*P*³: Distributed Deep Graph Learning at Scale

Swapnil Gandhi and Anand Padmanabha Iyer, *Microsoft Research*

https://www.usenix.org/conference/osdi21/presentation/gandhi

This paper is included in the Proceedings of the 15th USENIX Symposium on Operating Systems Design and Implementation.

July 14–16, 2021

978-1-939133-22-9

Open access to the Proceedings of the 15th USENIX Symposium on Operating Systems Design and Implementation is sponsored by USENIX.

P^3: Distributed Deep Graph Learning at Scale

Swapnil Gandhi*
Microsoft Research

Anand Padmanabha Iyer
Microsoft Research

Abstract

Graph Neural Networks (GNNs) have gained significant attention in the recent past, and become one of the fastest growing subareas in deep learning. While several new GNN architectures have been proposed, the scale of real-world graphs—in many cases billions of nodes and edges—poses challenges during model training. In this paper, we present P^3, a system that focuses on scaling GNN model training to large real-world graphs in a *distributed* setting. We observe that scalability challenges in training GNNs are fundamentally different from that in training classical deep neural networks and distributed graph processing; and that commonly used techniques, such as intelligent partitioning of the graph do not yield desired results. Based on this observation, P^3 proposes a new approach for distributed GNN training. Our approach effectively eliminates high communication and partitioning overheads, and couples it with a new *pipelined push-pull* parallelism based execution strategy for fast model training. P^3 exposes a simple API that captures many different classes of GNN architectures for generality. When further combined with a simple caching strategy, our evaluation shows that P^3 is able to outperform existing state-of-the-art distributed GNN frameworks by up to 7×.

1 Introduction

Deep learning, in the form of Deep Neural Networks (DNNs), has become the de-facto tool for several challenging applications in diverse fields such as computer vision [27], speech recognition [28] and natural language processing [18], where they have produced results on par with human experts [9]. In the recent past, there has been a significant interest in Graph Neural Networks (GNNs)—neural networks that operate on *graph structured data*—which has made them one of the fastest growing subareas in deep learning [25]. Due to the expressiveness of graphs in capturing the rich relational information between input elements, GNNs have enabled breakthroughs in many important domains including recommendation systems [51, 66], knowledge graphs [53], and drug discovery [46, 58].

In a GNN, the nodes in the input graph are associated with features and labels. Typical tasks in GNNs include node classification (predicting the class label of a node) [41], link prediction (predicting the possibility of a link between given nodes) [70] and graph classification (predicting the class label of a graph) [8]. To do these tasks, GNNs combine feature information with graph structure to learn *representations*—low-dimensional vector embeddings—of nodes. Thus, learning such *deep encodings* is the key goal of GNNs. Several novel GNN architectures exist today, including GraphSAGE [24], Graph Convolution Networks (GCNs) [17, 41] and Graph Attention Networks (GATs) [59]. While each have their own unique advantages, they fundamentally differ in *how* the graph structure is used to learn the embeddings and *what* neural network transformations are used to aggregate neighborhood information [64].

At a high level, GNNs learn embeddings by combining iterative graph propagation and DNN operations (e.g., matrix multiplication and convolution). The graph structure is used to determine *what* to propagate and neural networks direct *how* aggregations are done. Each node creates a k-hop *computation graph* based on its neighborhood, and uses their features to learn its embedding. One of the key differentiators between training GNNs and DNNs is the presence of dependencies among data samples: while traditional DNNs train on samples that are independent of each other (e.g., images), the connected structure of graph imposes dependencies. Further, it is common to have a large number of dense features associated with every node—ranging from 100s to several 1000s [29, 66, 68]—in the graph. Due to this, the k-hop computation graphs created by each node can be prohibitively large. Techniques such as *neighborhood sampling* [24] help to some extend, but depending on the graph structure, even a sampled computation graph and associated features may not fit in the memory of a single GPU, making scalability a fundamental issue in training GNNs [71]. With the prevalence of large graphs, with *billions* of nodes and edges, in academia and the industry [55], enabling GNN training in a *distributed* fashion[1] is an important and challenging problem.

In this paper, we propose P^3,[2] a system that enables efficient *distributed* training of GNNs on large input graphs. P^3 is motivated by three key observations. First, due to the data dependency, we find that *in distributed training of GNNs, a major fraction of time is spent in network communication to generate the embedding computation graph with features*. Second, we notice that relying on distributed graph processing techniques such as advanced partitioning schemes, while useful in the context of graph processing, do not benefit GNNs

*Work done during an internship at Microsoft Research.

[1] Using more than one machine, each with 1 or more GPUs.
[2] for *Pipelined Push-Pull*.

and in many cases could be detrimental. Finally, due to the network communication issue, we observed that GPUs in distributed GNN training are underutilized, and spend as much as 80% of the time blocked on communication (§2). Thus, P^3 focuses on techniques that can reduce or even eliminate these inefficiencies, thereby boosting performance.

P^3 is not the first to address GNN scalability challenges. While there are many available frameworks for GNN training, a majority of them have focused on single machine multi-GPU training and limited graph sizes [20, 45, 47]. Popular open-source frameworks, such as the Deep Graph Library (DGL) [1] have incorporated distributed training support. But as we show in this paper, it faces many challenges and exhibits poor performance due to high network communication. ROC [36] is a recent system that shares the same goal as P^3 but proposes a fundamentally different approach. ROC extensively optimizes GNN training using a sophisticated *online* partitioner, memory management techniques that leverage CPU and GPU, and relies on hardware support such as high speed interconnects (NVLink and InfiniBand). In contrast, P^3 only assumes PCIe links and Ethernet connection, and doesn't rely on any intelligent partitioning scheme. During training, ROC requires movement of features across machines, while in P^3, features are never transmitted across the network. Finally, our evaluation datasets are significantly larger than ROCs, which helped us uncover several challenges that may have been missed with smaller graphs.

why no transmit?

To achieve its goal, P^3 leverages a key characteristic of GNNs: unlike traditional DNNs where the data samples (e.g., images) are small and model parameters are large (e.g., 8 billion for Megatron [56], 17 billion for TuringNLG [7]), GNNs have small model parameters but large data samples due to the dense feature vectors associated with each node's sampled computation graph. As a result, movement of these feature vectors account for the majority of network traffic in existing GNN frameworks. In P^3, we avoid movement of features entirely, and propose distributing the graph structure and the features across the machines *independently*. For this, it only relies on a random hash partitioner that is fast, computationally simple and incurs minimal overhead. Additionally, the hash based partitioning allows work balance and efficiency when combined with other techniques P^3 incorporates (§3.1).

During embedding computation, P^3 takes a radically different approach. Instead of creating the computation graph by pulling the neighborhood of a node and the associated features, P^3 only pulls the graph structure, which is significantly smaller. It then proposes *push-pull* parallelism, a novel approach to executing the computation graph that combines *intra-layer model parallelism* with *data parallelism*. P^3 *never* moves features across the network, instead it *pushes* the computation graph structure in the most compute intensive layer (layer 1) to all the machines, and thereafter executes operations of layer 1 using *intra-layer model parallelism*. It then *pulls* much smaller *partial activations*, accumulates them, and

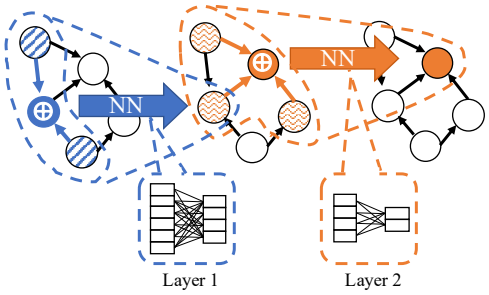


Figure 1: A two-layer GNN that uses DNN at each layer along with iterative graph propagation for learning.

proceeds to execute operations of the remaining $k-1$ layers using *data parallelism* (§3.2).

Due to the partitioning strategy and the push-pull parallelism based execution, P^3 is able to use a simple pipelining technique that overlaps most of the computation and communication efficiently, thus effectively hiding (the already small) communication latencies (§3.3). Further, the partitioning strategy also enables P^3 to propose a simple caching mechanism that greedily caches graph and/or feature partitions on multiple machines if memory permits for further reduction in network communication (§3.4). P^3's proposed techniques are general and are applicable to several state-of-the-art GNN architectures. P^3 also wraps all these optimizations in a simple `P-TAGS` API (`partition`, `transform`, `apply`, `gather`, `scatter` and `sync`) that developers can use to write new GNN architectures that can benefit from its techniques (§3.5).

The combination of these techniques enable P^3 to outperform DGL [1], a state-of-the-art distributed GNN framework, by up to 7×. Further, P^3 is able to support much larger graphs and scale gracefully (§5).

We make the following contributions in this paper:

- We observe the shortcomings with applying distributed graph processing techniques for scaling GNN model training (§2). Based on this, P^3 takes a radically new approach of relying only on *random hash* partitioning of the graph and features *independently*, thus effectively eliminating the overheads with partitioning. (§3.1)
- P^3 proposes a novel *hybrid parallelism* based execution strategy that combines intra-layer model parallelism with data parallelism that significantly reduces network communication and allows many opportunities for pipelining compute and communication. (§3.2)
- We show that P^3 can scale to large graphs gracefully and that it achieves significant performance benefits (up to 7× compared to DGL [1] and up to 2× compared to ROC [36]) that increase with increase in input size. (§5)

2 Background & Challenges

We begin with a brief background on GNNs, and then motivate the challenges with distributed training of GNNs.

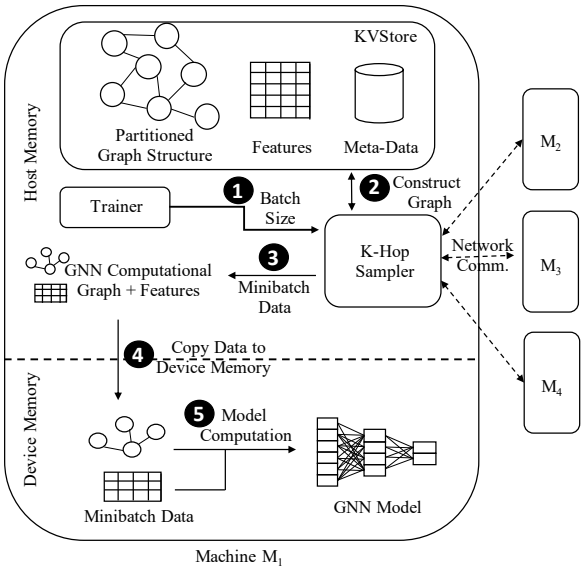


Figure 2: Existing GNN frameworks combine distributed graph processing and DNN techniques.

