GSplit: Scaling Graph Neural Network Training on Large Graphs via Split-Parallelism

Sandeep Polisetty[1], Juelin Liu[1], Kobi Falus[1], Yi Ren Fung[3], Seung-Hwan Lim[2], Hui Guan[1], and Marco Serafini[1]

[1]University of Massachusetts, Amherst
[2]Oak Ridge National Laboratory, Tennessee
[3]University of Illinois, Urbana-Champaign

Abstract

Large-scale graphs with billions of edges are ubiquitous in many industries, science, and engineering fields such as recommendation systems, social graph analysis, knowledge base, material science, and biology. Graph neural networks (GNN), an emerging class of machine learning models, are increasingly adopted to learn on these graphs due to their superior performance in various graph analytics tasks. Mini-batch training is commonly adopted to train on large graphs, and data parallelism is the standard approach to scale mini-batch training to multiple GPUs. In this paper, we argue that several fundamental performance bottlenecks of GNN training systems have to do with inherent limitations of the data parallel approach. We then propose split parallelism, a novel parallel mini-batch training paradigm. We implement split parallelism in a novel system called GSplit and show that it outperforms state-of-the-art systems such as DGL, Quiver, and PaGraph.

1 Introduction

Graph neural networks (GNN), an emerging class of machine learning models, are increasingly adopted to analyze graph-structured data due to their superior performance in various graph analytics tasks. An input graph for a GNN can have millions of vertices and billions of edges [14]. GNNs are broadly adopted in companies such as Pinterest [35] and Twitter [8] to improve user experiences and in engineering and science domains for computer vision [23], natural language understanding [12], quantum chemistry [11], material design [39], and drug discovery [10].

A common approach to train a GNN on large-scale graphs is to use *mini-batch* gradient descent. This approach partitions the training data into subsets called mini-batches. Each training *iteration* calculates updates to the model parameters, called gradients, based on a different mini-batch. In GNN training, we want to learn how to compute the features of the vertices in the training set, which are called *target vertices*, based on the input features of the vertices in their k-hop neighborhood. A mini-batch consists of a subset of target vertices and a *sample* of their k-hop neighborhood, which could be excessively large if sampling was not used (see Figure 1).

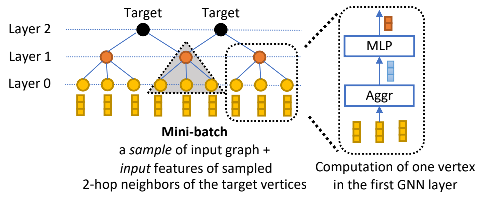
Mini-batch training is commonly used in production and research GNN training systems such as DGL [30], PyTorch Geometric [7], Quiver [4], AliGraph [38], PaGraph [18], and GNN Lab [34]. These systems use *data parallelism* to execute each training iteration across multiple GPUs (see Figure 2). At each iteration, data parallelism partitions the mini-batch into *micro-batches*, which consist of a partition of the target vertices in the mini-batch and their sampled k-hop neighbors, and assigns each micro-batch to one GPU. The entire GNN model is replicated at each GPU. Each GPU then loads the input features of all the vertices in its micro-batch and computes gradients locally and independently from other GPUs. At the end of the iteration, all GPUs aggregate and apply the gradients they computed.

Limitations of Existing Work. Unfortunately, data parallel training for GNNs is *inherently redundant* due to the *overlaps* between the k-hop neighborhoods of target vertices that are assigned to different micro-batches (gray area in Figure 1). This creates both data loading and computation overheads.

Loading training data to GPUs at each iteration is known to be a significant overhead in data-parallel training since each GPU must gather the input features of all the vertices in its micro-batch, which can be large [18]. This overhead is exacerbated by the overlaps across micro-batches, since several vertices are likely to appear in micro-batches assigned to multiple GPUs. The input features of these vertices are sent to multiple GPUs, which increases the communication cost.

