, robustness, fairness, sustainability, deployment, and governance. The analysis distinguishes between graph partitioning, feature partitioning, model parallelism, pipeline execution, tensor parallel decomposition, and hybrid scheduling, showing that each strategy redistributes bottlenecks rather than eliminating them. It argues that the most consequential decisions are not merely about scaling throughput but about aligning graph topology, communication topology, memory capacity, and organizational incentives. The paper also considers failure management, reproducibility, energy consumption, and equitable access to infrastructure. Through cross-domain comparisons with recommender systems, scientific computing, and language model training, it identifies persistent tensions among locality, load balance, staleness, and scalability. The conclusion suggests that future multi-GPU graph learning systems must treat parallelism as a socio-technical design problem, not only a kernel optimization problem. Robust progress will depend on co-designed algorithms, hardware-aware schedulers, transparent benchmarking, and governance frameworks that make large-scale graph learning both efficient and accountable.

Keywords

multi-GPU systems, graph neural networks, parallelization, distributed training, graph partitioning, memory hierarchy, sustainability, infrastructure governance.

1. Introduction

Graph learning has evolved from early recurrent graph neural models into a broad family of architectures used for node classification, link prediction, graph classification, recommendation, fraud detection, and scientific discovery [1], [2], [3], [4]. Unlike image or

sequence data, graphs are irregular. Their memory access patterns are sparse, their degree distributions are often highly skewed, and their computational density varies sharply across neighborhoods. These properties make graph learning difficult to accelerate even on a single GPU. When the graph grows to billions of edges or the model requires many layers and extensive feature transformations, multi-GPU execution becomes unavoidable. Yet adding GPUs does not automatically produce proportional gains. It introduces partitioning costs, inter-device communication, synchronization delays, memory duplication, and complex failure modes.

The central challenge is structural rather than incidental. A graph learning system must decide how to divide graph topology, node features, edge features, model parameters, and intermediate activations across devices. Each division changes the communication pattern. For example, partitioning nodes by community can reduce cross-device edges, but it may create severe load imbalance if some partitions contain high-degree hubs. Partitioning edges randomly may improve balance but increases remote feature fetches. Replicating popular nodes can reduce communication but consumes memory. In practice, no single strategy dominates across all graphs, models, and hardware interconnects [5], [6]. The best design depends on graph density, degree distribution, batch construction, model depth, GPU memory, interconnect bandwidth, and the economic or operational constraints of the deployment environment.

This paper argues that multi-GPU graph learning should be understood as a socio-technical infrastructure problem. The technical layer involves kernels, memory allocators, collective communication libraries, graph partitioners, and schedulers. The organizational layer involves who can access large clusters, how workloads are prioritized, how failures are handled, and how energy and cost are accounted for. A system that achieves high throughput on a benchmark but cannot be reproduced, governed, or sustained in production is not a complete solution. The sections that follow analyze conceptual foundations, hardware architecture, parallelization strategies, data movement, scheduling, robustness, fairness, sustainability, deployment, and future directions.

2. Conceptual Foundations of Large-Scale Graph Learning

Graph learning models typically combine neighborhood aggregation with learned transformations. Early formulations established the idea that nodes can iteratively exchange information with neighbors until representations stabilize [1]. Later work simplified and scaled this idea through localized spectral filtering and convolutional aggregation [2], [6]. Attention-based aggregation further allowed models to weight neighbors adaptively, improving expressiveness at additional computational cost [3]. Inductive methods such as neighborhood sampling made it possible to train on unseen nodes and large graphs by limiting the size of each computation graph [4]. These algorithmic choices directly affect parallelization. Full-neighbor aggregation creates dense communication between partitions, while sampling reduces communication but introduces variance and may require careful seed management for reproducibility.

At large scale, graph learning is often memory-bound rather than compute-bound. Feature matrices may be enormous, embeddings can be high-dimensional, and intermediate activations for deep models can exceed GPU memory. The training process must therefore manage a hierarchy that includes high-bandwidth memory on each GPU, host memory, local non-volatile storage, and remote storage. Graph partitioning libraries and distributed platforms attempt to keep frequently accessed data close to the compute device while

minimizing cross-device traffic [15], [18]. However, graph topology is rarely uniform. Power-law degree distributions mean that a small number of nodes participate in a large fraction of edges. These hubs become communication hot spots. They can also cause memory hotspots if their features and adjacent edges are replicated across devices.