2.1 Graph Neural Networks

A GNN is a neural network that operates on graph structured data as input. The input graphs contain nodes (entities), and edges (relation between nodes) and features for all nodes. The basic operation in GNNs is to obtain the *representations* of nodes in the graph. They map nodes to a d-dimensional embedding space such that similar nodes (e.g., by proximity) in the graph are embedded close to each other. To obtain these embeddings, GNNs combine feature information associated with the nodes and the graph structure using information propagated and transformed from its neighborhood. In computing the embedding, the graph structure indicates *what* is propagated, and a neural network is used to determine *how* the propagated information is transformed. The exact neighborhood from which the embedding is derived is configurable, and typically GNNs use k (usually 2 or more) hops from a node [26]. The neural network which transforms information at each hop is called a *layer* in the GNN, hence a 2-hop (k-hop) neighborhood translates to a 2 (k) layer GNN (fig. 1).

Theoretically, the embedding z_v of node v after k layers of neighborhood aggregation can be obtained as h_v^k [24], where:

$$h_{N(v)}^k = \text{AGGREGATE}^{(k)}\left(\left\{h_u^{k-1} \mid u \in N(v)\right\}\right) \quad (1)$$

$$h_v^k = \sigma\left(W_k \cdot \text{COMBINE}^{(k)}(h_v^{k-1}, h_{N(v)}^k)\right) \quad (2)$$

Here, h_v^i is the representation of node v after i layers of aggregation and W_i is the trainable weight matrix that is shared by all nodes for layer i ($i \geq 1$). h_v^0 is initialized using the node features. The choice of $\text{AGGREGATE}^{(k)}(.)$ and $\text{COMBINE}^{(k)}(.)$ is crucial for how the layers are defined and the embeddings are computed.

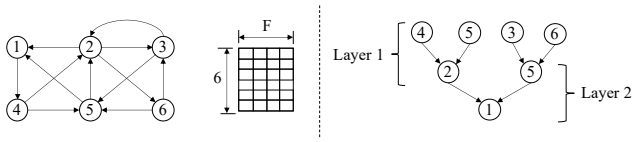


Figure 3: Sampling a two-layer GNN computation graph by restricting the neighborhood size to 2 (minibatch size: 1).

2.2 Distributed Training of GNNs

Existing frameworks for training GNNs, such as the Deep Graph Library (DGL) [1], support distributed training by combining distributed graph processing techniques with DNN techniques, as shown in fig. 2. The input graph along with the features is partitioned across machines in the cluster. Given a batch size (1), the computation graph for each node, commonly referred to as a *training sample*, in the batch is generated by pulling the k-hop neighborhood of each node along with the associated features (2). This requires communication with other machines in the cluster. Once the computation graphs are constructed, standard DNN techniques such as *data parallelism* is used to execute the computation—minibatches are created and copied to the GPU memory (4), and then the model computation is triggered (5).

2.3 Challenges in Distributed GNN Training

There are several challenges that need to be addressed to make distributed GNN training efficient.

2.3.1 Challenge #1: Communication Bottlenecks Due to Data Dependencies

Unlike traditional DNNs where the training data are independent of each other (e.g., images), GNNs impose a dependency between the training inputs in the form of graph structure. Thus, even though provided batch size could be small (e.g., 1000), the computation graph samples could be exponentially larger due to the k-hop neighborhood and the associated features. A major reason for such large size is not the graph structure in itself, but the features, whose sizes typically range in 100s to several 1000s [29, 41, 66, 68]. In real world graphs consisting of billions of nodes and edges [15, 55], the 2-hop neighborhoods could be up to an order of magnitude larger than the 1-hop neighborhood [43]. When combined with the features, the resulting computation graph may easily exceed the memory of a single GPU or even main memory of a server. A common technique used to address such *neighborhood explosion* is sampling [24]. That is, instead of getting all the neighbors of the node in each hop, we select only a fixed number. An example is shown in fig. 3, where node 1's 2-hop computation graph is generated by sampling two neighbors at each hop. However, even with sampling, the size of the computation graph could grow substantially, based on the sampling used and the number of layers in the GNN. Since these neighborhood nodes and their features must be obtained through the network, distributed training of GNNs spend a major fraction of time in network communication.

Scheme	Time(s)	Memory(GB)	Epoch(s)
Hash [48]	2.87	58	9.833
METIS [38]	4264	63	5.295
RandomVertexCut [22]	36.95	185	5.494
GRID [23]	51.82	128	6.866
3D [69]	134	118	6.027

Table 1: Partitioning techniques are not effective in GNNs. All schemes, except hash, reduce the epoch time, but at the cost of significant partitioning time or memory overheads.

2.3.2 Challenge #2: Ineffectiveness of Partitioning

Partitioning is a common approach used to achieve scalability in distributed graph processing, and existing GNN frameworks leverage popular partitioning strategies to distribute the graph and features across machines. However, there are two shortcomings with this approach.

First, many partitioning scheme incur a cost in terms of computation and/or memory overhead. In table 1, we show the partitioning time, memory consumption and the time to complete one epoch of training on a representative GNN, GraphSAGE [24] for four different partitioning schemes: Hash [48], which partitions nodes using random hashing, METIS [38], a balanced min edge-cut partitioner, RandomVertexCut [22] and GRID [23], are vertex-cut partitioners, and 3D [69], a recently proposed scheme for machine learning workloads. We see that the best performing partitioning schemes (e.g., edge-cut) incur high computation overheads. Computationally faster schemes incur either high memory overhead (due to replication, e.g., vertex-cut) or performance hit.

Second, the benefits of partitioning are severely limited as the layers in the GNN increase. Recall that GNNs use k-hop neighborhood to compute the embedding. While partitioning schemes reduce communication, *they only optimize communication at the first hop*. Thus, when the number of layers increase, all partitioning schemes fail.

2.3.3 Challenge #3: GPU Underutilization

Existing GNN frameworks utilize DNN techniques, such as *data parallelism* to train GNN models. In data parallel execution, each machine operates on a different set of samples. However, due to the data dependency induced communication bottleneck described earlier, we observed that in distributed GNN training using the popular framework DGL, GPUs are only being utilized $\approx 20\%$ of the time. For a large fraction ($\approx 80\%$) of the time, GPUs are waiting on communication. Recent work has reported data copy to be the major bottleneck in training GNNs in *single machine multi-GPU* setup [45], but we found that data copy only accounts for 5% of the time while training GNNs using *distributed multi-GPU* setup. We note that the proposed techniques in [45] are orthogonal to our work and can benefit if applied to P^3. Thus, GPUs are heavily underutilized in distributed GNN training due to network communication. Alternative parallelism techniques, such as model parallelism do not work for GNNs. This is because for each layer, they would incur intra-layer communication in-addition to data dependency induced communication and thus perform even worse compared to data parallelism.

3 P^3: Pipelined Push-Pull

P^3 proposes a new approach to distributed GNN training that reduces the overhead with computation graph generation and execution to the minimum. To achieve this, P^3 incorporates several techniques, which we describe in detail in this section.

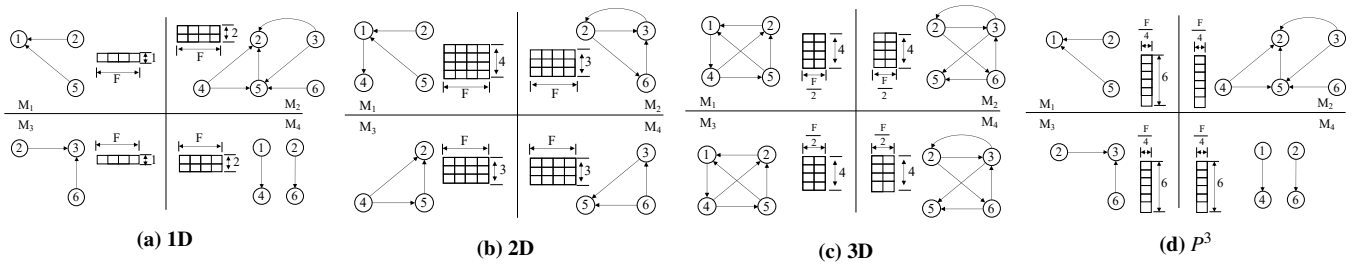
3.1 Independent Hash Partitioning Graph & Features

As we show in §2, partitioning of the input graph in an intelligent manner doesn't benefit GNN architectures significantly due to the characteristics of GNNs. Hence, in P^3, we use the simplest partitioning scheme and advocate for *independently* partitioning the graph and its features.

The nodes in the input graph are partitioned using a random hash partitioner, and the edges are co-located with their incoming nodes. This is equivalent to the commonly used 1D partitioning scheme available in many distributed graph processing frameworks [22, 67], and is computationally simple. Unlike other schemes (e.g., 2D partitioning), this scheme doesn't require any preprocessing steps (e.g., creating local ids) or maintaining a separate routing table to find the partition where a node is present, it can simply be computed on the fly. Note that this partitioning of the graph is only to ensure that P^3 can support large graphs. In several cases, the graph structure (nodes and edges without the features) of real-world graphs can be held in the main memory of modern server class machines. In these cases, P^3 can simply replicate the entire graph structure in every machine which can further reduce the communication requirements.

While the graph structure may fit in memory, the same cannot be said for input features. Typical GNNs work on input graphs where the feature vector sizes range in 100s to several 1000s [29, 41, 66, 68]. P^3 partitions the input features along the feature dimension. That is, if the dimension of features is F, then P^3 assigns F/N features of every node to each of the machines in a N machine cluster. This is in contrast to existing partitioning schemes tuned for machine learning tasks, including the recently proposed 3D partitioning scheme [69]. Figure 4 shows how P^3 partitions a simple graph in comparison with existing popular partitioning schemes.

As we shall see, this independent, simple partitioning of the graph and features enable many of P^3's techniques. Breaking up the input along the feature dimension is crucial, as it enables P^3 to achieve work balance when computing embeddings; as the hash based partitioner ensures that the nodes and features in the layers farther from the node whose embedding is computed to be spread across the cluster evenly. The simplicity of independently partitioning the structure and features also lets P^3 cache structure and features independently in its caching mechanism (§3.4).



(a) 1D (b) 2D (c) 3D (d) P^3

Figure 4: P^3 *independently* partitions the graph structure and the associated features (shown in fig. 3) using simple random hash partitioning in contrast to more intelligent schemes. This allows P^3 to achieve work balance, and enables many of its techniques.

3.2 Push-Pull Parallelism

With the input graph and features partitioned, P^3 adopts the common, minibatch centric computation for GNNs, similar to existing GNN frameworks, where it first generates the computation graph for a node and then executes it. We use fig. 5 to explain this in detail.

