Applying GraphSAGE to Large-Scale Node Classification

Programming Machine Learning Algorithms for HPC - Final Report

Thomas Gantz
University of Luxembourg
Luxembourg
thomas.gantz.001@student.uni.lu

Alberto Finardi
University of Luxembourg
Luxembourg
alberto.finardi.001@student.uni.lu

Tommaso Crippa
University of Luxembourg
Luxembourg
tommaso.crippa.001@student.uni.lu

Jan Esquivel Marxen
University of Luxembourg
Luxembourg
jan.esquivel.001@student.uni.lu

ABSTRACT

This report documents our study and application of GraphSAGE (Graph SAmple and aggreGatE) [Hamilton et al. 2017], 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.

Code and experiments available at: https://github.com/albertofinardi/Sage_GNN

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**, One of the possible applications of 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 [Grover and Leskovec 2016; Perozzi et al. 2014] learned embeddings via random walks but were *transductive* - adding new nodes required retraining. GCNs [Kipf and Welling 2016] 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 $S_1, \ldots, S_K$, 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 of neighbor embeddings), **LSTM** (applies an LSTM network to aggregate neighbor embeddings sequentially), and **pooling** (applies a neural network to each neighbor embedding followed by element-wise max-pooling). The mean aggregator is simplest and computationally efficient; LSTM and pooling provide greater expressiveness at higher computational cost.

3 IMPLEMENTATION

3.1 Dataset and Challenge

We evaluate GraphSAGE on the **ogbn-products** dataset [Hu et al. 2020], an Amazon co-purchase network with 2.4M nodes (products), 61M edges, and 47 classes (8% train, 2% validation, 90% test split). The scale makes full-batch training infeasible, as the entire graph cannot fit in GPU memory and full message passing on 2.4M nodes is computationally prohibitive. This necessitates **mini-batch training with neighbor sampling** as the core solution strategy.

3.2 Sampling Strategy

We employ two complementary strategies:

Mini-Batch Sampling. We select a small subset of seed nodes per iteration (e.g., 128 nodes). Only embeddings for these nodes and their neighborhoods are computed, and model parameters are updated based on the batch loss. This enables processing large graphs that cannot fit entirely in GPU memory.

Neighbor Sampling (k-hop). For each seed node, we sample neighbors recursively for k layers. The **fanout** $[S_1, S_2, \ldots, S_k]$ specifies the number of neighbors sampled per layer. For example, fanout $[15, 10, 10, 10, 10]$ samples 15 neighbors at layer 1 and 10 neighbors at layers 2–5.

The Memory Challenge. This sampling strategy causes **exponential growth**: with fanout $[15, 10, 10, 10, 10]$, each seed node expands to approximately $15 \times 10^4 = 150{,}000$ neighbors. With 128 seed nodes, this yields $128 \times 150{,}000 = 19.2$M nodes per batch. The memory bottleneck is clear: sampled subgraphs and intermediate activations consume 30–40 GB, while model parameters require only ~8 MB. Thus, **subgraph storage dominates over model parameters**.

Figures 1 and 2 illustrate this exponential growth pattern.

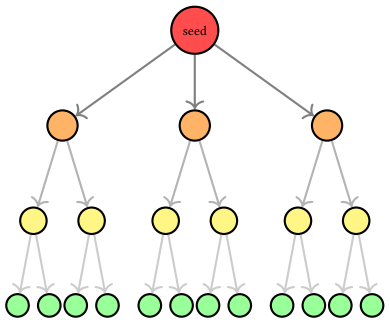

Figure 1: K-hop neighborhood expansion visualization with simplified fanout [3, 2, 2]. Starting from a single seed node, each layer multiplies the node count by the fanout value, demonstrating the exponential growth pattern inherent in neighbor sampling.

