

Applying GraphSAGE to Large-Scale Node Classification

Programming Machine Learning Models for HPC
Final Report

Thomas Gantz Alberto Finardi Tommaso Crippa Jan Marxen

December 2025

Abstract

This report documents our study and application of GraphSAGE (Graph SAmple and aggreGatE) [**graphsage**], an inductive framework for learning node embeddings on large-scale graphs. We provide background on graph neural networks and the message passing paradigm, then describe how GraphSAGE addresses the scalability limitations of earlier approaches by learning aggregator functions rather than fixed node embeddings. We apply GraphSAGE to the ogbn-products benchmark dataset for node classification, detailing our implementation choices, hyperparameter tuning process, and experimental results.

Contents

1	Introduction	2
2	GraphSAGE Framework	2
2.1	Neighbor Sampling	2
2.2	Aggregator Functions	2
3	Implementation	2
4	Hyperparameters	3
5	Results	3
5.1	Experimental Setup	3
5.2	Mini-batch Sampling Benchmarking	3
5.3	Neighbor Sampling Benchmarking	3
5.4	GPU Scaling Benchmarking	4
6	Discussion	5
6.1	Bottlenecks Discovered	5
6.2	What Worked Well	5
6.3	Key Trade-offs	5
6.4	Lessons Learned	5
7	Conclusion	6

1 Introduction

Graph-structured data appears in many domains, including social networks, citation graphs, biological networks, and recommendation systems. A central challenge is learning node representations (embeddings) that capture both node features and structural context.

Graph Prediction Tasks. Machine learning on graphs encompasses node-level prediction (classifying individual nodes), edge-level prediction (link prediction or edge classification), and graph-level prediction (predicting properties of entire graphs). This report focuses on **node classification**, the task addressed by GraphSAGE.

Graph Neural Networks. A Graph Neural Network (GNN) is a learnable transformation that updates node features using neural networks, respects graph structure by aggregating neighbor information, and is permutation invariant. GNNs build node representations through iterative **message passing**: at each layer, nodes gather neighbor embeddings, aggregate them with a permutation-invariant function, and apply a learned transformation.

Limitations of Prior Methods. Before GraphSAGE, methods like DeepWalk and node2vec [**deepwalk**, **node2vec**] learned embeddings via random walks but were *transductive*—adding new nodes required retraining. GCNs [**gcn**] use message passing but operate on the full adjacency matrix, limiting scalability. The fundamental problem: **no method provided a parametric function to generate embeddings for unseen nodes**.

The GraphSAGE Insight. GraphSAGE’s innovation is to learn a **function** that generates embeddings rather than learning fixed embeddings per node. By learning aggregator functions that sample and combine neighborhood features, GraphSAGE becomes **inductive**: the model generalizes to new nodes or graphs without retraining.

2 GraphSAGE Framework

GraphSAGE (Graph SAmpLE and aggreGatE) implements inductive learning through neighbor sampling and learned aggregation. At each layer k , node v updates its representation by sampling neighbors, aggregating their embeddings, concatenating with its own embedding, and applying a learned transformation with non-linearity.

2.1 Neighbor Sampling

For scalability, GraphSAGE uniformly samples a fixed-size set of neighbors at each layer rather than using all neighbors. For a K -layer model with sample sizes S_1, \dots, S_K , per-node complexity is $O(\prod_{k=1}^K S_k)$, independent of actual node degrees.

2.2 Aggregator Functions

GraphSAGE supports several aggregators: **mean** (element-wise average), **LSTM** (sequential processing of a random neighbor permutation), and **pooling** (neural network followed by max-pooling). The mean aggregator is simplest and often effective; pooling provides more expressiveness.

3 Implementation

Implementation details will be provided here, including our PyTorch/DGL code structure, data loading pipeline, and distributed training setup.

4 Hyperparameters

Hyperparameter tuning methodology and final configurations will be documented here, including learning rate schedules, hidden dimensions, number of layers, sample sizes, dropout rates, and batch sizes.

5 Results

5.1 Experimental Setup

All benchmarking experiments were conducted on the MeluXina supercomputer (Luxembourg National Supercomputer) using the Accelerator Module with 4x NVIDIA A100 GPUs (40GB each) per node and HDR200 InfiniBand interconnect (400 Gb/s aggregate bandwidth). We evaluate three key dimensions: mini-batch sampling, neighbor sampling strategies, and GPU scaling.

5.2 Mini-batch Sampling Benchmarking

Mini-batch sampling controls the trade-off between training speed and GPU memory utilization. Figure 1 presents the impact of batch size on training time and memory consumption.