3.2.1 Computation Graph Generation

At the beginning of every minibatch, each node whose embedding is being computed generates its computation graph. To do so P^3 *pulls* [3] the k-hop neighborhood for the node. If the GNN architecture supports a sampling based embedding computation, P^3 pulls the sampled k-hop neighborhood, otherwise it pulls the full k-hop neighborhood. Note that unlike existing GNN frameworks, the *features are not pulled* in either case. This significantly reduces the network communication necessary for creating the computation graph. If the entire graph structure is available in every machine, this is a local operation, otherwise it results in minimal network communication as the graph structure is very light weight. At the end of this process, P^3 ends up with the k-layer computation graph of each node in the minibatch at the machine which owns the node (e.g., the four *samples* in fig. 5 correspond to computation graphs of four nodes in the minibatch). Note that existing GNN frameworks *pulls* features in addition to the structure, so in these frameworks, the machine owning the node ends up with both the computation graph and all the features necessary for embedding computation.

In the case of existing GNNs, each machine can now independently execute the complete computation graph with features it obtained in a data parallel fashion, starting at layer 1 and invoking global gradient synchronization at each layer boundary as shown in fig. 5a in the backward pass. However, since P^3 does not move features, the computation graphs cannot be executed in a data parallelism fashion. Here, P^3 proposes a hybrid parallelism approach that combines model parallelism and data parallelism, which we term *push-pull parallelism*. While model parallelism is rarely used in traditional DNNs due to the underutilization of resources and difficulty in determining how to partition the model [49], P^3 uses it to its advantage. Due to the nature of GNNs, the model (embedding computation graphs) is easy to partition cleanly since the boundaries (hops) are clear. Further, due to P^3's partitioning strategy, model parallelism doesn't suffer from underutilization of resources in our context.

[3] *Pulling* refers to copying, possibly over the network if not local.

3.2.2 Computation Graph Execution

To start the execution, P^3 first *pushes* the computation graph for layer 1 to all the machines, as shown in ❶ in fig. 5b. Note that layer 1 is the most compute intensive, as it requires input features from layer 0 (having most fan-out) which are evenly spread in P^3 due to our partitioning scheme. Each machine, once it obtains the computational graph, can start the forward pass for layer 1 in a *model parallel* fashion (layer 1_M). Here, each machine computes *partial* activations for layer 1_M using the partition of input features it owns (❷). Since all GPUs in the cluster collectively execute the layer which requires input from the most fan-out, this avoids underutilization of GPUs. We observed that GPUs in existing GNN frameworks (e.g., DGL) spend $\approx 80\%$ of the time blocked on network compared to $\approx 15\%$ for P^3. Once the partial activations are computed, the machine assigned to each node in our hash partitioning scheme *pulls* them from all other machines. The node receiving the partial activations aggregates them using a reduce operation (❸). At this point, P^3 switches to *data parallelism* mode (layer 1_D). The aggregated partial activations are then passes through the rest of layer 1_D operations (if any, e.g., non-linear operations that cannot be partially computed) to obtain the final activations for layer 1 (❹). The computation proceeds in a *data-parallel* fashion to obtain the embedding at which point the forward pass ends (❺).

The backward pass proceeds similar to existing GNN frameworks in a *data parallel* fashion, invoking global gradient synchronizations until layer 1_D (❻). At layer 1_D, P^3 *pushes* the error gradient to all machines in the cluster (❼) and switches to *model parallelism*. Each machine now has the error gradients to apply the backward pass for layer 1_M *locally* (❽) and the backward pass phase ends.

While the partial activation computation in a *model parallel* fashion seemingly works in the general sense, they are restricted to transformations that can be aggregated from partial results. However, in certain GNN architectures

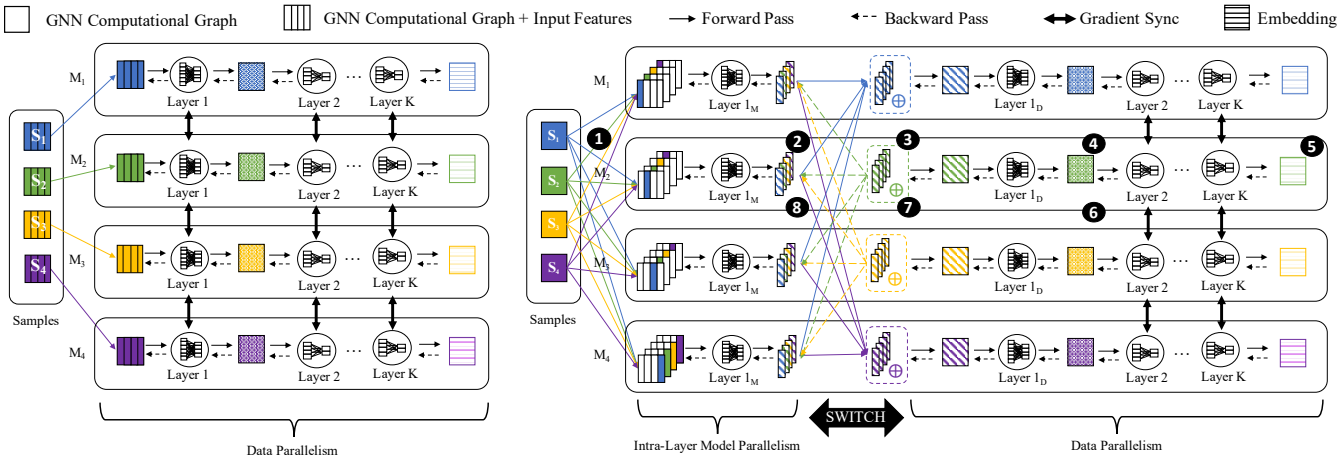


Figure 5: How existing GNN frameworks generate and execute computation graphs (left) and how P^3 does it (right) (§3.2).

(e.g., GAT [59]), layer 1_M in itself may introduce non-linear transformations. P^3 relies on developer input to determine the tensors that require global synchronizations during the model parallel execution to ensure correctness (§3.5).

At a first glance, the additional steps in P^3, namely the need to push graph structure in layer 1, aggregation of partial activations during the forward pass and the additional movement of gradients in the backward pass may seem like overheads that may lead to inefficiencies compared to simply pulling the features along with the graph structure and executing everything locally as in existing GNN frameworks. However, P^3's approach results in significant savings in network communication. First, P^3 doesn't pull features at all which tremendously reduces network traffic—typically the 2-hop neighborhood in the GNN computation graphs is an order of magnitude more than the 1-hop neighborhood. Second, regardless of the number of layers in the GNN, only layer 1 needs to be partially computed and aggregated. Finally, the size of the activations and gradients are small in GNNs (due to the smaller number of hidden dimensions), thus transferring them incurs much less overhead compared to transferring features.

To illustrate this, we use a simple experiment where we run a representative GNN, a 2-layer GraphSAGE [24] on the open-source OGB-Product [29] dataset on 4 machines. We pick 1000 labeled nodes to compute the embeddings and use neighborhood sampling (fan-out:25,10). The nodes are associated with feature vectors of size 100, and there are 16 hidden dimensions. At layer 0 (2-hops), there are 188339 nodes and at layer 1 (1-hop) there are 24703 nodes. Pulling features along with graph structure would incur 71.84 MB of network traffic. On the other hand, the activation matrix is of size input $\times$ hidden dimension. P^3 only needs to transfer the partial activations from 3 other machines, thus incurring just 5 MB ($3 \times 24703 \times 16$). Hence, by distributing the activation computation of the layer that holds the largest number of features, P^3 is able to drastically reduce network communication.

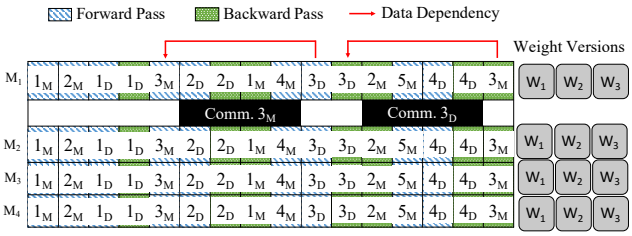


Figure 6: We employ a simple pipelining mechanism in P^3, inspired by the work in PipeDream [49]. This allows P^3 to effectively hide communication latencies by overlapping communication with computation.

3.3 Pipelining

Although P^3's push-pull parallelism based GNN computation graph creation and execution incurs less network communication compared to existing GNN frameworks, it needs to communicate more times: P^3 needs to push the graph structure of layer 1, pull partial activations in the forward pass and finally push the gradients in the backward pass. Further, since P^3 focuses on distributed settings, data copy is necessary between CPU and GPU. As a result, the computation is stalled during communication unless they are overlapped using pipelining techniques. Note that current GNN frameworks (e.g., DGL) already overlap computation and communication—while the CPU is busy creating the computation graph, the GPU is used to execute an already prepared mini batch.

In P^3, we employ a simple pipelining mechanism, inspired by PipeDream's pipeline parallelism [49]. Due to the approach we take in P^3 to enable push-pull parallelism, namely switching between model and data parallelism at layer 1, P^3 needs to execute four phases per minibatch: a model parallel

API	Description
`partition(graph, feat, topo_part_fn, ft_part_fn)`	Partition graph and input features *independently* using `topo_part_fn` and `ft_part_fn` respectively.
`scatter(graph, feat, udf)` → `msg`	Generates message `msg` by combining `src_ft`, `edge_ft`, and `dst_ft`.
`gather(msg, udf)` → `a_ft`	Computes aggregated neighbourhood representation from incoming messages.
`transform(v_ft, a_ft)` → `t_ft`	Computes partial output representation from partial input representation; requires sync.
`sync(t_ft, op='sum')` → `sync_ft`	Accumulates partial representations using user-defined arithmetic operation.
`apply(v_ft, sync_ft)` → `ft`	Computes output representation from input representation.

Table 2: The simple P-TAGS API exposed by P^3. Developers can use this API to scale up new GNN architectures.

phase in the forward pass, a data parallel phase in the forward pass, a data parallel phase in the backward pass and finally a model parallel phase in the backward pass. This provides us the opportunity to schedule 3 minibatches of computation before a data dependency is created between phases. Thus, we overlap computations between these, as shown in fig. 6. As shown, in the forward pass, the data parallel phase of minibatch 3 (denoted as 3_D) has a data dependency on the model parallel phase (3_M) in the forward pass. Hence, when phase 3_M starts communication, we schedule two forward and two backward passes from other minibatches. This 2 forward, 2 backward static scheduling strategy allows us to avoid stalls. We currently use static pipeline scheduling—while a profiling based methodology to identify a pipeline schedule may provide benefits, we defer it to a future work.

Bounded Staleness The main challenge with using pipelining as described above is the introduction of *staleness* which can be characterized by *pipeline delay*: the number of optimizer steps that have passed between when a weight is read for computing gradient and when that computed gradient is used for updating the weight. This delay is bound by the number of minibatches in the pipeline at any time and also exists in prior work [49, 52]. For P^3, this delay is fixed and bound to three, resulting in weight updates of the form:

$$w_{t+1} = w_t - \alpha \cdot \nabla f(w_{t-3}) \tag{3}$$

where, w_t is weight values after t optimizer steps, ∇f is the gradient function, α is learning rate and w_{t-3} the weight used in forward and backward passes. While unbounded stale gradient updates can negatively affect the statistical efficiency of the network, preventing convergence and producing models with lower accuracy, bounded delay enables P^3 reach target accuracy in the same number of epochs as data parallelism.