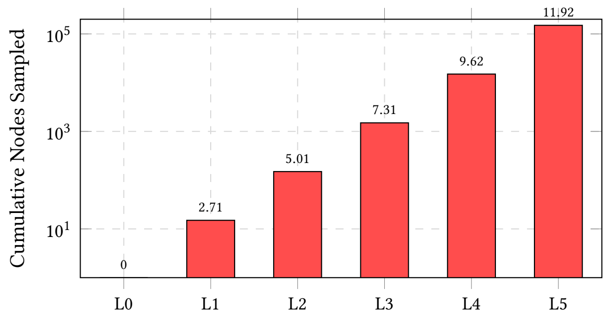

Figure 2: Memory explosion with real fanout configuration [15, 10, 10, 10, 10]. Each layer multiplies the sampled nodes, reaching 150,000 neighbors per seed node at layer 5. Combined across 128 seed nodes per batch, this yields 19.2M nodes consuming 30–40 GB of GPU memory. Note that the y-axis is logarithmic.

3.3 Infrastructure and Software

We use **PyTorch 2.1.2** (CUDA 12.1, cuDNN 8) with **PyTorch Geometric** for neighbor sampling (`NeighborLoader`) and GraphSAGE layers (`SAGEConv`). The pipeline is containerized via **Apptainer/Singularity** for reproducibility, bundling PyG, OGB, and compiled CUDA extensions.

3.4 Distributed Training

To overcome single-GPU memory limits, we use **PyTorch DDP** (Distributed Data Parallel) for multi-GPU training.

Why DDP and Not DeepSpeed? The choice of DDP over more sophisticated frameworks like DeepSpeed is deliberate and based on our specific bottleneck analysis:

- **Model size is tiny:** Our GraphSAGE model has only ~400K parameters (~8 MB), fitting comfortably in GPU memory. No model parallelism, tensor parallelism, or pipeline parallelism is needed.
- **Memory bottleneck is data, not parameters:** The memory pressure comes from sampled subgraphs (30–40 GB per batch), not model weights. DeepSpeed's Zero Redundancy Optimizer (ZeRO) optimizes parameter storage - irrelevant when parameters are negligible.
- **Data parallelism is sufficient:** The problem requires distributing *data* (batches) across GPUs, not *model* partitioning. DDP's simple data-parallel strategy is exactly what we need.
- **Simplicity and maintainability:** DDP requires minimal code changes and integrates seamlessly with PyTorch. DeepSpeed adds complexity (configuration files, custom optimizers) without tangible benefits for our use case.

In summary: **DeepSpeed solves the wrong problem**. Our bottleneck is subgraph storage, not model size. DDP's straightforward data parallelism is the optimal solution.

DDP Training Process. The distributed training workflow consists of four key steps:

(1) **Process Initialization:** SLURM launches independent ranks per GPU (one process per GPU), each with its own Python interpreter and CUDA context. Processes communicate via the **NCCL** backend for GPU-to-GPU communication.
(2) **Data Partitioning:** The training set is split across ranks with no overlap. Each rank maintains an independent `NeighborLoader` for k-hop sampling, with 4–8 worker processes prefetching batches per GPU.
(3) **Forward + Backward (Parallel):** Each GPU independently processes its batch, computes loss, and backpropagates gradients.
(4) **Gradient Synchronization:** DDP performs an all-reduce operation to average gradients across all GPUs, ensuring synchronized parameter updates via the optimizer step.

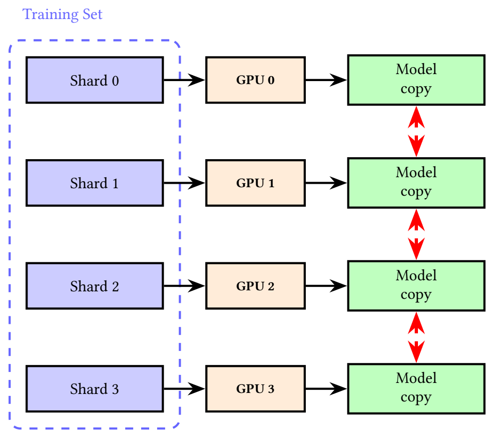

Figure 3: DDP architecture: data partitioned across GPUs, full model replicated per rank. Gradients synchronized via NCCL all-reduce after each batch. This data-parallel strategy is optimal when model size is small but data volume is large.