An orthogonal problem caused by overlaps across micro-batches is *redundant computation*. GNN models organize the vertices of a micro-batch sample into layers. They compute the hidden features of the vertices in a layer by aggregating the features of their neighbors in the lower layer. If the same vertex appears in multiple micro-batches assigned to multiple GPUs, these will compute their hidden features redundantly, which can be a significant overhead.

Proposed Approach. In this paper, rather than patching



Target
Target
Layer 2
Layer 1
Layer 0
Mini-batch
a *sample* of input graph +
input features of sampled
2-hop neighbors of the target vertices
MLP
Aggr
Computation of one vertex
in the first GNN layer

Figure 1: Example of a mini-batch.

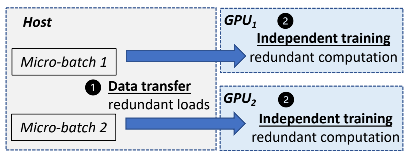


Figure 2: Data parallelism.

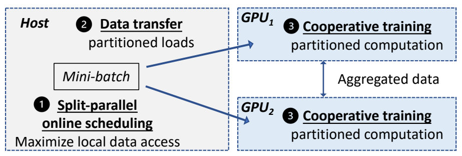


Figure 3: Split parallelism.

the data-parallel pipeline incrementally, we introduce a new paradigm for parallel mini-batch GNN training called *split parallelism* (see Figure 3). At each iteration, split parallelism splits the vertices of the mini-batch into multiple partitions. Unlike micro-batches, partitions do not have overlaps. Each partition is assigned to a GPU for computation and partitions are created in a way that maximizes local computation. During training, GPUs cooperate with each other to exchange lightweight intermediate results in each iteration.

Split parallelism solves the two fundamental problems of data parallelism. It eliminates redundant computation because each vertex is uniquely associated to one GPU that is responsible for computing its hidden features. It also eliminates redundant data loading because each vertex belongs to the split of exactly one GPU, which is the only one responsible for loading its input features.

Split parallelism builds on some ideas used in systems for full-graph training, which are designed to train the GNN on the entire graph at each iteration [16, 19, 20]. To scale to large graphs, these systems partition the input graph and exchange intermediate data across GPUs. Mini-batch training, however, presents two unique challenges and opportunities to make split parallelism efficient.

First, *mini-batches change at each iteration*, since they are randomly sampled. Split parallelism splits each mini-batch into non-overlapping partitions, each corresponding to one GPU, to eliminate redundancies. We call this novel scheduling problem *split-parallel online scheduling*. The scheduling algorithm is executed on the critical path of each iteration so it must be very fast. We show how to achieve this goal by combining offline partitioning and online scheduling. In full-graph training, online scheduling is not necessary since each iteration operates on the same graph using the same schedule, which is computed offline. Data parallelism also uses the same schedule at each iteration (see Figure 2).

The consequence of splitting a mini-batch into non-overlapping partitions is that split parallelism requires inter-GPU communications to complete one iteration of training, because some local partitions may have a few edges that require data from other GPUs. We implement *cooperative training* that invokes *split-aware* kernels and transparently supports GNN models written using the standard high-level SAGA operators offered by existing GNN training systems. Our implementation leverages fast GPU-GPU buses such as NVLink, when available, to speed up coordination. We present the first experimental evaluation of cooperative training in the context of mini-batch training. Our evaluation demonstrates that the inter-GPU communication required for cooperative training represents a much smaller cost than the redundancies in input data loading and computation inherent in data parallelism.

Second, *mini-batches are only subsets of the whole graph and some vertices are much more likely to be included in a mini-batch than others.* This offers caching opportunities to make split parallelism efficient. Existing work on mini-batch training has shown that *caching in the GPU memory* the input features of frequently-loaded vertices can reduce data transfers and significantly speed up training [4, 18, 34]. In full-graph training, caching is not used because the input features of *all* vertices of a potentially large graph must be loaded to the GPUs at every iteration.