Surveys of graph neural networks emphasize that scalability is not a single property but a combination of algorithmic scalability, memory scalability, and system scalability [19], [20]. Algorithmic scalability concerns whether the model can learn effectively as depth and neighborhood size grow. Memory scalability concerns whether parameters, features, and activations fit within the available hierarchy. System scalability concerns whether adding devices reduces training time or increases throughput. These dimensions interact. A sampling algorithm may improve memory scalability but worsen statistical efficiency. A partitioning strategy may improve system scalability but reduce model accuracy if it cuts too many edges. A multi-GPU system must therefore expose trade-offs to users rather than hiding them behind a single configuration.

3. Hardware and System Architecture for Multi-GPU Graph Training

Multi-GPU graph training inherits many architectural ideas from large-scale deep learning. Data parallelism replicates the model across devices and splits the training batch, which is effective when the model fits on one GPU and communication is limited to gradient synchronization [7]. Model parallelism partitions layers or parameter tensors across devices, which is necessary when a single model cannot fit in memory [8]. Pipeline parallelism assigns different layers or stages to different devices and streams micro-batches through the pipeline to improve utilization [7]. Tensor parallelism splits individual operations across devices, requiring frequent synchronization but enabling very large layers [8]. Memory optimization techniques such as partitioned optimizer states and gradient sharding reduce duplication and allow larger models to be trained on the same hardware [9]. These techniques are not mutually exclusive; production systems often combine them.

Graph learning complicates these standard approaches because the data itself is structured. In data parallelism, each GPU may receive a set of seed nodes and their sampled neighborhoods. If neighborhoods overlap, devices fetch or compute the same embeddings multiple times. If sampling is independent, each device may need remote features from nodes owned by other devices. This creates an all-to-all communication pattern that is less regular than gradient synchronization. In model parallelism, splitting layers across GPUs may be feasible, but the adjacency-dependent aggregation still requires access to graph structure. A layer partition cannot simply own a subset of neurons; it must also coordinate with the partition that owns the relevant edges and features. Pipeline parallelism can hide some latency, but graph sampling and aggregation stages may have variable durations, causing pipeline bubbles.

Hardware topology matters. NVLink, PCIe, InfiniBand, and Ethernet provide different bandwidth and latency profiles. A partitioning strategy that performs well on a fully connected NVLink domain may fail on a multi-node cluster with slower interconnects. Similarly, GPUs with different memory capacities and cache hierarchies may require different replication policies. The system must be aware of the physical placement of graph partitions, not just logical assignment. Modern graph learning frameworks attempt to abstract this complexity, but abstractions can obscure performance-critical details [12], [13]. A robust multi-GPU system must therefore provide tunable partitioning, communication, and placement policies rather than a one-size-fits-all configuration.

4. Parallelization Strategies and Structural Trade-offs

The most common strategy for large graphs is partition-based training. Nodes or edges are divided among GPUs, and each device computes embeddings for its assigned elements. Cross-partition edges trigger remote feature fetches or message passing. The quality of the partition determines the communication volume. Metis-style multilevel partitioning has long been used to reduce edge cuts in irregular graphs [16]. However, minimizing edge cuts is not always equivalent to minimizing runtime. A partition with few cut edges may still be imbalanced if it concentrates high-degree nodes. A balanced partition may have many cut edges but better GPU utilization. Advanced systems use hybrid partitioning, caching, and replication to handle hubs. They may replicate high-degree nodes across devices so that neighbors can access them locally, accepting extra memory cost to reduce communication.

Another strategy is sampling-based parallel training. Each GPU samples a subgraph for a mini-batch, and the model aggregates only sampled neighbors. This reduces memory and communication per step. However, sampling can increase variance and may require careful coordination to avoid duplicate work. If different GPUs sample overlapping neighborhoods, the same node embeddings may be recomputed. The system can cache embeddings, but cache coherence becomes a challenge when parameters are updated. Stale embeddings can accelerate training but may reduce accuracy or slow convergence. Sampling also interacts with batching. Random sampling may create balanced workloads, while importance sampling may target high-degree nodes but create hotspots. The design space is therefore not only about parallel execution but about the statistical properties of the training process.