Figure 4 illustrates the complete system architecture, showing how SLURM orchestrates containerized training across multiple GPUs with partitioned data and NCCL-based gradient synchronization.

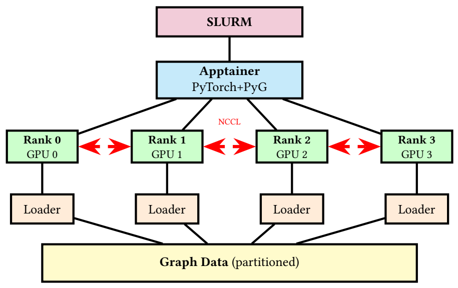

Figure 4: System architecture for distributed GraphSAGE training. SLURM launches 4 containerized processes (one per GPU), each running an independent rank with its own NeighborLoader sampling from partitioned graph data. Gradients are synchronized across ranks via NCCL all-reduce.

3.5 Model Architecture

Our GraphSAGE model consists of 5 layers, each using the **SAGEConv** layer with mean aggregation, followed by **LayerNorm**, **ReLU** activation, and **Dropout** (rate 0.5). The final layer is followed by a linear classifier for the 47-class prediction task. The architecture follows: 100-dimensional input (ogbn-products node features) $\rightarrow$ 256-dimensional hidden representations $\rightarrow$ 47-dimensional output (class logits). The model contains approximately 400K parameters, totaling ~8 MB in memory.

4 HYPERPARAMETERS

4.1 Hyperparameter Tuning Methodology

To systematically explore the hyperparameter space and identify optimal configurations for large-scale GraphSAGE training, we developed a scalable and reproducible benchmarking pipeline. The goal is to enable **systematic, parallel hyperparameter optimization** on HPC infrastructure while maintaining complete reproducibility.

Benchmark Pipeline Architecture. Our pipeline follows a CSV-driven workflow:

(1) **Configuration Encoding:** Hyperparameter combinations are encoded in a `hyperparams.csv` file, with each row representing one training configuration.
(2) **SLURM Job Arrays:** Job arrays execute multiple training runs in parallel across the cluster. Each array task reads a specific configuration row by index (`$SLURM_ARRAY_TASK_ID`).
(3) **Containerized Training:** All experiments run within Apptainer containers (based on `pytorch/pytorch:2.1.2-cuda12.1-cudnn8-runtime`) to ensure identical environments across cluster nodes and eliminate dependency variability.
(4) **DDP Training Execution:** Each job launches distributed training across 4 GPUs using PyTorch DDP, with training progress and final metrics logged.
(5) **Result Aggregation:** Training jobs write results (accuracy, loss, time, memory) to a shared `results.csv` file with file locking to prevent concurrent write conflicts.
(6) **Automated Analysis:** Python plotting scripts (`plotter.py`) aggregate results, compute statistics, and generate comparative visualizations.

This architecture enables efficient parallel exploration of multiple hyperparameter configurations while maintaining scientific rigor through containerization and automated logging.

4.2 Hyperparameter Space

We explored three primary dimensions affecting training performance and model quality: **mini-batch sampling** (batch size and gradient accumulation), **neighbor sampling** (fanout strategies), and **GPU scaling** (distributed training). Table 1 summarizes the tested parameters and their ranges, as well as fixed architectural and training hyperparameters.