Although caching is used in some data-parallel–based GNN training systems [4, 18, 34], split parallelism is much better suited than data parallelism to leverage in-GPU input feature caches. Data parallelism cannot take full advantage of caches because it moves input data (the micro-batches) to computation (the GPUs training independently). A GPU still needs to load all the input features in its micro-batch that are not cached locally. Split parallelism, instead, uses online scheduling to split mini-batches so that if a GPU caches the input features of a vertex, it is given the responsibility of processing those features locally. This follows the well-established principle of moving computation to data, not data to computation.

By moving computation to data, split parallelism enables multiple GPUs to cache *non-overlapping subsets* of input features and form a *distributed GPU cache*, whose size scales with the number of GPUs in the system. Input features only need to be transferred from the CPU memory to a GPU when there are *global* cache misses, that is, a feature is not cached by *any* GPU. Data parallelism can take advantage of a distributed GPU cache only in systems with fast GPU-GPU buses, such as NVLink. When such a bus is available and a GPU has a local cache miss, it is faster to load input features from another GPU memory than from the CPU memory, as done by systems like Quiver [4] and WholeGraph [33]. Without a fast GPU-GPU bus, a GPU running data-parallel training must load data through the PCIe bus regardless of whether the features are cached by another GPU or stored in the host memory, so a distributed GPU cache is not effective. Split parallelism improves data loading time independent of the system configuration because it does not require transferring cached features across GPUs.

Results. We implement the proposed solutions in GSplit, a novel split-parallel GNN training framework targeting *synchronous multi-GPU* training over large graphs on a single host. We compare GSplit against state-of-the-art single-host multi-GPU systems for mini-batch training: DGL [30], PaGraph [18], and Quiver [4]. GSplit consistently and significantly outperforms all these baselines, sometimes by more than one order of magnitude. Split parallelism reduces data loading time, one of the main bottlenecks in the end-to-end training pipeline, by a large margin. Its training time is competitive with data-parallel training because the communication cost of cooperative training is balanced by the gains of eliminating computational redundancy. Online scheduling is fast enough to not become a bottleneck, yet it produces partitions that are good enough to deliver consistent speedups over the baselines.

This paper makes the following key contributions:

- We introduce split parallelism, a novel paradigm for mini-batch GNN training that maximizes data access locality and avoids the redundant computation and data transfer overheads of data parallelism.
- We design a fast online scheduling algorithm to obtain per-GPU computation graphs from each mini-batch at each iteration.
- We implement cooperative training to support GNN models written using standard SAGA operators. We present the first experimental comparison of cooperative mini-batch training and data parallel training.
- We implement GSplit, an end-to-end multi-GPU GNN training system based on split parallelism that GSplit implements an efficient split-aware operator.

2 Background and Motivations

This section first provides a brief background on GNN Training and then discusses the limitations of existing approaches to motivate this work.

2.1 Graph Neural Network Training

Mini-batch training on GNN models. Stochastic Gradient Descent (SGD) trains on a mini-batch of the training data at each iteration. In GNNs, a mini-batch consists of a subset of vertices of the graph, which are called *target vertices*, and a *sample* of their k-hop neighborhood (see Figure 1). In data-parallel training, micro-batches are the partitions of a mini-batch that are trained in parallel, one per GPU, in one iteration. They are obtained by sampling the k-hop neighborhood of a partition of the target vertices in the mini-batch.

A GNN model is defined as a sequence of *GNN layers*.[1] In the forward propagation, each GNN layer $l > 0$ aggregates and transforms the features of the vertices in the layer $l-1$ of the sample and produces the features of the vertices in the layer l (see Figure 1). The last GNN layer computes the features of the target vertices, which are used to compute the loss. In the backward propagation, the layers are executed in an reversed order to compute gradients.