Model-level parallelism offers another path. Layers can be distributed across GPUs, with activations communicated between stages. For graph models, this approach can be combined with graph partitioning. A device may own a subset of nodes and also be responsible for a subset of layers. This hybrid design reduces the memory burden of deep models but increases synchronization. Recent kernel-level work shows that specialized GPU kernels can substantially accelerate graph neural network training by exploiting sparsity and memory access patterns [17]. Even so, kernel efficiency alone cannot resolve partitioning imbalances or inter-device communication. The most effective systems treat kernels, partitioners, and communication schedulers as co-designed components. A fast kernel on an imbalanced partition may still wait for stragglers, while a slower kernel on a well-balanced partition may achieve better end-to-end throughput.

Tensor parallelism and pipeline parallelism are less common in graph learning than in language model training, but they are increasingly relevant as graph models grow. Tensor parallelism can split large feature transformations across GPUs, while pipeline parallelism can overlap aggregation and transformation stages. The difficulty is that graph aggregation has irregular dependencies. A pipeline stage may depend on neighbors that reside in multiple partitions, forcing cross-stage communication. If the graph is dynamic or the batch is sampled, the dependency pattern changes at every step. Static pipeline schedules may therefore be inefficient. Dynamic scheduling can adapt to workload variation but adds control overhead and makes reproducibility harder. The structural trade-off is between predictability and efficiency.

5. Data Movement, Memory Hierarchy, and Communication Governance

In multi-GPU graph learning, data movement is often the dominant cost. Feature fetching, embedding exchange, gradient synchronization, and partition metadata all consume

bandwidth. The memory hierarchy adds further complexity. GPU memory is fast but limited; host memory is larger but slower; storage is larger still but far slower. Systems must decide what to cache, what to replicate, and what to fetch on demand. Caching high-degree nodes can reduce remote access, but cache invalidation becomes difficult when embeddings change. Replicating features can reduce communication but increases memory pressure. Sharding optimizer states and gradients can reduce memory duplication but requires collective communication [9]. The design must balance these competing pressures.

Communication governance is both a technical and organizational issue. In a shared cluster, multiple jobs compete for interconnect bandwidth. A graph learning job that aggressively uses all-to-all communication can degrade other workloads. Cluster schedulers may therefore impose bandwidth quotas, priority classes, or topology-aware placement. Within a job, the system may use hierarchical communication: first aggregate within a node, then across nodes. This reduces cross-node traffic but requires the partitioner to align graph communities with hardware topology. If graph communities and network topology are misaligned, the system pays unnecessary cross-node costs. Topology-aware partitioning is therefore a form of infrastructure governance as much as an optimization.

Memory pressure also affects robustness. When a GPU runs out of memory, training may fail or fall back to slower paths. Out-of-memory errors are particularly problematic in graph learning because memory usage can spike unpredictably due to high-degree nodes or long sampled neighborhoods. Systems can mitigate this through gradient checkpointing, activation recomputation, and adaptive sampling. However, these techniques trade computation for memory and may reduce throughput. They also complicate reproducibility because the effective batch composition changes. A transparent system should report memory policies alongside accuracy and speed. Otherwise, comparisons between papers or deployments may be misleading.

6. Scheduling, Robustness, and Failure Management

Scheduling in multi-GPU graph learning must handle irregular workloads. Unlike dense matrix multiplication, graph aggregation has variable cost per node. Some nodes have few neighbors and finish quickly; others have thousands of neighbors and become stragglers. Static scheduling may assign equal numbers of nodes to each GPU but unequal work. Dynamic scheduling can reassign work, but it adds coordination overhead and may increase communication. A practical approach is to combine graph partitioning with work stealing or priority scheduling for high-degree nodes. The scheduler must also consider memory availability, because a GPU with less free memory cannot accept large neighborhoods.