Effective Batch Size. The effective batch size is computed as:

$$\text{Effective Batch Size} = \text{batch\_size} \times \text{accum\_steps} \times \text{world\_size}$$

Table 1: Hyperparameter configurations tested and fixed values

Tested Parameters	
Batch size	16, 32, 64, 128, 160, 192, 256, 512
Gradient accumulation	1, 2, 3, 4, 5
Neighbor fanout	16 configurations
GPU count	1, 2, 4, 8
Fixed Parameters	
Layers	5
Hidden dimensions	256
Learning rate	0.003
Epochs	150
Early stopping patience	10
Dropout rate	0.5

where `world_size` is the number of GPUs. For example, with batch size 128, accumulation steps 5, and 4 GPUs, the effective batch size is $128 \times 5 \times 4 = 2560$.

4.3 Configurations Tested

In total, we evaluated **36 hyperparameter configurations** organized into three benchmark categories:

Mini-Batch Sampling (16 configurations). We tested combinations of batch size (16, 32, 64, 128, 160, 192, 256, 512) and gradient accumulation steps (1–5). The goal was to understand the trade-off between training time per epoch, peak GPU memory usage, and overall training efficiency. Batch sizes beyond 160 exceeded the 40 GB A100 memory limit and resulted in out-of-memory (OOM) errors.

Neighbor Sampling (16 configurations). We explored diverse fanout strategies to balance accuracy, training time, and memory consumption. We tested the original baseline configuration ([15, 10, 10, 10, 10]), uniform samplers (e.g., [10,10,10,10,10]), increasing tapers (e.g., [3,5,10,15,20]), and layer ablation studies (e.g., [15,10,10,0,0]).

GPU Scaling (4 configurations). We evaluated distributed training scalability across 1, 2, 4 (single node), and 8 GPUs (2 nodes), with fixed batch size 128 and fanout [15,10,10,10,10]. The focus was on measuring speedup, parallel efficiency, and the impact of effective batch size on convergence.

5 RESULTS

5.1 Experimental Setup

All benchmarking experiments were conducted on the MeluXina supercomputer using the Accelerator Module with 4x NVIDIA A100 GPUs (40GB each) per node. 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. We varied batch size from 16 to 512 and gradient accumulation steps from 1 to 5, measuring training time per epoch and peak memory consumption. Figures 5 and 6 present the impact of batch size on these metrics.

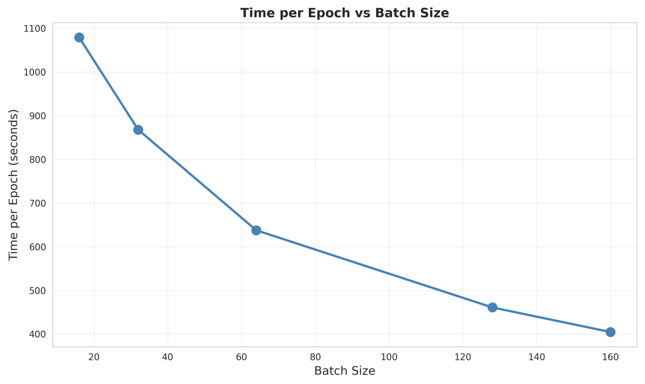

Figure 5: Training time per epoch decreases with larger batch sizes due to better GPU utilization and reduced forward/backward pass overhead.

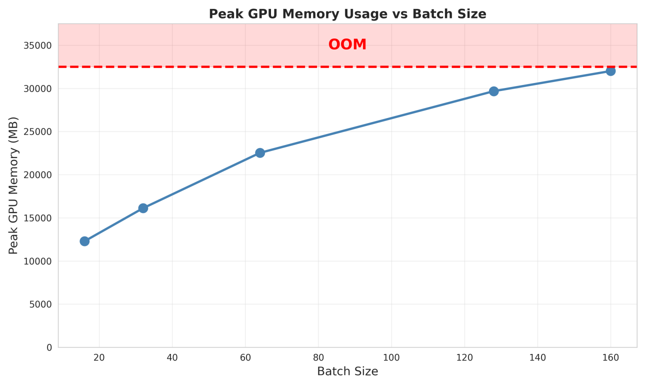

Figure 6: Peak memory consumption scales linearly with batch size. Maximum reached batch size is 160, using ∼32 GB (with ∼7.5 GB system overhead).