GNN layers. Each GNN layer l calculates the features of $V^{(l)}$, the vertices in layer l, using features of $V^{(l-1)}$, the vertices in layer $l-1$, and sometimes also features of $E^{(l)}$, the edges between vertices in layer $l-1$ and l. The implementation typically follows a message-passing framework that executes scatter function, message function, gather function, and update function sequentially.

The *scatter function* $\sigma^{(l)}$ prepares an *edge tensor* of edge-wise vectors to compute messages. The edge tensor combines, for each edge $e(u,v) \in E^{(l)}$, vectors for the source and destination vertices, typically the feature vectors $h_u^{(l-1)}$ and $h_v^{(l-1)}$, and optionally an edge feature vector $w_e^{(l-1)}$:

$$\sigma_e^{(l)} = [h_u^{(l-1)}, h_v^{(l-1)}, w_e^{(l-1)}], \quad e(u,v) \in E^{(l)}$$

This function does not perform any computation: it merely collects and combines sparse data from different vectors based on the structure of the graph.

The *message function* $\phi^{(l)}$ produces a message tensor M containing a message for each edge in $E^{(l)}$. It typically takes as input an edge tensor from the scatter operates and applies $\phi^{(l)}$ on each edge e to build a message:

$$m_e^{(l)} = \phi^{(l)}(\sigma_e^l), \quad e(u,v) \in E^{(l)}.$$

The ϕ function is defined by the user and can be, for example, a Multi-Layer Perceptron (MLP). In some simpler GNN models, a message function uses only the source vector of each edge, so it does not require building a full edge tensor using a scatter. In the example of Figure 1, ϕ outputs the features of the source vertex for each edge.

The *gather* function $\oplus$ takes a message tensor as input and aggregates all messages to the same destination vertex using a commutative and associative aggregation function such as sum or mean (see the `Aggr` block in Figure 1). The entry in the output tensor for a vertex $v \in V^{(l)}$ is:

$$m_v^{(l)} = \oplus^{(l)}_{e(u,v)\in E^{(l)}} m_e^{(l)}, \quad e(u,v) \in E^{(l)}.$$

Finally, the *update* function ψ computes a new hidden feature for a vertex v based on the aggregated message to v.

[1] In the following, the term "layer" will refer to GNN layers, not to neural network layers, unless otherwise stated.

This can also be an arbitrary neural network, similar to the message function ϕ (see the `MLP` block in Figure 1).

$$h_v^{(l)} = \psi^{(l)}(h_v^{(l-1)}, m_v^{(l)}), \quad v \in V^{(l)}.$$

We call a chain of Scatter, Apply-message, Gather, Apply-update operations a *SAGA*, following the terminology introduced by NeuGraph [19].

2.2 Limitations of Data Parallelism

In data parallelism, micro-batches often overlap, resulting in redundant data loading and computation. Suppose that the mini-batch of Figure 1 is the union of two micro-batches, one for each of the target vertices. The two micro-batches are associated with two different GPUs. The triangle in the Figure shows overlaps in the k-hop sample of different target vertices. The input features of the layer-0 vertices in the overlap need to be loaded by both GPUs. Similarly, the hidden features of the layer-1 vertex in the overlap need to be computed by both GPUs. We now discuss the overheads resulting from these redundancies in detail.

Issue 1: Redundant data loading. Data loading transfers training data (including the micro-batch graph structures and the involved input vertex features) to GPUs for computation at each iteration. Data-parallel training systems cache input vertex features on GPUs to mitigate the data loading bottleneck. Some work, such a PaGraph [18] or GNNLab [34] focus on configurations where GPUs are connected with each other through the same bus as the CPU (the PCIe bus). These systems keep independent per-GPU caches. To decide which vertices to cache on each GPU, they logically partition the training vertices across GPUs. They then construct a subgraph for each partition as the union of the k-hop neighborhoods of the training vertices. Finally, they cache at each GPU a subset of the vertices in each partition. Because the k-hop subgraphs have large overlaps, the caches of each GPU also overlap.