Robustness is a first-order concern for long-running training jobs. Large clusters experience hardware failures, network congestion, and software faults. Checkpointing is standard, but graph learning checkpoints are large and may include graph partitions, features, model parameters, optimizer states, and sampler states. Restoring a consistent state across devices is non-trivial. If one device fails, the system must reassign its partitions without losing progress. This may require repartitioning the graph, which can be expensive. Elastic training systems can add or remove devices, but graph partitions are harder to rebalance than dense model shards. The system must therefore plan for failure recovery from the beginning, not as an afterthought.

Reproducibility is closely related to robustness. Graph sampling introduces randomness, and parallel execution changes the order of aggregation. Floating-point reductions may differ

across devices, producing slightly different results. For scientific and regulated applications, small differences can matter. A multi-GPU system should provide deterministic modes when needed, even if they reduce performance. It should record the random seed, partition assignment, communication schedule, and hardware topology. Without such records, debugging and auditing become difficult. Robustness therefore includes not only staying alive but also producing trustworthy and explainable results.

7. Fairness, Sustainability, and Policy Implications

Large-scale graph learning consumes significant energy and hardware resources. Training a deep graph model on a multi-GPU cluster can require substantial electricity, cooling, and capital investment. This creates sustainability concerns. The most efficient system is not always the one with the highest peak throughput; it may be the one that achieves acceptable accuracy with lower energy per useful result. Scheduling policies can shift flexible jobs to times or regions with cleaner energy. Hardware sharing can improve utilization. Algorithmic choices such as smaller embeddings, sparse attention, and adaptive sampling can reduce energy. However, these choices may affect accuracy and fairness. A model trained with aggressive sampling may underperform on minority subgroups or low-degree nodes. Efficiency and equity must therefore be evaluated together.

Fairness in graph learning is also a data and infrastructure issue. Graph datasets often encode social, economic, or political relations. If a multi-GPU system samples or partitions the graph in ways that underrepresent certain communities, the resulting model may amplify bias. For example, if high-degree nodes are cached and low-degree nodes are frequently dropped, the model may learn better representations for already privileged nodes. Similarly, if some regions or organizations lack access to large GPU clusters, they cannot train competitive models. The governance of graph learning infrastructure thus affects who can participate in AI development. Policy frameworks may need to address data access, compute allocation, and transparency in large-scale graph training.

Sustainability also involves the lifecycle of hardware. GPUs have embodied carbon costs from manufacturing and transport. Extending their useful life through efficient scheduling and multi-tenant sharing can reduce environmental impact. However, sharing raises security and privacy concerns. Graph data may contain sensitive relationships. Multi-GPU systems must isolate jobs, protect feature stores, and prevent side-channel leakage. These requirements add overhead and may reduce performance. A responsible system design must make these trade-offs explicit. It should provide users with information about energy use, data access, and fairness metrics, not only throughput and latency.

8. Deployment Models and Cross-Domain Comparisons

Deployment models for multi-GPU graph learning range from on-premises clusters to cloud instances and edge-cloud hybrids. On-premises clusters offer control over topology and data locality but require capital investment and operational expertise. Cloud environments offer elasticity but can suffer from variable interconnect performance and data transfer costs. Hybrid deployments may keep sensitive graph data on-premises while using cloud GPUs for burst training. Each model changes the parallelization strategy. In a cloud setting, topology-aware partitioning may be less predictable because the physical placement of GPUs is abstracted. In an on-premises setting, the system can exploit known network topology. Edge-cloud hybrid systems may partition the graph so that local subgraphs are processed near data

sources, while global aggregation occurs in the cloud. This reduces bandwidth but increases synchronization complexity.

Cross-domain comparisons reveal both similarities and differences. Recommender systems use graph learning on user-item interaction graphs, often with billions of edges [5]. They prioritize low-latency inference and frequent updates. Scientific computing uses graphs for molecular structures, particle simulations, and climate models. These workloads may prioritize accuracy and reproducibility over latency. Financial fraud detection uses transaction graphs where temporal order and privacy are critical. The same multi-GPU strategy may not work across all domains. For example, a recommender system may tolerate stale embeddings to achieve higher throughput, while a scientific application may require deterministic updates. A financial system may need strict audit trails and access controls. The deployment model must therefore be co-designed with domain requirements.