Key Findings:

- **Inverse relationship:** Larger batch sizes yield faster training per epoch due to better GPU utilization and amortized computational overhead.
- **Memory ceiling:** Batch sizes beyond 160 exceed the 40 GB A100 memory limit, resulting in OOM errors. The 160 batch size uses approximately 32 GB, leaving ∼7.5 GB for system overhead.
- **Optimal range:** Batch sizes 128–160 provide the best balance between memory efficiency and training speed.

5.3 Neighbor Sampling Benchmarking

Neighbor sampling strategy critically impacts both accuracy and computational cost. We tested 16 fanout configurations, including uniform strategies and layer ablation studies. The fanout determines the receptive field size and thus the amount of structural context available to the model, but also directly affects memory consumption through exponential neighborhood expansion. Figures 7 and 8 illustrate the trade-offs between accuracy, training time, and memory consumption across various fanout configurations.

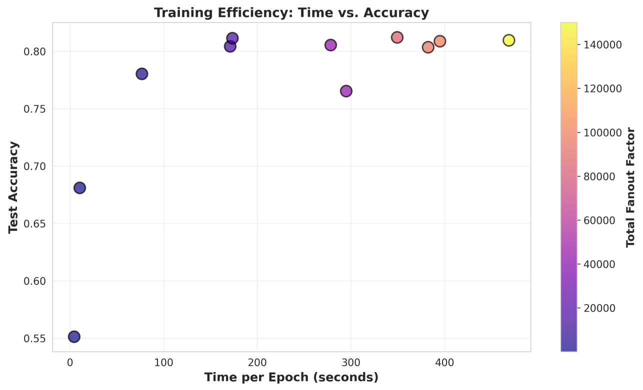

Figure 7: Accuracy-time trade-off. Upper-left region (high accuracy, low time) indicates efficient configurations. Moderate fanouts achieve the best balance.

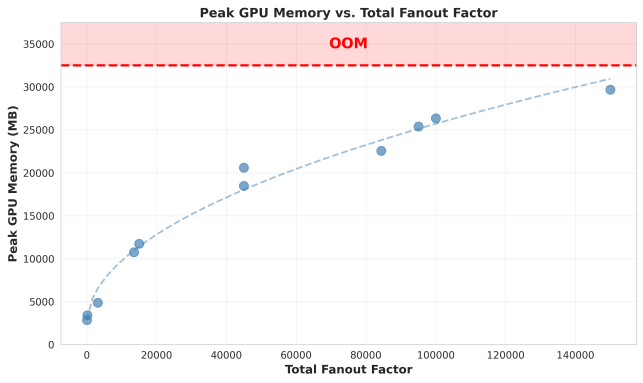

Figure 8: Memory consumption scales rapidly with total fanout ($F_{\text{total}} = \prod_{i=1}^{5} S_i$). High fanouts hit the 40 GB memory limit.

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
5	[20, 15, 10, 5, 3]	0.8055	11.60

Key Findings:

- **Optimal efficiency:** Configuration [15,10,10,10,0] achieves near-baseline accuracy (81.14% vs. 80.96%) in only 7.22 hours, representing a **2.7× speedup** compared to the baseline [15,10,10,10,10]. This suggests that the model can be reduced to a 4-layer aggregation without significant accuracy loss.
- **Moderate fanouts dominate:** The upper-left region of the accuracy-time plot is populated by moderate fanout configurations. Very high fanouts provide no proportional accuracy gains and incur excessive memory and time costs.
- **Aggressive sampling hurts:** Extremely low fanouts (e.g., [5,5,5,5,5]) or single-layer strategies (e.g., [50,0,0,0,0]) either underperform or exhibit training instability.