Memory Overhead While P^3's peak memory footprint is relatively on par to data parallelism, stashed weights can result in additional memory overhead. Presently, GNN models typically contain only a couple of layers of small DNN models, and therefore even with weight stashing the overhead is relatively small. This however may change in future as GNN models become larger and complex. P^3's memory overhead can be further reduced by leveraging prior work aimed at decreasing memory footprint of training DNN models [33, 34].

3.4 Caching

The use of *independent* partitioning for the graph structure and features allows P^3 to employ a simple caching scheme that can reduce the already minimal communication overhead. This is based on the observation that depending on the graph and the size of the features, either the graph or the features may be accommodated in less machines than what is available. By default, the features and the graph are partitioned without duplication across all available machines. However, when host memory is available, P^3 uses a simple greedy approach to utilize all the available free memory by caching the partitions of the graph and/or features on multiple machines using a user-defined setting. In its current state, we do simply caching, where we try to fit the input in the minimum number of machines, and create copies (caches) of partitions on other machines. We assume homogeneous machines, which is typically the standard in DNN/GNN training [35]. We believe that there are opportunities to design a better caching scheme, and plan to explore it in the future.

3.5 P^3 API

P^3 wraps its independent partitioning strategy, pipelined push-pull parallelism and caching in a simple API, which developers can use to speed up new GNN architectures. The API, shown in table 2, consists of the following six functions :

- `partition` is a user-provided composite function which independently partitions graph topology and input features across machines. This step is essential to balance load and reduce communication.
- `scatter` uses a user-provided message function defined on each edge to generate a message by combining the edge representation with the representations of source and destination vertices.
- `gather` uses a user-provided commutative and associative function (e.g. sum, mean) defined on each vertex to compute the neighborhood representation by aggregating incoming messages.
- `transform` is a user-provided composite function defined on each vertex to generate partial representation by applying zero or more element-wise NN operations[4] (e.g. add),

[4]Element-wise NN operation operates on elements occupying the same index position within respective tensors.

```
class GraphSAGE(nn.module):
    def __init__(in_ft, out_ft):
        fc_self = fc_neigh = Linear(in_ft, out_ft)
    # Generates message
    def scatter_udf(s_ft, e_ft, d_ft): return s_ft
    # Aggregates messages
    def gather_udf(msg): return mean(msg)
    # Computes partial activation; requires sync.
    def transform(v_ft, a_ft):
        return fc_self(v_ft) + fc_neigh(a_ft)
    # Computes vertex representation
    def apply(v_ft, t_ft):
        return ReLU(t_ft)
    def forward(graph, feat):
        graph['m'] = scatter(graph, feat, scatter_udf)
        graph['n_p'] = gather(graph['m'], gather_udf)
        graph['n_p'] = transform(feat, graph['n_p'])
        return apply(feat,sync(graph['n_p'],op='sum'))
```

Listing 1: Using P^3's P-TAGS API to implement GraphSAGE.

followed by at most one non-element-wise NN operation (e.g. convolution) on vertex features and the aggregated neighborhood representation.

- `sync` accumulates partial representation (generated by the `transform` API) over the network using the user-provided arithmetic operation.
- `apply` is a user-provided composite function defined on each vertex to generate representation by applying zero or more element-wise and non-element-wise NN operations on vertex features and input representation.

Listing 1 outlines how GraphSAGE [24] can be implemented in P^3. Using our API, the developer composes `forward` function—function which generates output tensors from input tensors. Generated computational graph (see §3.2.1) and representation computed in the previous layer (or input vertex features partitioned along feature dimension if the first layer is being trained) are inputs to the `forward` function. For every layer in the GNN model, each vertex first aggregates the representation of its immediate neighborhood by applying element-wise mean (see `gather_udf`) over the incoming source vertex representation (see `scatter_udf`). Next, vertex's current representation and aggregated neighborhood representation are fed through a fully connected layer, element-wise summed (see `transform`) and passed through the non-linear function ReLU (see `apply`), which generates the representation used by the next layer. If this is the last layer, the generated representation is used as the vertex embedding for downstream tasks.

While training the first layer, the input representation is partitioned along the contracting (feature) dimension and evenly spread across machines, which results in the output representation generated by non-element-wise operators requiring synchronization. Notably, element-wise operations can still be applied without requiring synchronization. Since `transform` feeds the partitioned input representation through a fully connected layer, a non-element wise operator, its output representation must be synchronized before applying other downstream operators. `sync` accumulates partial representation over the network and produces the output representation which can be consumed by `apply`. Input representation in all layers except the first are partitioned along the batch dimension, and therefore the corresponding output representations do not require synchronization; thus `sync` is a no-op for all layers except the first.

4 Implementation

P^3 is implemented on Deep Graph Library (DGL) [1], a popular open-source framework for training GNN models. P^3 uses DGL as a graph propagation engine for sampling, neighborhood aggregation using message passing primitives and other graph related operations, and PyTorch as the neural network execution runtime. We extended DGL in multiple ways to support P^3's pipelined push-pull based distributed GNN training. First, we replaced the dependent graph partitioning strategy—features co-located with vertices and edges—in DGL with a strategy that supports partitioning graph structure and features independently. We reuse DGL's k-hop graph sampling service: for each minibatch a sampling request is issued via an Remote Procedure Call (RPC) to local and remote samplers. These samplers access locally stored graph partitions and return sampled graph—topology and features—to the trainer. Unlike DGL, sampling service in P^3 only returns the sampled graph topology and does not require input features to be transferred. Second, trainers in P^3 execute the GNN model using pipelined data and model parallelism. Each minibatch is assigned a unique identifier, and placed in a work queue. The trainer process picks minibatch samples and its associated data from the front of the queue,minibatch and applies neural network operations. P^3 schedules 3 concurrent minibatches using 2 forward, 2 backward static scheduling strategy (§3.3) to overlap communication with computation. Before the training mode for a minibatch can be switched from model to data parallelism, partial activations must be synchronized. To do so, we extended DGL's `KVStore` to store partial activations computed by trainers. `KVStore` uses RPC calls to orchestrate movement of partial activation across machines, and once synchronized, copies accumulated activation to device memory and places a pointer to the associated buffer in the work queue, shared with the trainer process. PyTorch's `DistributedDataParallel` module is used to synchronize weights before being used for weight update.

5 Evaluation

We evaluate P^3 on several real-world graphs and compare it to DGL and ROC, two state-of-the-art GNN frameworks that support distributed training. Overall, our results show that:

- P^3 is able to improve performance compared to DGL by up to 7× and ROC by up to 2.2×; and its benefits increase with graph size.

Graph	Nodes	Edges	Features
OGB-Product [29]	2.4 million	123.7 million	100
OGB-Paper [29]	111 million	1.6 billion	128
UK-2006-05 [10, 11]	77.7 million	2.9 billion	256
UK-Union [10, 11]	133.6 million	5.5 billion	256
Facebook [19]	30.7 million	10 billion	256

Table 3: Graph datasets used in evaluating P^3. Features column shows the number of features *per* node.

- We find that P^3 can achieve graceful scaling with number of machines and that it matches the published accuracy results for known training tasks.
- Our caching and pipelining techniques improve performance by up to 1.7×, with benefits increasing with more caching opportunities.

Experimental Setup: All of our experiments were conducted on a GPU cluster with 4 nodes, each of which has one 12-core Intel Xeon E5-2690v4 CPU, 441 GB of RAM, and four NVIDIA Tesla P100 16 GB GPUs. GPUs on the same node are connected via a shared PCIe interconnect, and nodes are connected via a 10 Gbps Ethernet interface. All servers run 64-bit Ubuntu 16.04 with CUDA library v10.2, DGL v0.5, and PyTorch v1.6.0.

Datasets & Comparison: We list the five graphs we use in our experiments in table 3. The first two are the largest graphs from the OGB repository [29]—OGB-Products [29], an Amazon product co-purchasing network, and OGB-Papers [29], a citation network between papers indexed by Microsoft Academic Graph [57]—where we can ensure correctness and validate the accuracy of P^3 on various tasks compared to the best reported results [4]. The latter three—UK-2006-05 [10, 11], a snapshot of .uk domains, UK-Union [10, 11], a 12-month time-aware graph of the same and Facebook [19], a synthetic graph which simulates the social network—are used to evaluate the scalability of P^3. We selected these due to the lack of open-source datasets of such magnitude specifically for GNN tasks. The two OGB graphs contain features. For the remaining three, we generate random features ensuring that the ratio of labeled nodes remain consistent with what we observed in the OGB datasets. Together, these datasets represent some of the largest open-source graphs used in the evaluation in recent GNN research[5]. We present comparisons against DGL [1, 61] and ROC [36], two of the best performing open-source GNN frameworks that support distributed training—to the best of our knowledge—at the time of our evaluation. However, due to the limitations imposed by ROC at the time of writing, specifically its support for only full-batch training and the availability of GCN implementation only, we compare against ROC only when it is feasible to do so and use DGL for the rest of the experiments. While DGL uses the METIS partitioner as the default, we change it to use hash partitioning in all the evaluations unless specified. This is due to two reasons. First, hash is the only partitioner that can handle all the five graphs in our datasets without running out of memory. Second, METIS incurs significant computational overheads that often exceed the total training time (see §2).

Models & Metrics: We use four different GNN models: S-GCN [63], GCN [17, 41], GraphSAGE [24] and GAT [59], in the increasing order of model complexity. These models represent the state-of-the-art architectures that can support all GNN tasks (§2). Unless mentioned otherwise, we use a standard 2-layer GNN model for all tasks. We enable sampling (unless stated) for all GNN architectures because it represents the best case for our comparison system and one of standard approaches to scaling. The sampling approach we adopted, based on recent literature [24], is a (25, 10) neighborhood sampling where we pick a maximum of 25 neighbors for the first hop of a node, and then a maximum of 10 neighbors for each of those 25. Both GraphSAGE and GCN use a hidden-size of 32. For the GAT model, we use 8 attention heads. Minibatch size is set to 1000 in all our experiments. We use a mix of node classification and link prediction tasks where appropriate for the input. Graph classification tasks are usually done on a set of small graphs, hence we do not include this task. We report the average *epoch time*, which is the time taken to perform one pass over the entire graph, unless otherwise stated. We note that for training tasks to achieve reasonable accuracy, several 100s or even 1000s of epochs are needed. In the experiment evaluating the accuracy attained by the model, we report the total time it takes to achieve the best reported accuracy (where available). For experiments that evaluate the impact of varying configurations (e.g., features), we pick a middle of the pack dataset in terms of size (OGB-Paper) and GNN in terms of complexity (GraphSAGE).