High-end multi-GPU systems have fast GPU-GPU buses such as NVLink [3], which are faster than CPU-GPU buses like PCIe. Quiver is a recent GNN training system that leverages these hardware features to partition cached input features across multiple GPUs [4]. Whenever a GPU needs to load the features of a vertex that is not cached locally, it loads it from the cache of another GPU if possible, and loads from the CPU memory otherwise. This strategy ensures that caches have no overlaps.

Both strategies make data loading cheaper, but they still suffer from significant data loading costs by redundantly loading the same features on multiple GPUs. Table 1 quantifies the data transfer volume and epoch time overhead in prior data-parallel systems. The experiments consider single-server systems with 4 GPUs connected with a PCIe bus for PaGraph and NVLink for Quiver (see Section 5 for the detailed experimental setup). The Table considers different cache sizes, expressed as the fraction of the input features that are stored in the cache by each GPU. In Quiver, GPUs cache non-overlapping sets of vertex features. These systems can cache the input features for the entire graph using a per-GPU cache size of 25%, so we consider that as the maximum cache size in this experiment.

System	Cache %	0%	10%	25%
PaGraph	CPU-GPU	13682	10829	7404
	GPU-GPU	0	0	0
	% load/epoch	79%	78%	68%
Quiver	CPU-GPU	19267	6334	0
	GPU-GPU	0	9698	14448
	% load/epoch	95%	71%	38%

Table 1: **Redundant data loading.** Averaged data volume in MB loaded by all GPUs per iteration and percentage of loading time over epoch time. (Dataset: ogbn-products. Model: GraphSage. Mini-batch size: 4096.)

Dataset	4x Micro	1x Mini	% redundancy
ogbn-products	195M	155M	25.5%
ogbn-papers100M	327M	131M	148.9%
amazon	473M	263M	79.2%

Table 2: **Redundant computation.** Total number of edges computed over one epoch when each mini-batch is sampled as 4 micro-batches of size 1024 (`4x Micro`) vs. 1 mini-batch of size 4096 (`1x Mini`).

The size of unique input features in an average mini-batch represents the minimal amount of data transferred without using caching. PaGraph and Quiver load much more data than that, even with caching, because in data parallelism, different GPUs must load the same input features redundantly. When comparing the two systems for the same cache size in Table 1, caching is more beneficial in Quiver because it uses a distributed cache, whereas PaGraph keeps independent per-GPU caches. As the size of the cache grows, Quiver transfers less data from CPU than PaGraph. Quiver, however, still needs to perform redundant loads from the CPU, and some of the CPU-GPU data transfers are replaced by GPU-GPU transfers due to cache misses. Inter-GPU transfers are faster but not free and still make up a high fraction of the epoch time.

These results show that redundant data loading is a fundamental problem in data parallelism, even when using a distributed cache and fast GPU-GPU buses. Split parallelism eliminates this problem by design.

Issue 2: Redundant computation. After loading all the input features of a micro-batch, each GPU runs the GNN model to compute the hidden features of the vertices in the upper layers. The overlaps among micro-batches create computation overheads at this stage. The target vertices of the micro-batches for the same iteration are non-overlapping sets. However, the samples of the k-hop neighborhood of those vertices