5.4 GPU Scaling Benchmarking

Distributed training via PyTorch DDP enables efficient multi-GPU utilization. We evaluated scaling across 1, 2, 4 (single node), and 8 GPUs (2 nodes) with fixed hyperparameters (batch size 128, fanout [15,10,10,10,10]) to isolate the effect of parallelization. Training was limited to 10 epochs for faster benchmarking. Figures 9 and 10 show training time per epoch and training loss convergence across GPU counts.

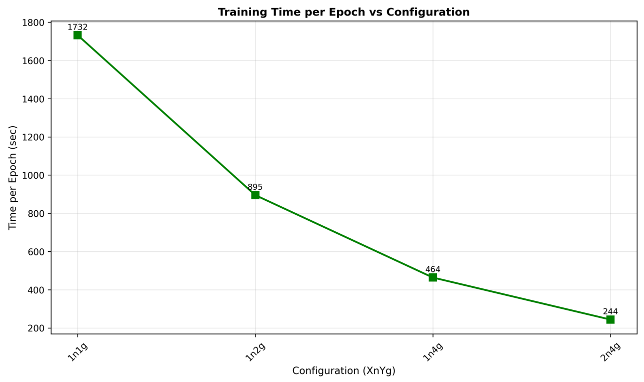

Figure 9: Strong scaling demonstrates near-linear speedup up to 4 GPUs (93% efficiency). Inter-node scaling to 8 GPUs achieves 89% efficiency with modest degradation due to inter-node communication overhead.

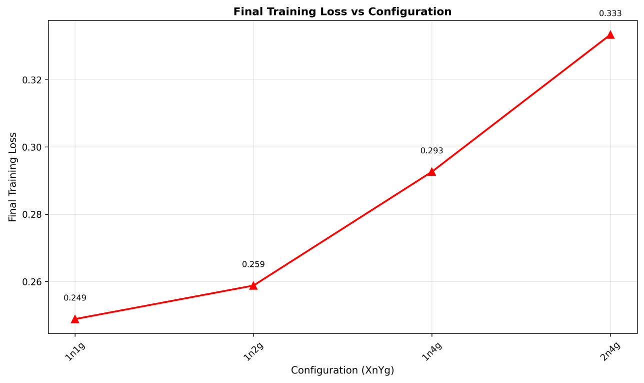

Figure 10: Training loss increases with GPU count due to larger effective batch sizes (512 for 4 GPUs, 1024 for 8 GPUs), resulting in reduced gradient noise.

Key Findings:

- **Strong intra-node scaling:** 1⟶2 GPUs achieves 1.93× speedup (96.5% efficiency); 1⟶4 GPUs achieves 3.73× speedup (93% efficiency). This demonstrates that PCIe/NVLink bandwidth is sufficient for gradient synchronization within a single node.
- **Acceptable inter-node scaling:** 1⟶8 GPUs achieves 7.1× speedup (89% efficiency). The 4% efficiency drop relative to intra-node scaling is attributable to NCCL all-reduce communication over multi nodes, which introduces latency for gradient synchronization across nodes.
- **Loss-speed trade-off:** Larger effective batch sizes (batch_size × world_size = 512 for 4 GPUs, 1024 for 8 GPUs) lead to higher training loss due to reduced gradient noise. However, test accuracy remains competitive, justifying the 7× wall-clock speedup for 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 approximately 150k neighbors. With batch size 128, this yields 19.2M nodes per batch, consuming 30–40 GB for subgraph storage and intermediate activations. In contrast, model parameters account for only ~8 MB. This reveals that **sampled subgraphs dominate memory usage by 3–4 orders of magnitude over model parameters**, making memory - not compute - the critical resource constraint.

Inter-node Communication Overhead. Gradient synchronization via NCCL over multiple nodes introduces communication overhead. Intra-node scaling (1⟶4 GPUs) achieves 93% efficiency via high-bandwidth NVLink/PCIe, while inter-node scaling (4⟶8 GPUs) drops to 89% efficiency. This 4% degradation reflects the latency cost of all-reduce operations across network fabric, motivating careful consideration of when multi-node training is warranted.