5.1 Overall Performance

We first present the overall results. Here, we compare DGL and P^3 in terms of per epoch time. For P^3, we disable caching (§3.4) so that it uses the *same amount of memory* as DGL for a fair comparison. Note that enabling caching only benefits P^3, and we show the benefits of caching later in this section. We train all the models on all the graphs, and report the mean time per epoch. The results are shown in table 4.

We see that P^3 outperforms DGL across the board, and the speedups range from 2.08× to 5.43×. The benefits increase as the input graph size increases. To drill down on why P^3 achieve such superior performance, we break down the epoch time into its constituents: embedding computation graph creation time (indicated as DAG), data copy time and the computation time which is the sum of the forward pass time, backward pass time and update time (§2). Clearly, P^3's independent partitioning strategy and the hybrid parallelism significantly reduces the time it takes to create the computation graph, which dominates the epoch time. We see a slight increase in data copy and compute times for P^3 due to the

[5] Larger industry datasets have been reported (e.g., [65, 68]) but they are unavailable to the public.

Graph	Model	DGL				P^3				Speedup
		Epoch	DAG	Copy	Compute	Epoch	DAG	Copy	Compute	
OGB-Product	SGCN	4.535	4.051	0.233	0.251	1.019	0.256	0.364	0.399	4.45
	GCN	4.578	3.997	0.253	0.328	1.111	0.248	0.372	0.491	4.12
	GraphSage	4.727	4.056	0.258	0.413	1.233	0.245	0.361	0.627	3.83
	GAT	5.067	4.164	0.271	0.632	1.912	0.248	0.379	1.285	2.65
OGB-Paper	SGCN	9.059	7.862	0.436	0.761	2.230	0.447	0.605	1.178	4.06
	GCN	9.575	8.117	0.461	0.997	2.606	0.457	0.619	1.530	3.67
	GraphSage	9.830	8.044	0.441	1.345	3.107	0.451	0.597	2.059	3.16
	GAT	10.662	8.094	0.462	2.106	5.138	0.462	0.652	4.024	2.08
UK-2006-05	SGCN	6.435	5.682	0.279	0.474	1.481	0.259	0.416	0.806	4.34
	GCN	7.023	6.146	0.282	0.595	1.509	0.252	0.408	0.849	4.65
	GraphSage	7.085	6.005	0.272	0.808	1.880	0.259	0.395	1.226	3.77
	GAT	8.084	6.378	0.330	1.376	3.379	0.234	0.472	2.673	2.39
UK-Union	SGCN	11.472	10.168	0.401	0.903	2.379	0.353	0.597	1.429	4.82
	GCN	12.523	10.815	0.444	1.264	2.864	0.343	0.624	1.897	4.37
	GraphSage	12.481	10.452	0.435	1.594	3.395	0.368	0.619	2.408	3.68
	GAT	13.597	10.693	0.480	2.424	5.752	0.371	0.652	4.729	2.36
Facebook	SGCN	22.264	19.765	0.627	1.872	4.102	0.509	0.907	2.686	5.43
	GCN	24.356	20.673	0.760	2.923	5.624	0.494	1.010	4.120	4.33
	GraphSage	23.936	19.756	0.755	3.425	6.298	0.554	1.027	4.717	3.80
	GAT	24.872	19.472	0.758	4.642	8.439	0.623	0.953	6.863	2.95

Table 4: P^3 is able to gain up to $5.4\times$ improvement in epoch time over DGL. The gains increase with graph size. The table also provides a split up of epoch time into its constituents: computation graph creation (DAG), data copy, and compute. The compute time is the sum of the forward pass, backward pass and update.

need for *pushing* the graph structure, and the overheads associated with additional CUDA calls necessary to push the activations (§3). We remind the reader that caching/replication for P^3 is disabled for this experiment, and enabling it would reduce the data copy time. However, P^3's aggressive pipelining is able to keep the additional overheads in forward pass to a minimum. We also notice that as the model complexity increases, the dominance of computation graph creation phase reduces in the overall epoch time as the forward and backward passes become more intensive.

5.2 Impact of Sampling

In the last experiment, we enabled aggressive sampling, which is a common strategy used by existing GNNs to achieve scalability and load balancing. However, sampling affects the accuracy of the task, and the number of epochs it is necessary to achieve the best accuracy. Further, some GNN architectures may not support sampling, or require more samples (compared to the (25, 10) setting we used). To evaluate how P^3 performs when the underlying task cannot support sampling, we repeat the experiment by disabling sampling. Everything else remains the same. Figure 7 shows the result.

Without sampling, we note that the largest graphs (UK-Union and Facebook) cannot be trained in our cluster. This is because the computation graphs exhaust the memory in the case of DGL and the only way to solve it is to enable sampling. Additionally, for the more complex model (GAT), DGL struggles to train on all datasets. Thus, we do not report

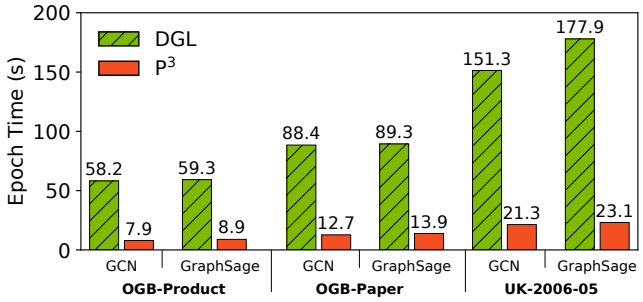
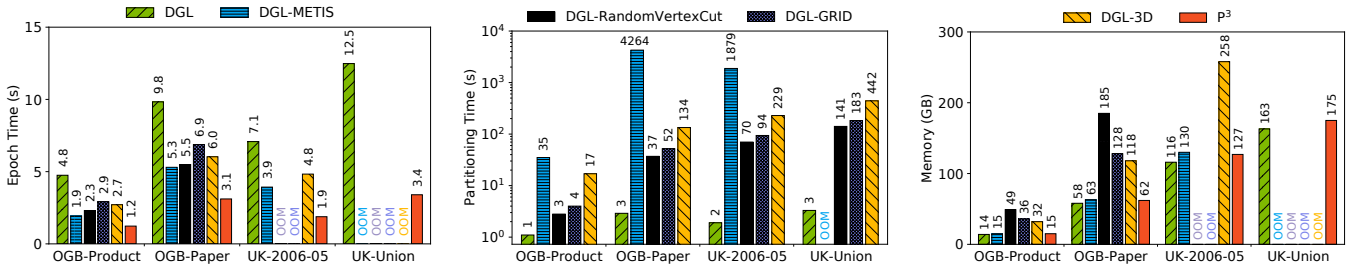


Figure 7: Without sampling, DGL struggles to train complex models and larger graphs. P^3's benefits increase up to 7.69 $\times$.

the results on these two large graphs and for GAT. Even otherwise, we note that P^3's benefits increase compared to the sampled case, with speed ups ranging from $6.45\times$ to $7.69\times$. This clearly indicates the benefits of pulling only the graph structure across the network.

5.3 Impact of Partitioning Strategy

Here, we investigate how different partitioning strategies affect the training time. DGL only supports edge-cut partitioning (using METIS [38]) by default, so we implemented four different partitioning schemes: hash, which is the same partitioner used by P^3, RandomVertexCut [22, 23] and GRID [12, 22] which are vertex-cut partitioners, and 3D, which is the 3-d partitioner proposed in [69]. We train one model, GraphSAGE in DGL with different partitioning strate-



(a) P^3 is able to gain up to 3.8× speed up in terms of epoch time compared to DGL.

(b) Schemes except random hash incur significant overheads in partitioning time (in log scale).

(c) Vertex-cut schemes incur significant memory overheads due to the need to replicate data.

Figure 8: P^3's random hash partitioning outperforms all schemes, including the best strategy in DGL (METIS).

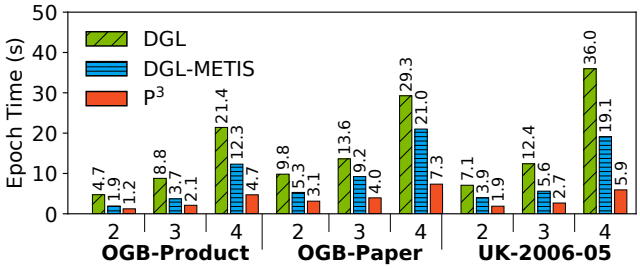


Figure 9: P^3's benefits increase with increase in layers in the GNN. With more layers, partitioning schemes do not help.

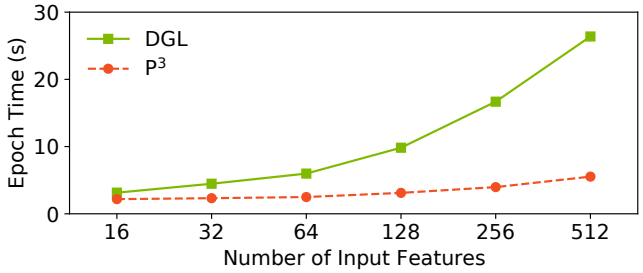


Figure 10: P^3's benefits increase as feature sizes increase, depicting the advantage of not moving features over the network.

gies, and compare against P^3 with its random hash partitioner. We report the average epoch time in fig. 8a.

We notice that P^3's random hash partitioning outperforms all schemes, even the best strategy in DGL (METIS), and the speedups for P^3 ranges from 1.7× (against METIS) to 3.85× (against random hash). The RandomVertexCut, GRID and 3D partitioners run out of memory for larger graphs. The only partitioning scheme that works for the Facebook graph is the random hash partitioner, so we omit it in this experiment. It may be tempting to think that an intelligent partitioner (other than hash partitioner) may benefit DGL. However, this is not true due to two reasons. First, partitioners incur preprocessing time as shown in fig. 8b. We see that METIS incurs the most time, and the overhead is often more than the total training time. It also cannot support large graphs. Other strategies may seem reasonable, but fig. 8c proves otherwise. This figure shows the memory used by various partitioning strategies. It can be seen that vertex cut schemes (RandomVertexCut, GRID, 3D) need to replicate data, and hence incur significant memory overhead. In contrast, not only does P^3's independent partitioning strategy outperform the best performing strategy in DGL (METIS) in terms of memory and epoch time, but it also incurs almost no preprocessing cost.

5.4 Impact of Layers

In this experiment, we evaluate the effect of the number of layers in the GNN. To do so, we pick GraphSAGE and create three different variants of the model, each having different number of layers, from 2 to 4. We then train the model using DGL and P^3. Sampling is enabled in this experiment, as DGL is unable to train deeper models (more layers) even on small graphs without it. The results are shown in fig. 9. We see that P^3's benefits increase with increase in the number of layers, outperforming DGL by up to 6.07×. This is because as the network becomes deeper, the computation graph also grows larger. Further, we see that as the network becomes deeper, the benefits of intelligent partitioning strategies (METIS) start to diminish compared to random hash partitioning. This is due to existing partitioning schemes being optimized for the first hop neighborhood. P^3 is not impacted by either due to its independent partitioning of graph and features and the hybrid parallelism in executing the GNN.

