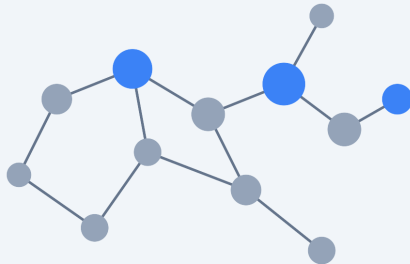


# GRAPHSAGE

## INDUCTIVE REPRESENTATION LEARNING ON LARGE GRAPHS

Thomas Gantz  
Alberto Finardi  
Tommaso Crippa  
Jan Marxen

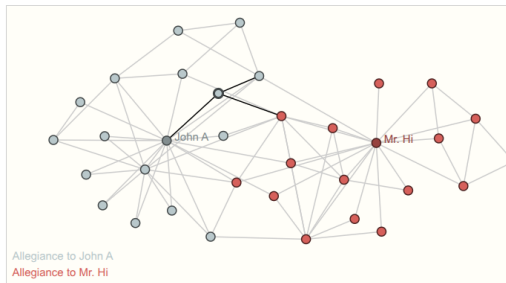
December 10, 2025



# Introduction

**Primary focus:** Node-level prediction — predict properties of individual nodes (classification, regression).

- **Node-level (focus):** Predict node attributes or labels using features and neighbor information.
- **Edge-level:** Predict relationships between node pairs (link prediction, edge classification).
- **Graph-level:** Predict properties of whole graphs (e.g., molecule properties).

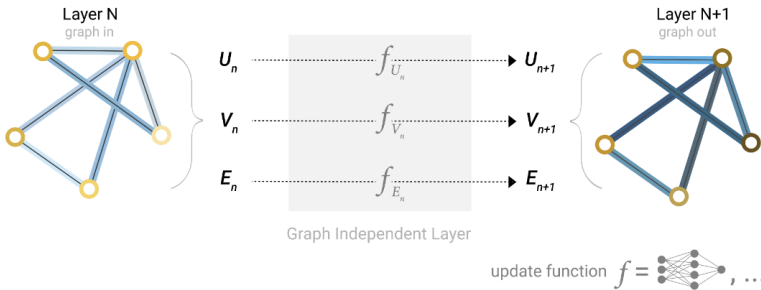


# What is a Graph Neural Network?

## Introduction

A **learnable transformation** on graph attributes that:

- Updates node/edge/graph features using **neural networks**
- **Respects graph structure** by aggregating information from neighbors
- Is **permutation invariant** (order of nodes doesn't matter)

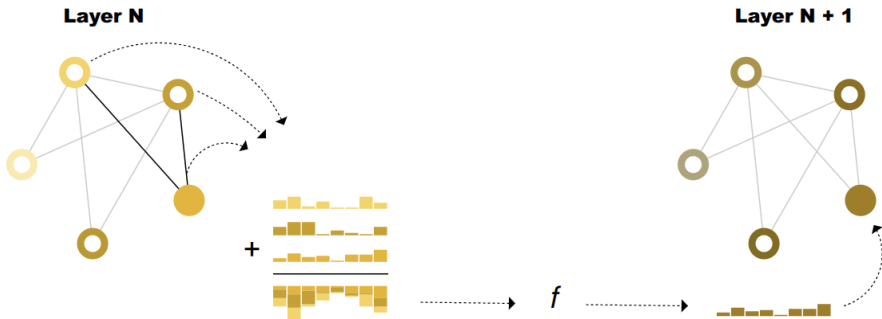


# Message Passing: The Core Idea

## Introduction

Three steps repeated at each layer:

1. **Gather:** Collect embeddings from neighboring nodes
2. **Aggregate:** Combine neighbors' info (sum / mean / max)
3. **Update:** Apply learned transform using the aggregated vector



# Message Passing: Notation

## Introduction

### Message Passing Equation

$$h_v^{(k)} = \sigma \left( W \cdot \left[ h_v^{(k-1)}, \text{AGG}(\{h_u : u \in N(v)\}) \right] \right)$$