Loss Convergence Trade-off. Larger effective batch sizes (product of batch_size, accum_steps, and world_size) lead to higher training loss. Effective batch sizes of 512 (4 GPUs) and 1024 (8 GPUs) reduce gradient noise, resulting in convergence to minima with higher loss. However, this increase in effective batch size can be mitigated by reducing the learning rate.

6.2 What Worked Well

DDP Implementation. PyTorch's DistributedDataParallel seamlessly handles gradient synchronization with minimal code changes from the single-GPU baseline. The automatic all-reduce operations ensure consistent parameter updates across all ranks, abstracting away the complexity of distributed communication. This enabled us to scale from single-GPU to 8-GPU training with only minor modifications to the training loop.

Strong Intra-node Scaling. The system scales nearly linearly up to 4 GPUs on a single node (3.73× speedup, 93% efficiency), demonstrating that PCIe/NVLink provides sufficient bandwidth for gradient synchronization. This validates DDP as an effective approach for datasets that fit within the memory constraints of a single node but benefit from data parallelism.

Containerization. Apptainer/Singularity ensured complete reproducibility across HPC environments by encapsulating the entire software stack (PyTorch 2.1.2, PyTorch Geometric, CUDA 12.1, cuDNN 8, and compiled CUDA extensions). This isolated complex dependencies, avoided version conflicts on shared clusters, and enabled seamless portability across different compute nodes.

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.

6.4 Lessons Learned

Profile First, Optimize Second. Early profiling revealed that memory - not computation - is the true bottleneck in large-scale GraphSAGE training. This insight guided our optimization efforts toward batch size tuning and fanout reduction rather than computational optimizations (e.g., kernel fusion, mixed-precision training). Without profiling, we might have pursued ineffective optimizations.

Trade-offs Are Inevitable. Every optimization involves trade-offs, and there is no "free lunch." Larger batch sizes accelerate training but increase memory consumption and can hurt convergence. Higher fanouts improve accuracy but cause exponential memory growth. Distributed training provides speedup but introduces communication overhead and loss degradation. Effective system design requires balancing these competing objectives based on task priorities.

Reproducibility matters. Containerization (Apptainer), fixed random seeds, and detailed logging are essential for rigorous scientific HPC work. Variability in system state (library versions, CUDA drivers, environment variables) can obscure true performance characteristics and make results non-reproducible. Our CSV-driven pipeline and containerized experiments ensure that all results can be independently verified.

Layer Ablation Insights. The discovery that configuration [15,10,10,10,0] achieves better accuracy than the baseline (81.14% vs. 80.96%) with a 2.7× speedup demonstrates that the 5th layer contributes minimal additional context. This suggests that 4-hop neighborhood aggregation captures sufficient structural information for this dataset, and that **deeper is not always better** in GNN design.

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.

REFERENCES

Aditya Grover and Jure Leskovec. 2016. node2vec: Scalable Feature Learning for Networks. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 855–864.

William L. Hamilton, Rex Ying, and Jure Leskovec. 2017. Inductive Representation Learning on Large Graphs. In *Advances in Neural Information Processing Systems (NeurIPS)*, Vol. 30. 1024–1034. https://arxiv.org/abs/1706.02216

Weihua Hu, Matthias Fey, Marinka Zitnik, Yuxiao Dong, Hongyu Ren, Bowen Liu, Michele Catasta, and Jure Leskovec. 2020. Open Graph Benchmark: Datasets for Machine Learning on Graphs. In *Advances in Neural Information Processing Systems (NeurIPS)*, Vol. 33. 22118–22133. https://arxiv.org/abs/2005.00687

Thomas N. Kipf and Max Welling. 2016. Semi-Supervised Classification with Graph Convolutional Networks. *arXiv preprint arXiv:1609.02907* (2016).

Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. 2014. DeepWalk: Online Learning of Social Representations. In *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 701–710.