Frameworks and platforms play a mediating role. PyTorch Geometric and Deep Graph Library provide abstractions for graph operations and distributed training [12], [13]. DistDGL and AliGraph offer specialized distributed graph storage and sampling [14], [15]. These platforms lower the barrier to entry but also shape what users can easily express. If a platform optimizes for one partitioning strategy, users may adopt it even when another would be better for their graph. Platform governance therefore matters. Open standards for graph partitioning, sampling, and checkpointing could improve portability and reproducibility. Without such standards, multi-GPU graph learning risks becoming a collection of isolated optimizations rather than a coherent engineering discipline.

9. Future Directions

Future multi-GPU graph learning systems are likely to become more adaptive. Instead of selecting a fixed parallelization strategy, a system could profile the graph and workload, then choose a hybrid strategy at runtime. It might combine graph partitioning, model parallelism, and sampling based on degree distribution, memory pressure, and interconnect bandwidth. Machine learning itself could assist scheduling, predicting which nodes will be expensive and preemptively balancing work. However, adaptive systems must remain explainable and reproducible. If the scheduler changes behavior unpredictably, debugging becomes difficult. A balanced approach is to expose adaptation policies as configurable and auditable components.

Hardware trends will also shape the field. GPUs are gaining larger memory, faster interconnects, and specialized sparse operations. This may reduce the need for aggressive partitioning, but it will not eliminate the problem. Larger graphs and models will continue to outpace memory growth. Multi-GPU systems may increasingly include heterogeneous accelerators, such as GPUs, CPUs, and specialized graph processors. Heterogeneous execution introduces new scheduling challenges. A system must decide which operations run on which device, how to move data between them, and how to maintain consistency. The most successful designs will likely treat the entire infrastructure as a single programmable fabric rather than a collection of independent devices.

Governance and policy will become more important as graph learning scales. Regulators may ask how models were trained, what data was used, and whether marginalized groups were harmed. A multi-GPU system that cannot answer these questions is not ready for high-stakes deployment. Research should therefore develop provenance tracking, fairness auditing, and energy reporting as first-class system features. These features may add overhead, but they can

be designed to run asynchronously or at checkpoint boundaries. The long-term viability of large-scale graph learning depends on public trust as much as on raw performance.

10. Conclusion

Multi-GPU parallelization for large-scale graph learning is a rich systems problem that cannot be reduced to a single optimization. Graph irregularity, skewed degree distributions, memory limits, and communication costs interact with model architecture and deployment environment. Partitioning, sampling, model parallelism, pipeline execution, tensor parallelism, and hybrid scheduling each address some bottlenecks while creating others. The most effective systems co-design kernels, partitioners, communication schedulers, and memory managers. They also account for robustness, reproducibility, fairness, energy use, and governance. As graph learning moves into high-stakes domains such as finance, healthcare, transportation, and scientific discovery, the quality of the infrastructure will determine not only speed but also accountability. The field should therefore pursue performance alongside transparency, sustainability, and equitable access. Multi-GPU graph learning is not merely a matter of adding more devices; it is a matter of designing a socio-technical system that can scale responsibly.

References

1. Scarselli, F., Gori, M., Tsoi, A. C., Hagenbuchner, M., & Monfardini, G. (2009). The graph neural network model. *IEEE Transactions on Neural Networks*, 20(1), 61–80. <https://doi.org/10.1109/TNN.2008.2005605>
2. Kipf, T. N., & Welling, M. (2017). Semi-supervised classification with graph convolutional networks. In *International Conference on Learning Representations*.
3. Veličković, P., Cucurull, G., Casanova, A., Romero, A., Liò, P., & Bengio, Y. (2018). Graph attention networks. In *International Conference on Learning Representations*.
4. Hamilton, W. L., Ying, R., & Leskovec, J. (2017). Inductive representation learning on large graphs. In *Advances in Neural Information Processing Systems* (pp. 1024–1034).
5. Ying, R., He, R., Chen, K., Eksombatchai, P., Hamilton, W. L., & Leskovec, J. (2018). Graph convolutional neural networks for web-scale recommender systems. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining* (pp. 974–983). Association for Computing Machinery. <https://doi.org/10.1145/3219819.3219890>
6. Defferrard, M., Bresson, X., & Vandergheynst, P. (2016). Convolutional neural networks on graphs with fast localized spectral filtering. In *Advances in Neural Information Processing Systems* (pp. 3844–3852).
7. Narayanan, D., Harlap, A., Phanishayee, A., Seshadri, V., Devanur, N. R., Ganger, G. R., Gibbons, P. B., & Zaharia, M. (2019). PipeDream: Generalized pipeline parallelism for DNN training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (pp. 1–15). Association for Computing Machinery. <https://doi.org/10.1145/3341301.3359646>
8. Shoenberger, M., Patwary, M., Puri, R., LeGresley, P., Casper, J., & Catanzaro, B. (2019). Megatron-LM: Training multi-billion parameter language models using model parallelism. *arXiv*. <https://arxiv.org/abs/1909.08053>