- $h_v^{(k)}$  — Node  $v$  repr. at layer  $k$
- $h_u$  — Neighbor node  $u$  repr.
- $N(v)$  — Neighbors of  $v$
- $k$  — Layer index (hops)
- $\text{AGG}(\cdot)$  — Aggregator (mean, sum, max)
- $W$  — Learnable weight matrix
- $\sigma$  — Activation (ReLU, tanh)

# Before GraphSAGE: The Problem

## Introduction

*Let's audit the limitations of existing approaches...*

- **DeepWalk / node2vec:** Transductive: embed every node; new nodes need full retraining.
  - ▷ Not GNNs — random-walk based embeddings, no message passing
- **GCNs:** Often require access to the full graph during training/inference; can be costly to scale.
  - ▷ GNNs — but transductive: fixed node set at training
- **Result:** No compact parametric function to generate embeddings for unseen nodes.

*This motivates GraphSAGE's key insight...*

# The Key Insight



# The Key Insight

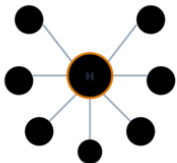
The Key Insight

Don't learn embeddings for each node...

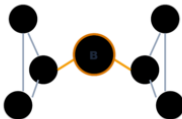
Learn a **FUNCTION** that generates embeddings

*By sampling & aggregating neighborhood features*

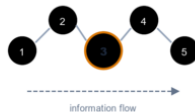
**Hub Pattern**  
Social Influencer



**Bridge Pattern**  
Community Connector



**Chain Pattern**  
Sequential Flow



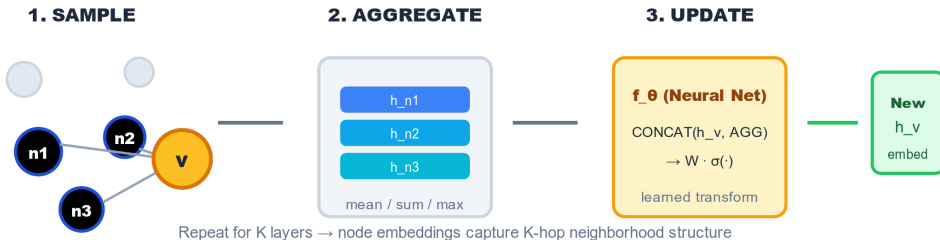
# GraphSAGE Framework

# GraphSAGE: Inductive Framework

## GraphSAGE Framework

### Core Principle: Sample + Aggregate

- Learn **aggregator functions** (not node embeddings)
- For any node  $v$ : **sample neighbors, aggregate their features**
- Pass through learned neural networks
- **Inference:** Apply same function to unseen nodes



# Implementation

# Training GraphSAGE: The Challenge

## Implementation

**Dataset:** ogbn-products (Amazon co-purchase network)

- 2.4M nodes (products), 61M edges, 47 classes
- 8% train / 2% val / 90% test split

**The Problem:**

- Cannot fit entire graph in GPU memory
- Cannot do full message passing on 2.4M nodes
- **Solution:** Mini-batch training + neighbor sampling

# Mini-Batch and Neighbor Sampling

## Implementation

### 1. Mini-Batch Sampling:

- Select small subset of seed nodes per iteration (e.g., 128 nodes)
- Only compute embeddings for these nodes + their neighborhoods
- Update model parameters based on batch loss

### 2. Neighbor Sampling (k-hop):

- For each seed node, sample neighbors recursively for  $k$  layers
- **Fanout**: number of neighbors sampled per layer  $[L_1, L_2, \dots, L_k]$
- Example:  $[15, 10, 10, 10, 10] = 15$  neighbors at layer 1, 10 at layers 2-5

### Why this matters:

- Fanout grows **exponentially**:  $\sim 150k$  neighbors per seed node
- $128$  seed nodes  $\times 150k$  expansion = **19.2M nodes per batch**
- Memory bottleneck: sampled subgraphs  $\gg$  model parameters