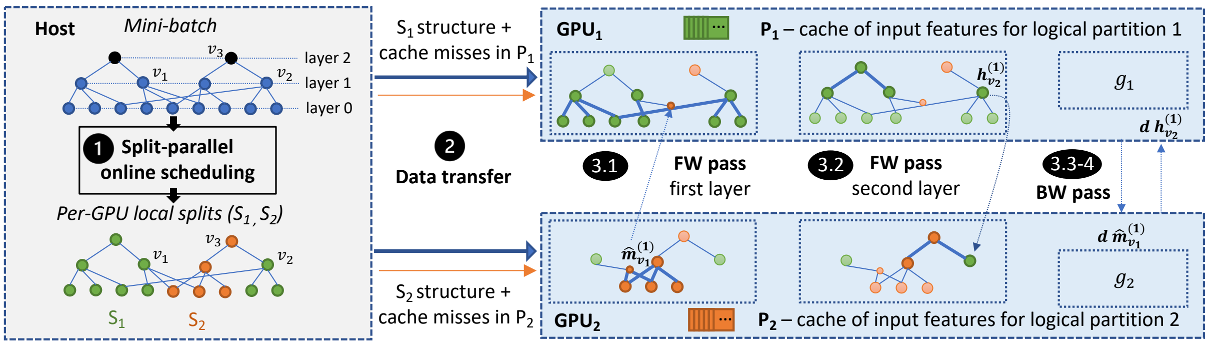

Host
Mini-batch
layer 2
layer 1
layer 0
1 Split-parallel online scheduling
Per-GPU local splits (S_1, S_2)
S_1
S_2
S_1 structure + cache misses in P_1
2 Data transfer
S_2 structure + cache misses in P_2
GPU$_1$
P_1 – cache of input features for logical partition 1
3.1 FW pass first layer
3.2 FW pass second layer
3.3-4 BW pass
GPU$_2$
P_2 – cache of input features for logical partition 2
g_1
g_2

Figure 4: Example of a Training Iteration in Split Parallel Training (GraphSage).

can have large overlaps. For this reason, the hidden features of the same vertex at the same layer can be computed by multiple GPUs, resulting in redundant computation. Note that this computational overhead is additional to the data transfer overhead discussed previously.

Table 2 evaluates the degree of computational redundancy in data-parallel training in terms of total number of edges processed during an iteration. With 4 GPUs, data parallelism creates 4 separate micro-batches, which have the total number of edges reported in the Table. Instead of creating multiple independent and overlapping micro-batches, split parallelism generates a single mini-batch for all the target vertices and splits it without overlaps, avoiding redundant computation. The total number of edges to process with this approach is between one to two orders of magnitude lower.

3 Split-Parallel Training

3.1 Overview

Each split-parallel training iteration consists of four main steps: sampling, online scheduling, data transfer, and cooperative training. Figure 4 shows an overview of one iteration, excluding the preliminary sampling step that produces a mini-batch and the final synchronous gradient aggregation and parameter update step.

In split parallelism, *sampling* returns a single mini-batch sample for all the target vertices in the mini-batch. *Split-parallel online scheduling* (Step 1 in Figure 4) splits the mini-batch into *local splits* (e.g., S_1 and S_2 in Figure 4), one for each GPU. Unlike micro-batches in data-parallel training, the local splits are non-overlapping partitions of the mini-batch sample. Layer-0 vertices are assigned based on the content of the GPU caches. Each GPU is responsible for loading and processing the input and hidden features for the vertices in its assigned split. The hidden features for vertices at the other layers are computed by only one GPU, which is selected to maximize data access locality. Splitting eliminates all redundant input data loading and hidden feature computation. The scheduling algorithm must be very fast because it is run online at each iteration on a different mini-batch.

The step following scheduling is *data transfer* (Step 2 in Figure 4). Split parallelism optimizes this step because it only transfers the features that are missing from the combined caches of all GPUs, and it transfers them to only one GPU. After data transfer, the system invokes training kernels of each local split on its corresponding GPU.

Finally, *cooperative training* (Step 3 in Figure 4) executes the local splits to train model parameters. This step could involve inter-GPU communications to complete one iteration of training because some local splits may have a few edges that require data from other GPUs. Cooperative training uses *split-aware* implementations of the message-passing functions of Section 2.1 to transparently supports GNN models written using the standard high-level SAGA operators offered by existing GNN training systems.