9. Rajbhandari, S., Rasley, J., Ruwase, O., & He, Y. (2020). ZeRO: Memory optimizations toward training trillion parameter models. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (pp. 1–16). IEEE. <https://doi.org/10.1109/SC41405.2020.00024>
10. Jia, Z., Zaharia, M., & Aiken, A. (2019). Beyond data and model parallelism for deep neural networks. In *Proceedings of Machine Learning and Systems* (Vol. 1, pp. 1–13).
11. Kwon, W., Li, Z., Zhuang, S., Sheng, Y., Zheng, L., Yu, C. H., Gonzalez, J. E., Zhang, H., & Stoica, I. (2023). Efficient memory management for large language model serving with PagedAttention. In *Proceedings of the 29th Symposium on Operating Systems Principles* (pp. 611–626). Association for Computing Machinery. <https://doi.org/10.1145/3600006.3613165>
12. Fey, M., & Lenssen, J. E. (2019). Fast graph representation learning with PyTorch Geometric. In *ICLR Workshop on Representation Learning on Graphs and Manifolds*.
13. Wang, M., Yu, L., Zheng, D., Gan, Q., Gai, Y., Ye, Z., Li, M., Zhou, J., Huang, Q., Ma, C., Huang, Z., Guo, Q., Zhang, H., Lin, H., Zhao, J., Li, J., Smola, A. J., & Zhang, Z. (2019). Deep Graph Library: A graph-centric, highly-performant package for graph neural networks. *arXiv*. <https://arxiv.org/abs/1909.01315>
14. Zheng, D., Ma, C., Wang, M., Zhou, J., Su, Q., Song, X., Gan, Q., Zhang, Z., & Karypis, G. (2020). DistDGL: Distributed graph neural network training for billion-scale graphs. In *2020 IEEE/ACM 10th Workshop on Irregular Applications: Architectures and Algorithms (IA3)* (pp. 36–44). IEEE. <https://doi.org/10.1109/IA351965.2020.00011>
15. Zhu, R., Zhao, K., Yang, H., Lin, W., Zhou, C., Ai, B., Li, Y., & Zhou, J. (2019). AliGraph: A comprehensive graph neural network platform. *Proceedings of the VLDB Endowment*, 12(12), 2094–2105. <https://doi.org/10.14778/3352063.3352127>
16. Karypis, G., & Kumar, V. (1998). A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing*, 20(1), 359–392. <https://doi.org/10.1137/S1064827595287997>
17. Peng, H., Xie, X., Shivdikar, K., Hasan, M. A., Zhao, J., Huang, S., ... & Ding, C. (2024, April). Maxk-gnn: Extremely fast gpu kernel design for accelerating graph neural networks training. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (pp. 683–698).
18. Huang, K., Xiao, X., Yang, H., Hu, Z., & Yang, Y. (2021). Distributed graph neural network training: A survey. *arXiv*. <https://arxiv.org/abs/2109.00661>
19. Zhou, J., Cui, G., Hu, S., Zhang, Z., Yang, C., Liu, Z., Wang, L., Li, C., & Sun, M. (2020). Graph neural networks: A review of methods and applications. *AI Open*, 1, 57–81. <https://doi.org/10.1016/j.aiopen.2021.01.001>
20. Wu, Z., Pan, S., Chen, F., Long, G., Zhang, C., & Yu, P. S. (2021). A comprehensive survey on graph neural networks. *IEEE Transactions on Neural Networks and Learning Systems*, 32(1), 4–24. <https://doi.org/10.1109/TNNLS.2020.2978386>