# Neighbor Sampling: Visual Example

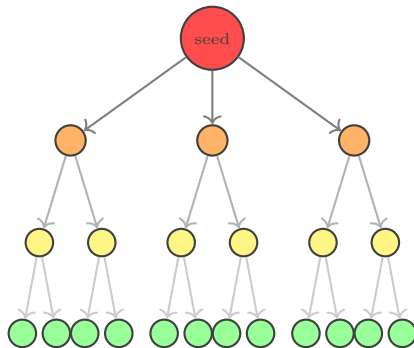
## Implementation

### K-Hop Expansion:

- Start with 1 seed node
- Sample neighbors recursively
- Fanout:  $[3, 2, 2]$  (simplified)

### Growth per layer:

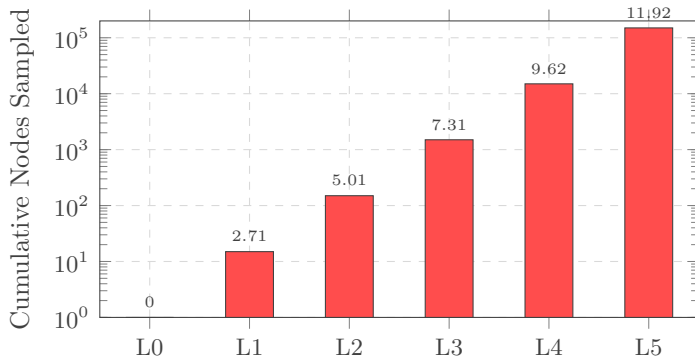
- Layer 0: 1 seed
- Layer 1:  $1 \times 3 = 3$
- Layer 2:  $3 \times 2 = 6$
- Layer 3:  $6 \times 2 = 12$



# The Memory Explosion

## Implementation

Real Configuration: [15, 10, 10, 10, 10]



Combined: 128 seed nodes  $\times$  150k neighbors = **19.2M nodes/batch**



## HPC Infrastructure:

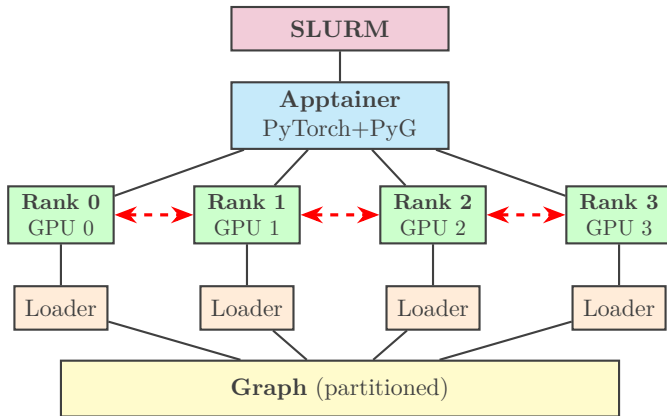
- **Cluster:** MeluXina (Luxembourg National Supercomputer)
- **GPUs:** Up to 4x NVIDIA A100 (40GB) per node
- **Framework:** PyTorch 2.1.2 + PyTorch Geometric
- **Containerization:** Apptainer/Singularity for reproducibility

## Why Distributed Training?

- Single GPU: limited batch size due to memory (sampled subgraphs  $\sim 30\text{-}40\text{GB}$ )
- **Solution:** Distributed Data Parallel (DDP) across multiple GPUs
- Partition training data, synchronize gradients

# System Architecture Diagram

## Implementation



Multi-GPU DDP: data partitioned across 4 GPUs, gradients synced via NCCL

# Containerization

## Implementation

- **Base:** `pytorch/pytorch:2.1.2-cuda12.1-cudnn8-runtime`
- **Dependencies:** PyTorch Geometric, pyg-lib, OGB, torch-scatter/sparse
- **Why containerize?**
  - Complex dependency graph (CUDA-compiled extensions)
  - Reproducibility across HPC environments
  - Avoid version conflicts on shared cluster