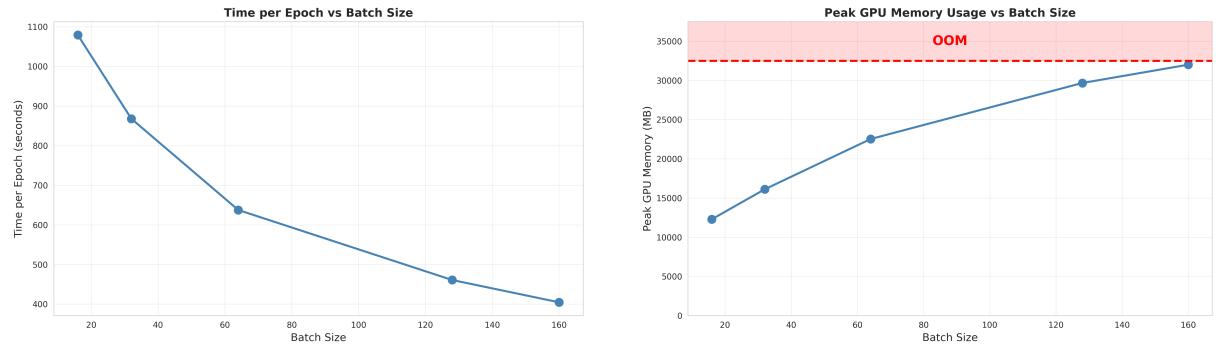
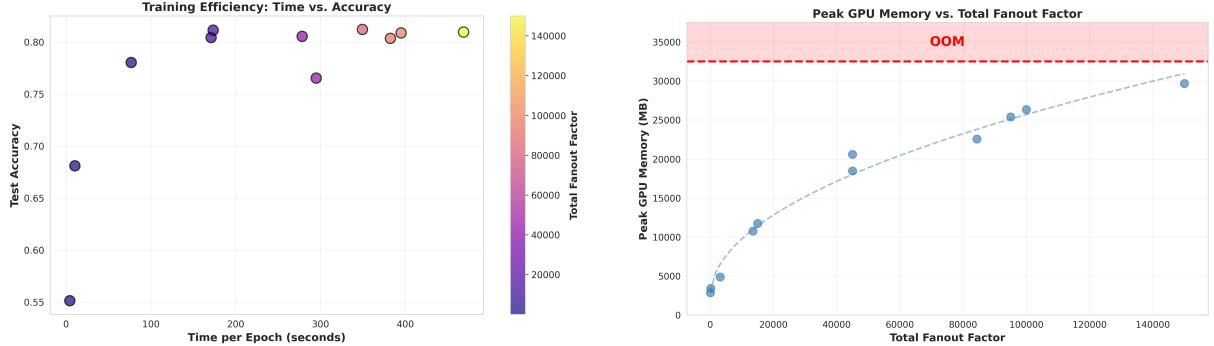


Figure 1: Mini-batch Sampling Analysis

Key Finding: Batch sizes 128–160 provide optimal balance between memory utilization and speed of training.

5.3 Neighbor Sampling Benchmarking

Neighbor sampling strategy critically impacts both accuracy and computational cost. Figure 2 illustrates the trade-offs between accuracy, training time, and memory consumption across various fanout configurations.



(a) Plot showing accuracy-time trade-off. Upper-left region indicates efficient configurations.

(b) Memory consumption scales rapidly with total fanout.

Figure 2: Neighbor Sampling Trade-offs

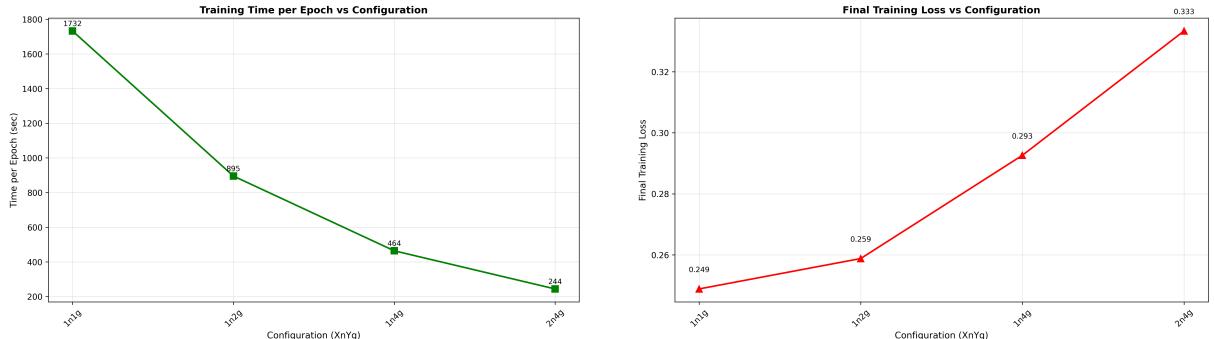
Top Configurations (by Test Accuracy):

Rank	Fanout	Test Acc	Time (h)
1	[15, 15, 15, 5, 5]	0.8121	14.56
2	[15, 10, 10, 10, 0]	0.8114	7.22
3	[15, 10, 10, 10, 10]	0.8096	19.53
4	[10, 10, 10, 10, 10]	0.8088	16.46

Key Finding: Configuration [15,10,10,10,0] achieves competitive accuracy (81.14%) in only 7.22 hours, 2.7x faster than the baseline. This suggests the final layer benefits from reduced neighbor sampling, as deeper aggregations capture sufficient context.