5.5 Impact of Features

To evaluate the impact of feature size on training performance, we vary the number of node features for OGB-Paper dataset from 16 to 512. Since the dataset originally had 128 features, we either prune or duplicate them to obtain the desired number of features. We use GraphSAGE model with sampling for training and report the average epoch time in fig. 10.

We clearly see the benefits of P^3's hybrid parallelism based execution. DGL's performance degrades with the increase in the number of features. This is expected, because to create the computation graph, DGL needs to pull the features, and with more features it incurs more network traffic. In contrast, since P^3 only needs to use network to get the activations, its performance incurs minimal degradation—the epoch time

Graph	Partitions Cached		Memory(GB)	Epoch(s)	Speedup
	Graph	Features			
OGB-Product	1	1	15	1.233	-
	4	4	68	0.724	1.703
OGN-Paper	1	1	62	3.107	-
	4	4	252	1.896	1.639
Facebook	1	1	158	6.298	-
	1	4	362	4.646	1.356
UK-2006-05	1	1	127	1.880	-
	2	2	262	1.524	1.233
UK-Union	1	1	175	3.395	-
	2	1	345	2.748	1.235

Table 5: By caching partitions of graph structure and features independently, P^3 is able to achieve up to 1.7× more performance.

only doubles when the number of features increase by a factor of 32. Here, P^3 outperforms DGL by 4.77×.

5.6 Microbenchmarks

Impact of Caching: In this experiment, we evaluate the benefits of P^3's caching (§3.4). Like in table 4, we use GraphSAGE for training on four graph datasets, but cache the partitions of the graph and features on multiple machines as memory permits. It is interesting to note that for some graphs, it is possible to replicate the structure on multiple machines (e.g., UK-Union) but not features and vice-versa (e.g., Facebook). This shows that independently partitioning the structure and features makes it possible to do caching which was otherwise not possible (i.e., DGL cannot leverage our caching mechanism). Here, P^3 is able to achieve up to 1.7× better performance, and the improvement increases with more caching opportunities. Moreover, caching extends training speedup of P^3 over DGL from 3.6× (in table 4) to 5.23× (here).

Impact of Pipelining: Here we evaluate the benefits of pipelining in P^3 (§3.3). To do so, we use P^3 to train GraphSAGE on four different datasets twice; first with pipelining enabled and then with it disabled. Figure 11 shows that pipelining effectively overlaps most of the communication with computation, and that P^3 is able to extract 30-50% more gains.

GPU Utilization: Figure 12 depicts the peak GPU utilization while training GraphSAGE model on OGB-Product dataset during a five second window for DGL and P^3. Here, the GPU utilization is measured every 50 milliseconds using the `nvidia-smi` [3] utility. We observe that the peak GPU utilization for both DGL and P^3 are similar (≈ 28%). This is due to the nature of GNN models, they perform sparse computations that fail to leverage peak hardware efficiency across all cores[6]. However, we see that during the duration of this experiment, DGL is able to keep the GPU *busy*—keep at least one of the many GPU cores active—for ≈ 20% of the time only. For the remaining ≈ 80%, GPU resources are blocked on the network and thus their utilization drops to zero. On the other hand, P^3 is able to keep the GPU *busy* for ≈ 85% of the time. As a result, it is able to complete 4 epochs of training in the five second duration, compared to 1 in the case of DGL.

[6]Improving peak utilization of accelerators such as GPUs by leveraging the sparsity of the workloads is outside the scope of this work.

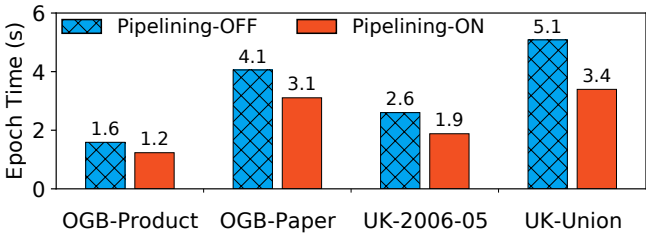


Figure 11: Pipelining boosts P^3's performance by up to 50%.

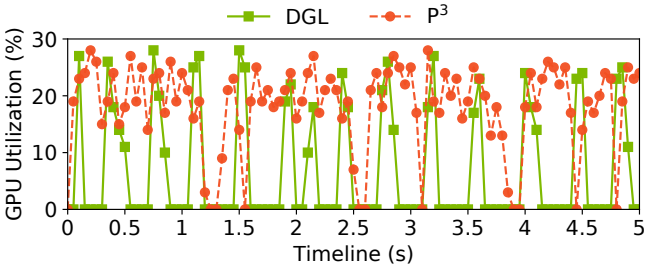


Figure 12: P^3 is able to keep the GPU busy for significantly more time (≈ 85%) compared to DGL (≈ 20%).

5.7 P^3's Scaling Characteristics

Here we evaluate the strong scaling properties of P^3. We again choose the OGB-Paper dataset and train GraphSAGE model on it. To understand the scaling properties, we vary the number of machines, there by varying the number of GPUs used by P^3 and DGL. We report the average throughput (the number of samples processed per second) in fig. 13.

P^3 exhibits near linear scaling characteristics; its throughput doubles when the number of machines (and hence the number of GPUs) are doubled. In contrast, DGL's throughput remains nearly the same as the number of machines increase. This is mainly because GPU resources in DGL are constrained by data movement over network, while P^3 is able to effectively eliminate this overhead using its proposed techniques. As the number of machines continue to grow, we expect P^3 to exhibit less optimal scaling. In P^3, each machine needs to pull activations from all other machines, and this grows linearly with the number of machines resulting in increased data movement that may adversely affect performance. This is a fundamental problem in model parallelism, and hence existing mitigation techniques are directly applicable to P^3.

5.8 Accuracy

Here, we evaluate the correctness of our approach in P^3. To do so, we train GraphSAGE model with sampling, but this time on the OBG-product graph (the smallest graph in our datasets). The best accuracy reported on this graph is approximately 78.2% using about 50 epochs [4]. Due to the lack of published accuracy results for larger graphs, we were unable to repeat this experiment for large graphs in our dataset. We run both DGL and P^3 on this dataset until we obtain the same accuracy

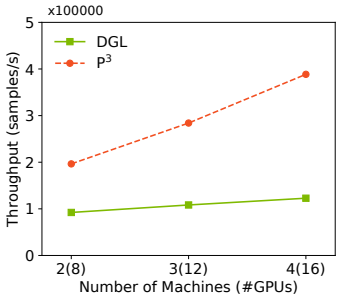


Figure 13: P^3 exhibits graceful and near linear scaling.

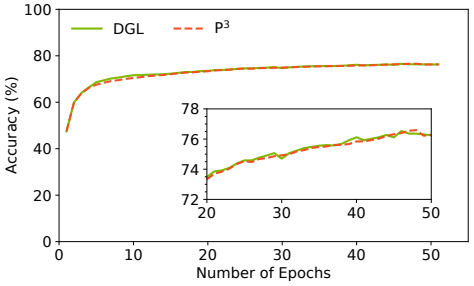


Figure 14: P^3 achieves the same accuracy as DGL, but much faster.

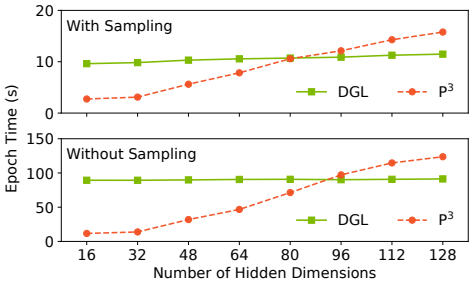


Figure 15: As the number of hidden dimensions increases, benefits of P^3 decreases.

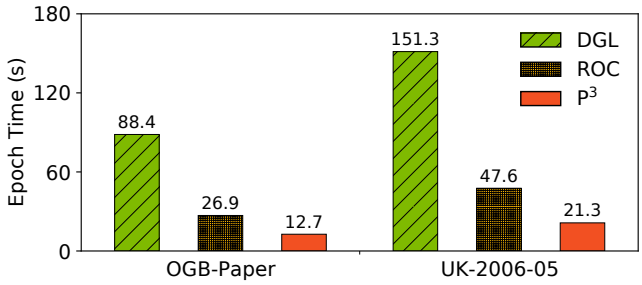


Figure 16: P^3 is able to outperform ROC by up to 2.2×, and its benefits increase with input graph size.

as reported. We show the results in fig. 14. We notice that stock DGL and P^3 both achieve the same accuracy iteration by iteration, and that they both achieve approximately 76.2% accuracy at the end of 51 iterations. P^3 is able to complete this training in 61.65 seconds, while DGL takes 236.37 seconds when using random hash partitioning of the input, and 96.3 seconds when using METIS partitioner. However, METIS takes 35.09 seconds to partition the graph, making the total training time 126.39 seconds. This experiment shows that not only is P^3 able to replicate the same accuracy as DGL thus ensuring its correctness, it is able to complete the training much faster than DGL even for the smallest of the graphs.

5.9 Comparison with ROC

Next we present comparison against ROC. Since ROC does not support sampling, we turn off sampling on all systems. At the time of evaluation, ROC only supported full batch training, and only had implementation for GCN available, so we use that as defaults for this experiment. We run 50 epochs of a 2-layer GCN on OGB-Paper and UK-2006-05 graphs. ROC uses an online partitioner that relies on moving parts of the graphs *during* the execution of the GNN. Due to this, we skip the first few epochs to allow ROC to complete its data movement and measure the average epoch time after that. The result of this experiment is shown in fig. 16.

ROC is able to process both the input graphs significantly faster than DGL due to its highly optimized graph engine. However, P^3 is able to outperform ROC, completing epochs up to 2.2× faster. We also notice that P^3's benefits increase with the size of the input graph. This is due to the fundamental differences in P^3 and ROC's design. While ROC's online partitioner is able to obtain superior partitions based on the access patterns, it still relies on moving features while training the GNN model. As the graph size increases, this results in more features being transferred across the network. In contrast, P^3's design tries to spread the computation of the bottle-necked layer across the cluster and avoids feature movement entirely. Moreover, as the number of layers increase, ROC (and DGL) would need to move exponentially more features, thereby resulting in increased network overhead.