# Why HPC and Parallelism?

## Implementation

### 1. Dimension of the Problem:

- Each batch node samples 5-hop neighborhoods
- Fanout  $\rightarrow$  exponential growth:  $\sim 150\text{k}$  neighbors per seed node
- Batch  $128 \text{ nodes} \times 150\text{k expansion} = 19.2\text{M nodes per batch}$

### 2. Memory Bottleneck:

- Model parameters: only  $\sim 8 \text{ MB}$
- Sampled subgraphs + intermediate activations: **30-40 GB** per batch
- Single GPU cannot handle large batches  $\rightarrow$  slow training

### 3. Solution: Data Parallelism with DDP

- More GPUs with fixed batch size  $\Rightarrow$  bigger effective batch size  $\Rightarrow$  faster convergence
- **DeepSpeed overkill**

# Distributed Data Parallel (DDP) Training

## Implementation

### 1. Process Initialization:

- SLURM launches 4 processes (1 per GPU)
- Each process: independent Python interpreter + CUDA context
- NCCL backend for GPU-to-GPU communication

### 2. Data Partitioning:

- Training set split across ranks (no overlap)
- Each rank: independent NeighborLoader for k-hop sampling
- 4-8 worker processes prefetch batches per GPU

### 3. Forward + Backward (Parallel):

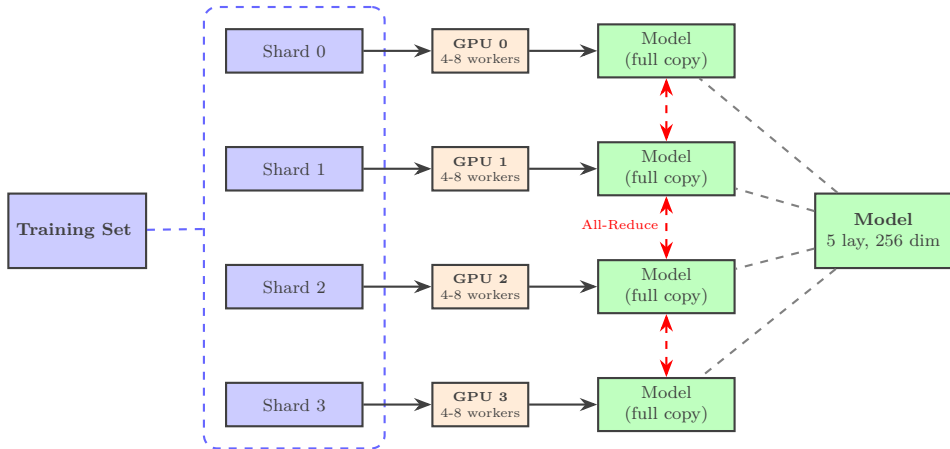
- Each GPU independently processes its batch
- Compute loss, backpropagate gradients

### 4. Gradient Synchronization:

- DDP all-reduce: average gradients across all GPUs
- Optimizer step with synchronized gradients

# DDP: Data Partitioned, Model Replicated

## Implementation



Distinct data shard + model copy per GPU. Gradient synchronization after each batch.

# Hyperparameters

🎯 **Goal:** Scalable and reproducible hyperparameter optimization

**Model Architecture:** GraphSAGE (SAGEConv + LayerNorm + ReLU)

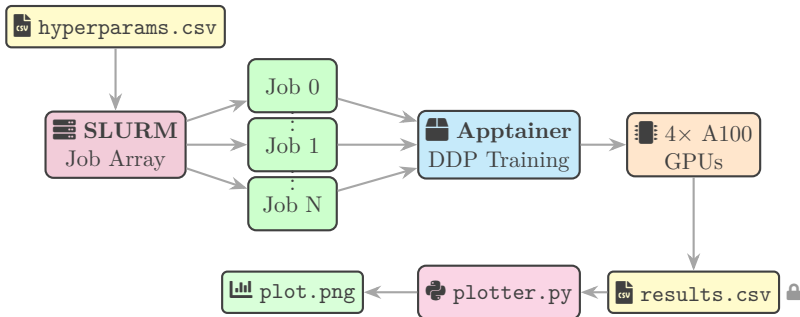
**Approach:**

- 📦 **Apptainer containers:** reproducible environments
- 📋 **SLURM job arrays:** parallel execution across configs
- 📄 **CSV-driven pipeline:** configuration & result logging