5.4 GPU Scaling Benchmarking

Distributed training via PyTorch DDP enables efficient multi-GPU utilization. We evaluate scaling from 1 to 8 GPUs (4 GPUs per node, 2 nodes).



(a) Strong scaling up to 4 GPUs (93% efficiency). 8 GPUs across 2 nodes: 89% efficiency.

(b) Training loss increases with GPU count due to larger effective batch sizes.

Figure 3: Distributed Training Analysis

Key Finding: Strong intra-node scaling achieves 3.73x speedup (4 GPUs, 93% efficiency). Inter-node scaling to 8 GPUs yields ~7x speedup (89% efficiency), showing acceptable degradation. The trade-off between training loss and wall-clock time is favorable: modest loss increase for 7x speedup justifies distributed training.

6 Discussion

6.1 Bottlenecks Discovered

Memory Wall: The primary bottleneck is not computation but memory. Neighbor sampling causes exponential subgraph expansion: fanout [15,10,10,10,10] expands each seed node to 150k neighbors. With batch size 128, this yields 19.2M nodes per batch, requiring most of the GPU memory for subgraph storage and intermediate activations. Model parameters account for only \sim 8MB.

Inter-node Communication Overhead: Gradient synchronization via InfiniBand incurs costs. Intra-node scaling (1 \rightarrow 4 GPUs) maintains 93% efficiency; inter-node scaling (4 \rightarrow 8 GPUs) drops to 89%, representing a 4% efficiency loss. This motivates careful consideration of multi-node training.

Loss Convergence Trade-off: Larger effective batch sizes (product of batch size and GPU count) lead to higher training loss. Effective batch size becomes 512 (4 GPUs), 1024 (8 GPUs), etc., contributing to wider minima and reduced gradient noise.

6.2 What Worked Well

DDP Implementation: PyTorch’s DistributedDataParallel seamlessly handles gradient synchronization with minimal code changes from single-GPU baseline. Automatic all-reduce operations ensure consistent parameter updates across ranks.

Strong Intra-node Scaling: The system scales nearly linearly up to 4 GPUs on a single node, with PCIe/NVLink providing sufficient bandwidth. This validates the approach for datasets fitting within node memory.

Containerization: Apptainer/Singularity ensured reproducibility across HPC environments, isolating complex PyTorch Geometric dependencies and avoiding version conflicts.

6.3 Key Trade-offs

- **Batch Size vs. Memory:** Batch 128–160 balances throughput and memory efficiency; further growth hits hard limits.
- **Receptive Field vs. Fanout:** Higher fanouts capture more context but cause exponential memory growth.
- **Speedup vs. Convergence:** 8 GPU training achieves 7x wall-clock speedup at the cost of slightly higher training loss. Test accuracy remains competitive, justifying the trade-off.

6.4 Lessons Learned

1. **Accuracy vs. Performance:** [15,10,10,10,0] shows better accuracy with 2.7x speedup compared to baseline [15,10,10,10,10], showing that the 5th layer can be removed without significant accuracy loss.
2. **Profile before optimizing:** Memory emerged as the true bottleneck, not computation. Early profiling revealed this, guiding optimization focus.
3. **No free lunch:** Every optimization involves trade-offs. Larger batches speed training but hurt convergence. Distributed training provides speedup at communication cost.
4. **Reproducibility is paramount:** Containerization, fixed seeds, and detailed logging are essential for scientific HPC work. Variability in system state can obscure true performance characteristics.

7 Conclusion

This study demonstrates that GraphSAGE, when properly tuned and deployed on HPC infrastructure, scales effectively to large-scale node classification tasks. Key findings include: (1) batch sizes 128–160 offer optimal memory-speed trade-offs; (2) neighbor sampling strategy [15,10,10,10,0] achieves baseline accuracy with 2.7x speedup; (3) strong intra-node scaling validates multi-GPU deployment on a single node; (4) memory is the critical bottleneck, not computation. Future work could explore gradient compression, asynchronous training, and adaptive neighbor sampling to further improve efficiency.