While perusing this result, we wish to remind the reader about few caveats. Our evaluation uses 10 Gbps Ethernet interconnect which favours techniques resulting in lesser data movement. Hence, some of the observed network overheads due to feature movement for ROC (and DGL) can be minimized by using faster interconnects such as InfiniBand. Further, unlike ROC, P^3 and DGL require training data—the graph topology, features, model parameters and activations—to fit in device memory, and failure to do so results in out-of-memory error during training. On the other hand, ROC only requires training data to fit in DRAM, and leverages a cost-based memory manager to selectively move tensors between device memory and DRAM, which may affect performance.

5.10 P^3 Shortcomings

Finally, we present cases where P^3 does not provide benefits. Recall that the fundamental assumption made by P^3 is that the hidden dimensions in GNNs are typically smaller which results in the activations being significantly smaller than features. As this assumption is violated, P^3 starts losing its benefits, and may even incur performance penalties.

To illustrate this, we evaluate the impact of hidden dimensions in this experiment. We train GraphSAGE on the OGB-Product dataset, and fix the number of features to 100. For varying number of hidden dimensions, we record the average epoch time for DGL and P^3 with and without sampling enabled. Figure 15 shows the result. As we expect, the benefits of P^3 decreases as we increase the number of hidden

dimensions (thereby increasing the size of the activations), and P^3 becomes strictly worser than DGL once the hidden dimension size reach close to the feature size. We note that P^3 also incurs additional overhead due to model parallelism, due to which the exact point of transition varies depending on the characteristics of the graph. Dynamically determining whether P^3 would provide benefits in a given scenario and switching appropriately is part of our planned future work.

6 Related Work

Graph Processing Systems Several large-scale graph processing systems that provide an iterative message passing abstraction have been proposed in literature for efficiently leveraging CPUs [12, 21–23, 31, 32, 48, 50] and GPUs [39, 62]. These systems have been shown to be capable of scaling to huge graphs, in order of trillion edges [15]. However, these are focused mainly on *graph analysis and mining*, and lack support for functionalities that are crucial for GNN training, such as auto differentiation and dataflow programming.

Deep Learning Frameworks like PyTorch [5], TensorFlow [6], and MXNet [2] commonly use **Data Parallelism** [44] and **Model Parallelism** [14, 16] to speedup parallel and distributed DNN training. To scale even further, some recent works have proposed combining data and/or model parallelism with pipelining, operator-level partitioning, and activation compression [30, 42, 49, 54]. GPipe [30] and PipeDream [49] are aimed at alleviating low GPU-utilization problem of model parallelism. Both permit partitioning model across workers, allowing all workers to concurrently process different inputs, ensuring better resource utilization. GPipe maintains one weight version, but requires periodic pipeline flushes to update weight consistently, thus limiting overall throughput. PipeDream keeps multiple weight versions to ensure consistency, thereby avoiding periodic flushes at the cost of additional memory overhead. Prior works [37, 60] have even shown how to automatically find fast parallelization strategy for a setting using guided randomized search.

GNN Frameworks Driven by emerging popularity in training GNN models, several specialized frameworks [1, 20, 36, 45, 47, 65, 68] and accelerators [40] have been proposed. They can be categorized in two broad classes: systems [36, 65, 68] which extend existing graph processing systems with NN operations, and systems [1, 20, 45, 47] which extend existing tensor-based deep learning frameworks to support graph propagation operations. Both use graph partitioning as a means of scaling GNN training across multiple CPUs and/or GPUs either in a single machine or over multiple machines. Some frameworks, like AliGraph [65] and AGL [68], only support training using CPUs, while others [1, 20, 36, 45, 47] support performing training on GPUs and use CPU memory for holding graph partitions and exchanging data across GPUs.

PyTorch-Geometric [20] and DGL [1] wrap existing deep learning frameworks with a message passing interface. They focus on designing a graph oriented interface for improving GNN programmability by borrowing optimization principles for traditional graph processing systems and DNN frameworks. However, as we show, they fail to effectively leverage the unique context of GNNs workload and thereby yield poor performance and resource underutilization.

ROC [36] is a recent distributed multi-gpu GNN training system that shares the same goal as P^3, but proposes a fundamentally different approach. It explores using a linear regression model as a sophisticated online partitioner, which is jointly-learned with GNN training workload. Unlike P^3, despite the sophisticated partitioner, ROC must still move graph structure and features over network, which as we show results in high overheads.

PaGraph [45] and NeuGraph [47] are single machine multi-gpu frameworks for training GNNs. PaGraph reports data copy to be a major bottleneck and focuses on reducing data movement between CPU and GPU by caching features of most frequently visited vertices. On the other hand, NeuGraph uses partitioning and a stream scheduler to better overlap data copy and computation. However, in distributed multi-gpu setting, we observe that network communication is a major bottleneck and accounts for a large fraction, up to 80%, of training time while data copy time only accounts for 5%. We note that the proposed techniques in PaGraph and NeuGraph are orthogonal to our work and can only benefit P^3, if applied.

Besides above mentioned system-side optimizations to alleviate scalability bottlenecks, node-wise [24], layer-wise [72], and subgraph-based [13] **sampling techniques** have been proposed. These are orthogonal to and compatible with P^3.

7 Conclusion

In this paper, we looked at the problem of scalability issues in distributed GNN training and their ability to handle large input graphs. We found that network communication accounts for a major fraction of training time and that GPUs are severely underutilized due to this reason. We presented P^3, a system for distributed GNN training that overcomes the scalability challenges by adopting a radically new approach. P^3 practically eliminates the need for any intelligent partitioning of the graph, and proposes *independently* partitioning the input graph and features. It then completely avoids communicating (typically huge) features over the network by adopting a novel *pipelined push-pull* execution strategy that combines intra-layer model parallelism and data parallelism and further reduces overheads using a simple caching mechanism. P^3 exposes its optimizations in a simple API for the end user. In our evaluation, P^3 significantly outperforms existing state-of-the-art GNN frameworks, by up to 7×.

Acknowledgements

We thank our shepherd, Chuanxiong Guo, all the anonymous OSDI reviewers and Ramachandran Ramjee for the invaluable feedback that improved this work. We also thank the contributors and maintainers of PyTorch and DGL frameworks.

References

[1] Deep Graph Library. https://www.dgl.ai/.

[2] MXNet. https://mxnet.apache.org/.

[3] NVIDIA System Management Interface. https://developer.nvidia.com/nvidia-system-management-interface.

[4] Open Graph Benchmark Leaderboards. https://ogb.stanford.edu/docs/leader_nodeprop/.

[5] PyTorch. https://pytorch.org/.

[6] TensorFlow. https://www.tensorflow.org/.

[7] Turing-NLG: A 17-billion-parameter language model by Microsoft. https://www.microsoft.com/en-us/research/blog/turing-nlg-a-17-billion-parameter-language-model-by-microsoft/.

[8] Davide Bacciu, Federico Errica, and Alessio Micheli. Contextual graph Markov model: A deep and generative approach to graph processing. In Jennifer Dy and Andreas Krause, editors, *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 294–303, Stockholmsmässan, Stockholm Sweden, 10–15 Jul 2018. PMLR.

[9] Y. Bengio, A. Courville, and P. Vincent. Representation learning: A review and new perspectives. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 35(8):1798–1828, 2013.

[10] Paolo Boldi, Marco Rosa, Massimo Santini, and Sebastiano Vigna. Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks. In Sadagopan Srinivasan, Krithi Ramamritham, Arun Kumar, M. P. Ravindra, Elisa Bertino, and Ravi Kumar, editors, *Proceedings of the 20th international conference on World Wide Web*, pages 587–596. ACM Press, 2011.

[11] Paolo Boldi and Sebastiano Vigna. The WebGraph framework I: Compression techniques. In *Proc. of the Thirteenth International World Wide Web Conference (WWW 2004)*, pages 595–601, Manhattan, USA, 2004. ACM Press.

[12] Rong Chen, Jiaxin Shi, Yanzhe Chen, and Haibo Chen. Powerlyra: Differentiated graph computation and partitioning on skewed graphs. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, New York, NY, USA, 2015. Association for Computing Machinery.

[13] Wei-Lin Chiang, Xuanqing Liu, Si Si, Yang Li, Samy Bengio, and Cho-Jui Hsieh. Cluster-gcn: An efficient algorithm for training deep and large graph convolutional networks. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, KDD '19, pages 257–266, New York, NY, USA, 2019. Association for Computing Machinery.

[14] Trishul Chilimbi, Yutaka Suzue, Johnson Apacible, and Karthik Kalyanaraman. Project adam: Building an efficient and scalable deep learning training system. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 571–582, Broomfield, CO, October 2014. USENIX Association.

[15] Avery Ching, Sergey Edunov, Maja Kabiljo, Dionysios Logothetis, and Sambavi Muthukrishnan. One trillion edges: Graph processing at facebook-scale. *Proc. VLDB Endow.*, 8(12):1804–1815, August 2015.

[16] Jeffrey Dean, Greg S. Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Quoc V. Le, Mark Z. Mao, Marc'Aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, and Andrew Y. Ng. Large scale distributed deep networks. In *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*, NIPS'12, page 1223–1231, Red Hook, NY, USA, 2012. Curran Associates Inc.

[17] Michaël Defferrard, Xavier Bresson, and Pierre Vandergheynst. Convolutional neural networks on graphs with fast localized spectral filtering. In D. Lee, M. Sugiyama, U. Luxburg, I. Guyon, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 29, pages 3844–3852. Curran Associates, Inc., 2016.

[18] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4171–4186, Minneapolis, Minnesota, June 2019. Association for Computational Linguistics.

[19] S. Edunov, D. Logothetis, C. Wang, A. Ching, and M. Kabiljo. Generating synthetic social graphs with darwini. In *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*, pages 567–577, 2018.

[20] Matthias Fey and Jan E. Lenssen. Fast graph representation learning with PyTorch Geometric. In *ICLR Workshop on Representation Learning on Graphs and Manifolds*, 2019.

[21] Swapnil Gandhi and Yogesh Simmhan. An Interval-centric Model for Distributed Computing over Temporal Graphs. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*, pages 1129–1140, 2020.

[22] Joseph Gonzalez, Reynold Xin, Ankur Dave, Daniel Crankshaw, and Ion Franklin, Stoica. Graphx: Graph processing in a distributed dataflow framework. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, Broomfield, CO, October 2014. USENIX Association.

[23] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, page 17–30, USA, 2012. USENIX Association.

[24] Will Hamilton, Zhitao Ying, and Jure Leskovec. Inductive representation learning on large graphs. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30, pages 1024–1034. Curran Associates, Inc., 2017.

[25] William Hamilton. Graph Representation Learning Book. https://www.cs.mcgill.ca/~wlh/grl_book/.

[26] William L. Hamilton, Rex Ying, and Jure Leskovec. Representation Learning on Graphs: Methods and Applications. *IEEE Data Engineering Bulletin*, page arXiv:1709.05584, September 2017.

[27] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, 2016.