# Benchmark Pipeline

## Hyperparameters



# Tunable Hyperparameters

## Hyperparameters

	Parameter	Default	Range Tested
	batch_size	128	16–512
	accum_steps	5	1–5
	num_neighbors	[15,10,10,10,10]	various
	num_layers	5	fixed
	hidden_dim	256	fixed
	lr	0.003	fixed
	epochs	150	fixed
	patience	10	fixed
	dropout	0.5	fixed

 **Effective batch size:**

`batch_size × accum_steps × world_size`

 **Configs tested: 36**

16 batch · 16 neighbor · 4 scaling

# Results

# Benchmarking Focus

## Results

1. **Mini-batch sampling:** Batch size impact on time & memory
2. **Neighbor sampling:** Fanout strategies (accuracy vs. efficiency)
3. **GPU scaling:** 1/2/4/8 GPU distributed training

*Section 6.1*

***Mini-batch Sampling***

# Mini-batch Benchmarking: Configurations

## Results

What's being tested?

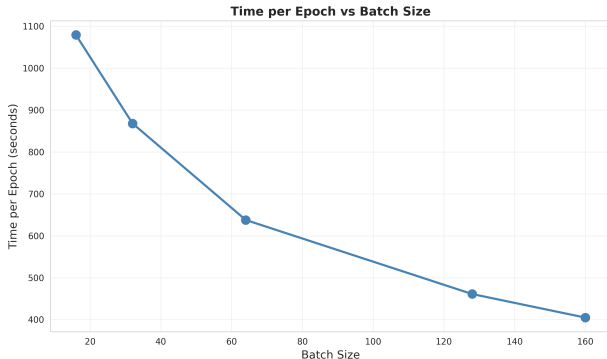
How does **batch size** and **gradient accumulation** affect:

- Training time per epoch
- Peak GPU memory usage
- Overall training efficiency

Batch Size	Accum. Steps
16	5
32	5
64	5
64	1, 2, 3, 4
<b>128</b>	<b>5</b>
128	1, 2, 3, 4
160	5
192	5
256	5
512	5

# Batch Size: Training Time

## Results

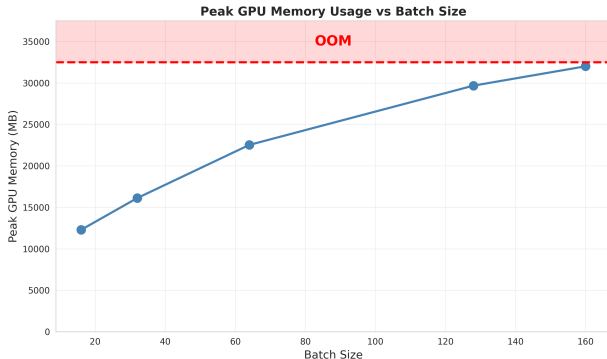


### Key Findings:

- Inverse relationship: larger batch = faster training
- Better GPU utilization with larger batches
- Reduced forward pass / backward pass overhead

# Batch Size: Memory Consumption

## Results



### Key Findings:

- Max batch size: 160
- Uses ~32GB (7.5GB system overhead)
- Bigger batches go OOM



*Section 6.2*

***Neighbor Sampling***

# Neighbor Sampling: Configurations

## Results

What's being tested?

How do different **fanout** strategies affect:

- Test accuracy
- Training time per epoch
- GPU memory usage

**Fanout** = neighbors sampled per layer

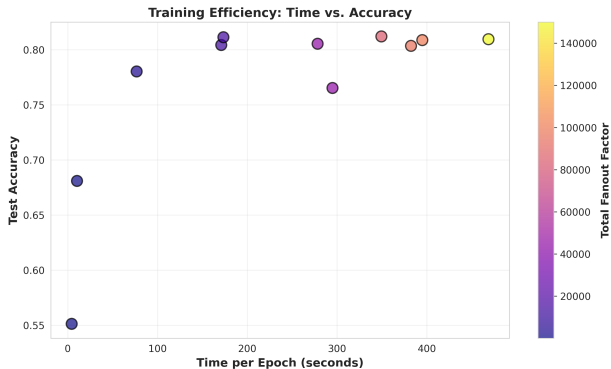
**Total Fanout** = product across all layers

$$F_{\text{total}} = \prod_{i=1}^5 L_i$$

L1	L2	L3	L4	L5
50	0	0	0	0
15	0	0	0	0
15	10	0	0	0
15	10	10	0	0
15	10	10	10	0
<b>15</b>	<b>10</b>	<b>10</b>	<b>10</b>	<b>10</b>
10	10	10	10	10
12	12	12	12	12
15	15	15	15	15
20	15	10	5	3
15	15	10	5	5
15	15	15	5	5
12	11	10	9	8
5	5	5	5	5
3	5	10	15	20
30	10	5	3	3

# Neighbor Sampling: Accuracy vs. Time

## Results

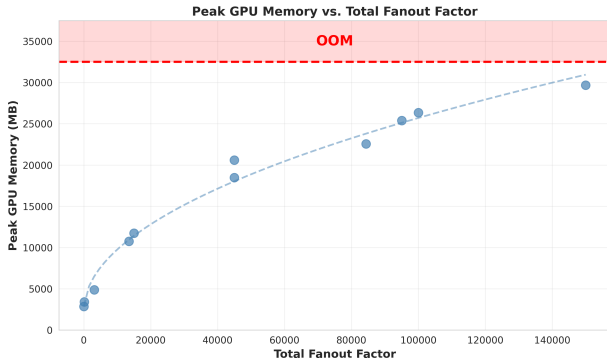


### Key Findings:

- Upper-left = optimal
- Moderate fanouts best balance
- Aggressive sampling hurts accuracy or stability
- Very high fanouts: no proportional gains

# Neighbor Sampling: Memory vs. Fanout

## Results



### Key Findings:

- Rapid memory growth with fanout
- Must balance receptive field vs. memory
- High fanouts hit memory limits and go OOM

# Top 5 Neighbor Configurations

## Results

Rank	Neighbor Sampling	Test Acc	Time (h)
1	[15, 15, 15, 5, 5]	0.8121	14.56
2	[15, 10, 10, 10, 0]	0.8114	<b>7.22</b>
<b>3</b>	<b>[15, 10, 10, 10, 10]</b>	<b>0.8096</b>	<b>19.53</b>
4	[10, 10, 10, 10, 10]	0.8088	16.46
5	[20, 15, 10, 5, 3]	0.8055	11.60

**Key insight:** [15,10,10,10,0] nearly matches accuracy, and **2.7×** faster

## *Section 6.3*

# *GPU Scaling*

# GPU Scaling: Configurations

## Results

What's being tested?

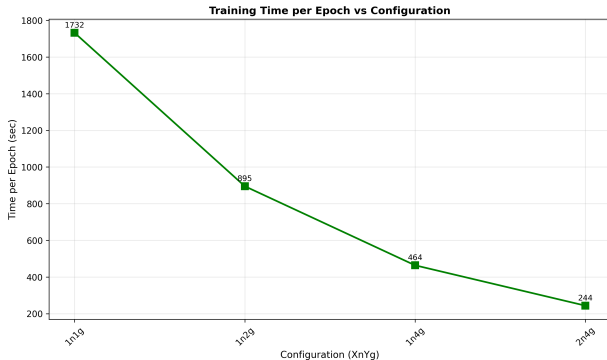
How does **distributed training** scale across:

- 1/2/4 GPUs per node
- 4 GPUs across 2 nodes (8 GPUs total)

Configuration	Setup
1 GPU	1 node
2 GPUs	1 node
4 GPUs	1 node
8 GPUs	2 nodes
<b>Fixed Hyperparameters</b>	
Batch size	128
Neighbors	[15,10,10,10,10]
Epochs	10

# GPU Scaling: Training Time

## Results



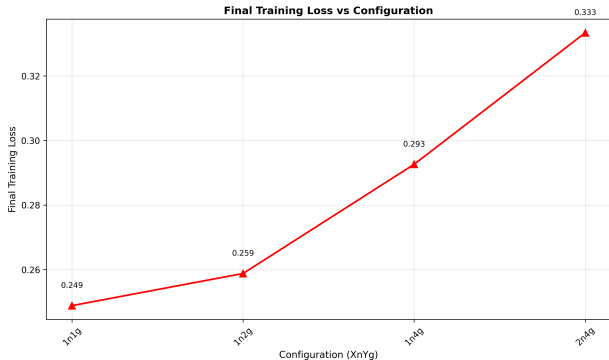
### Key Findings:

- **1→2 GPUs:**  
 $1.93\times$  (96.5% efficiency)
- **1→4 GPUs:**  
 $3.73\times$  (93% efficiency)
- **1→8 GPUs:**  
 $7.1\times$  (89% efficiency)



# GPU Scaling: Training Loss

## Results



### Key Findings:

- Loss increases with more GPUs
- Cause: larger effective batch size
- Reduced gradient noise

# Discussion

# Bottlenecks Discovered

## Discussion

- **Memory wall:** Neighbor sampling expansion dominates memory usage
  - Fanout  $[15,10,10,10,10] = 150\text{k}$  neighbors per seed node
  - $128 \text{ batch size} \times 150\text{k expansion} = 19.2\text{M nodes per batch}$
  - Sampled subgraph + activations  $\gg$  model parameters (8MB)
- **Inter-node communication:** InfiniBand NCCL all-reduce overhead
  - 4 GPUs (1 node): 93% efficiency
  - 8 GPUs (2 nodes): 89% efficiency (4% degradation)
- **Loss convergence trade-off:** Larger effective batch sizes hurt convergence
  - Training loss increases with GPU count
  - Reduced gradient noise

# What Worked Well

## Discussion

- **DDP implementation:** Seamless gradient synchronization across GPUs
  - PyTorch DDP handles all-reduce automatically
  - Minimal code changes from single-GPU baseline
- **Strong intra-node scaling:** Excellent up to 4 GPUs
  - Linear speedups:  $1 \rightarrow 4$  GPUs =  $3.73\times$  (93% efficiency)
  - PCIe/NVLink bandwidth sufficient for this dataset size
- **Containerization:** Reproducibility & portability
  - Apptainer/Singularity ensures consistent environment
  - Complex PyG dependencies properly managed

## Implementing High-Performance Systems:

1. **Profile first, optimize second:** Memory bound, not compute
2. **Trade-offs are everywhere:** No free lunch → accuracy, speed, memory, communication always compete
3. **Reproducibility matters:** Containerization and fixed seeds essential for scientific HPC work

Any Question?

- [1] Aditya Grover and Jure Leskovec. “node2vec: Scalable Feature Learning for Networks”. In: *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 2016, pp. 855–864.
- [2] William L. Hamilton, Rex Ying, and Jure Leskovec. “Inductive Representation Learning on Large Graphs”. In: *Advances in Neural Information Processing Systems (NeurIPS)*. Vol. 30. 2017, pp. 1024–1034. URL: <https://arxiv.org/abs/1706.02216>.
- [3] Thomas N. Kipf and Max Welling. “Semi-Supervised Classification with Graph Convolutional Networks”. In: *arXiv preprint arXiv:1609.02907* (2016).
- [4] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. “DeepWalk: Online Learning of Social Representations”. In: *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 2014, pp. 701–710.