[28] G. Hinton, L. Deng, D. Yu, G. E. Dahl, A. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T. N. Sainath, and B. Kingsbury. Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *IEEE Signal Processing Magazine*, 29(6):82–97, 2012.

[29] Weihua Hu, Matthias Fey, Marinka Zitnik, Yuxiao Dong, Hongyu Ren, Bowen Liu, Michele Catasta, and Jure Leskovec. Open Graph Benchmark: Datasets for Machine Learning on Graphs. *arXiv e-prints*, page arXiv:2005.00687, May 2020.

[30] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, et al. Gpipe: Efficient training of giant neural networks using pipeline parallelism. In *Advances in neural information processing systems*, pages 103–112, 2019.

[31] Anand Padmanabha Iyer, Zaoxing Liu, Xin Jin, Shivaram Venkataraman, Vladimir Braverman, and Ion Stoica. ASAP: Fast, approximate graph pattern mining at scale. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 745–761, Carlsbad, CA, October 2018. USENIX Association.

[32] Anand Padmanabha Iyer, Qifan Pu, Kishan Patel, Joseph E. Gonzalez, and Ion Stoica. TEGRA: Efficient ad-hoc analytics on evolving graphs. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 337–355. USENIX Association, April 2021.

[33] Animesh Jain, Amar Phanishayee, Jason Mars, Lingjia Tang, and Gennady Pekhimenko. Gist: Efficient data encoding for deep neural network training. In *Proceedings of the 45th Annual International Symposium on Computer Architecture*, ISCA '18, page 776–789. IEEE Press, 2018.

[34] Paras Jain, Ajay Jain, Aniruddha Nrusimha, Amir Gholami, Pieter Abbeel, Joseph Gonzalez, Kurt Keutzer, and Ion Stoica. Checkmate: Breaking the memory wall with optimal tensor rematerialization. In I. Dhillon, D. Papailiopoulos, and V. Sze, editors, *Proceedings of Machine Learning and Systems*, volume 2, pages 497–511, 2020.

[35] Myeongjae Jeon, Shivaram Venkataraman, Amar Phanishayee, unjie Qian, Wencong Xiao, and Fan Yang. Analysis of large-scale multi-tenant gpu clusters for dnn training workloads. In *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '19, page 947–960, USA, 2019. USENIX Association.

[36] Zhihao Jia, Sina Lin, Mingyu Gao, Matei Zaharia, and Alex Aiken. Improving the accuracy, scalability, and performance of graph neural networks with roc. In I. Dhillon, D. Papailiopoulos, and V. Sze, editors, *Proceedings of Machine Learning and Systems*, volume 2, pages 187–198, 2020.

[37] Zhihao Jia, Matei Zaharia, and Alex Aiken. Beyond data and model parallelism for deep neural networks. In A. Talwalkar, V. Smith, and M. Zaharia, editors, *Proceedings of Machine Learning and Systems*, volume 1, pages 1–13, 2019.

[38] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput.*, 20(1):359–392, December 1998.

[39] Farzad Khorasani, Keval Vora, Rajiv Gupta, and Laxmi N. Bhuyan. Cusha: Vertex-centric graph processing on gpus. In *Proceedings of the 23rd International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '14, page 239–252, New York, NY, USA, 2014. Association for Computing Machinery.

[40] Kevin Kiningham, Christopher Re, and Philip Levis. GRIP: A Graph Neural Network Accelerator Architecture. *arXiv e-prints*, page arXiv:2007.13828, July 2020.

[41] Thomas N. Kipf and Max Welling. Semi-Supervised Classification with Graph Convolutional Networks. In *Proceedings of the 5th International Conference on Learning Representations*, ICLR '17, 2017.

[42] Alex Krizhevsky. One weird trick for parallelizing convolutional neural networks. *arXiv e-prints*, page arXiv:1404.5997, April 2014.

[43] Jurij Leskovec. *Dynamics of Large Networks*. PhD thesis, USA, 2008.

[44] Mu Li, David G. Andersen, Jun Woo Park, Alexander J. Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J. Shekita, and Bor-Yiing Su. Scaling distributed machine learning with the parameter server. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, page 583–598, USA, 2014. USENIX Association.

[45] Zhiqi Lin, Cheng Li, Youshan Miao, Yunxin Liu, and Yinlong Xu. Pagraph: Scaling gnn training on large graphs via computation-aware caching. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, SoCC '20, pages 401–415, New York, NY, USA, 2020. Association for Computing Machinery.

[46] Yu-Chen Lo, Stefano E. Rensi, Wen Torng, and Russ B. Altman. Machine learning in chemoinformatics and drug discovery. *Drug Discovery Today*, 23(8):1538 – 1546, 2018.

[47] Lingxiao Ma, Zhi Yang, Youshan Miao, Jilong Xue, Ming Wu, Lidong Zhou, and Yafei Dai. Neugraph: Parallel deep neural network computation on large graphs. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 443–458, Renton, WA, July 2019. USENIX Association.

[48] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: A system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, page 135–146, New York, NY, USA, 2010. Association for Computing Machinery.

[49] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R. Devanur, Gregory R. Ganger, Phillip B. Gibbons, and Matei Zaharia. Pipedream: Generalized pipeline parallelism for dnn training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, pages 1–15, New York, NY, USA, 2019. Association for Computing Machinery.

[50] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. A lightweight infrastructure for graph analytics. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, page 456–471, New York, NY, USA, 2013. Association for Computing Machinery.

[51] Aditya Pal, Chantat Eksombatchai, Yitong Zhou, Bo Zhao, Charles Rosenberg, and Jure Leskovec. Pinnersage: Multi-modal user embedding framework for recommendations at pinterest. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, KDD '20, page 2311–2320, New York, NY, USA, 2020. Association for Computing Machinery.

[52] Jay H. Park, Gyeongchan Yun, Chang M. Yi, Nguyen T. Nguyen, Seungmin Lee, Jaesik Choi, Sam H. Noh, and Young ri Choi. Hetpipe: Enabling large DNN training on (whimpy) heterogeneous GPU clusters through integration of pipelined model parallelism and data parallelism. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 307–321. USENIX Association, July 2020.

[53] Namyong Park, Andrey Kan, Xin Luna Dong, Tong Zhao, and Christos Faloutsos. Estimating node importance in knowledge graphs using graph neural networks. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, KDD '19, page 596–606, New York, NY, USA, 2019. Association for Computing Machinery.

[54] Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery &; Data Mining*, KDD '20, page 3505–3506, New York, NY, USA, 2020. Association for Computing Machinery.

[55] Siddhartha Sahu, Amine Mhedhbi, Semih Salihoglu, Jimmy Lin, and M. Tamer Özsu. The ubiquity of large graphs and surprising challenges of graph processing. *Proc. VLDB Endow.*, 11(4):420–431, December 2017.

[56] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro.

Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism. *arXiv e-prints*, page arXiv:1909.08053, September 2019.

[57] Arnab Sinha, Zhihong Shen, Yang Song, Hao Ma, Darrin Eide, Bo-June (Paul) Hsu, and Kuansan Wang. An overview of microsoft academic service (mas) and applications. In *Proceedings of the 24th International Conference on World Wide Web*, WWW '15 Companion, page 243–246, New York, NY, USA, 2015. Association for Computing Machinery.

[58] Jonathan M. Stokes, Kevin Yang, Kyle Swanson, Wengong Jin, Andres Cubillos-Ruiz, Nina M. Donghia, Craig R. MacNair, Shawn French, Lindsey A. Carfrae, Zohar Bloom-Ackermann, Victoria M. Tran, Anush Chiappino-Pepe, Ahmed H. Badran, Ian W. Andrews, Emma J. Chory, George M. Church, Eric D. Brown, Tommi S. Jaakkola, Regina Barzilay, and James J. Collins. A deep learning approach to antibiotic discovery. *Cell*, 180(4):688 – 702.e13, 2020.

[59] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. Graph attention networks. In *International Conference on Learning Representations*, 2018.

[60] Minjie Wang, Chien-chin Huang, and Jinyang Li. Supporting very large models using automatic dataflow graph partitioning. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19, New York, NY, USA, 2019. Association for Computing Machinery.

[61] Minjie Wang, Da Zheng, Zihao Ye, Quan Gan, Mufei Li, Xiang Song, Jinjing Zhou, Chao Ma, Lingfan Yu, Yu Gai, Tianjun Xiao, Tong He, George Karypis, Jinyang Li, and Zheng Zhang. Deep Graph Library: A Graph-Centric, Highly-Performant Package for Graph Neural Networks. *arXiv e-prints*, page arXiv:1909.01315, September 2019.

[62] Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D. Owens. Gunrock: A high-performance graph processing library on the gpu. *SIGPLAN Not.*, 51(8), February 2016.

[63] Felix Wu, Amauri Souza, Tianyi Zhang, Christopher Fifty, Tao Yu, and Kilian Weinberger. Simplifying graph convolutional networks, 2019.

[64] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, and P. S. Yu. A comprehensive survey on graph neural networks. *IEEE Transactions on Neural Networks and Learning Systems*, pages 1–21, 2020.

[65] Hongxia Yang. Aligraph: A comprehensive graph neural network platform. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, KDD '19, pages 3165–3166, New York, NY, USA, 2019. Association for Computing Machinery.

[66] Rex Ying, Ruining He, Kaifeng Chen, Pong Eksombatchai, William L. Hamilton, and Jure Leskovec. Graph convolutional neural networks for web-scale recommender systems. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, KDD '18, pages 974–983, New York, NY, USA, 2018. Association for Computing Machinery.

[67] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, NSDI'12, pages 2–2, Berkeley, CA, USA, 2012. USENIX Association.

[68] Dalong Zhang, Xin Huang, Ziqi Liu, Jun Zhou, Zhiyang Hu, Xianzheng Song, Zhibang Ge, Lin Wang, Zhiqiang Zhang, and Yuan Qi. AGL: A scalable system for industrial-purpose graph machine learning. *Proc. VLDB Endow.*, 13(12):3125–3137, August 2020.

[69] Mingxing Zhang, Yongwei Wu, Kang Chen, Xuehai Qian, Xue Li, and Weimin Zheng. Exploring the hidden dimension in graph processing. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 285–300, Savannah, GA, November 2016. USENIX Association.

[70] Muhan Zhang and Yixin Chen. Link prediction based on graph neural networks. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems 31*, pages 5165–5175. Curran Associates, Inc., 2018.

[71] Jie Zhou, Ganqu Cui, Zhengyan Zhang, Cheng Yang, Zhiyuan Liu, Lifeng Wang, Changcheng Li, and Maosong Sun. Graph neural networks: A review of methods and applications, 2019.

[72] Difan Zou, Ziniu Hu, Yewen Wang, Song Jiang, Yizhou Sun, and Quanquan Gu. Layer-dependent importance sampling for training deep and large graph convolutional networks. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 32, pages 11249–11259. Curran Associates, Inc., 2019.