The GNN model in Figure 4 has two layers, each having edges across splits. Both layers require inter-GPU communication (Steps 3.1-2), which we will discuss more in detail in Section 3.3.2. After this coordination, each GPU is able to compute an embedding for the target vertices in their local split, compute the loss, and start propagating partial gradients backward. This requires GPUs to send back the partial gradients over the hidden features they received in the forward pass (Step 3.3-4). At the end of the backward pass, each GPU has gradients for the model parameters (g_1, g_2).

As we will discuss shortly and show in the evaluation, the inter-GPU communication required for cooperative training is a cost that is comparable to, and typically lower than, the redundant input data loading and redundant computation costs of data parallelism.

3.2 Split-Parallel Online Scheduling

Split-parallel online scheduling splits a mini-batch into per-GPU local splits by assigning each vertex of a mini-batch sample to one local split. The scheduling algorithm needs to meet the following two requirements:

1. It should not become a performance bottleneck of the end-to-end training pipeline.

2. it should minimize the number of edge cuts in order to reduce the inter-GPU communication during forward and backward computation.

The cheapest possible sampling algorithm would be to randomly assign each vertex in a mini-batch to a split. Besides, a uniform random distribution would ensure that all splits are perfectly balanced. However, this would produce a bad edge cut, since the probability that two endpoint vertices of an edge are in the same split is only $1/g$, where g is the number of splits. A large edge cut increases the communication cost of cooperative training. On the other hand, min-cut graph partitioning algorithms such as Metis [17] could meet the second requirement, but applying them for every mini-batch is very time-consuming.

To meet both requirements, we propose a two-step procedure that combines online splitting with offline partitioning and caching. The first step, offline partitioning, applies min-cut graph partitioning algorithms to the full input graph. The output is a vertex partitioning map, which assigns each vertex of the entire input graph to one GPU and per-GPU cache sets, which specifies the set of vertices that are cached by that GPU, if caches are used. The second step online splitting slices each mini-batch into local splits by looking up the vertex partitioning map and the per-GPU cache sets. The lookup operation is much more lightweight than applying a min-cut graph partitioning algorithm.

Specifically, the online splitting algorithm assigns vertices of the mini-batch to GPUs by looking up the vertex partitioning map. All the vertices assigned to a specific GPU and their induced computation graph is a local split for that GPU. GPUs are responsible for the computation only in their local splits. If the destination vertex v belongs to a different split than the source vertex u, the algorithm adds a special *reference vertex* for v to the split of u. The online splitting algorithm also looks up the caching set to determine which layer 0 vertices must be loaded. If a layer-0 vertex is assigned to a GPU but its feature is not cached, its features are loaded only by that GPU at the beginning of the iteration.

In the example of Figure 4, the mini-batch sample is split into two local splits S_1 and S_2 in Step 1. Vertex v_1 is included in S_2 as a reference vertex, since it belongs to S_1 but it has an incoming edge from a source vertex in S_2. Similarly, v_2 is included in S_1 as a reference vertex.

Using full-graph partitions to split mini-batches, which are subgraphs of the input graph, does not necessarily result in as balanced splits and low edge cut as running graph partitioning on each mini-batch. However, it is a good compromise to keep both the scheduling time and the training time low, as we show in our evaluation.

3.3 Cooperative Training

Splitting a mini-batch into non-overlapping local splits results in edge cuts across splits, which requires coordination across GPUs in each training iteration.

In this paper, we present the first implementation and experimental evaluation of cooperative training for mini-batch GNN training. We transparently support GNN models specified using existing message passing primitives used by practitioners to implement GNN models (see Section 2.1). Each GNN layer runs of one or more SAGAs. Two of the message passing functions, scatter and gather, operate on edges and thus need to be aware of edges that connect vertices across different local splits.

3.3.1 Split-Aware Scatter and Gather

There are two possible approaches to make scatter and gather split-aware. Recall that scatter builds an edge tensor by combining the source and destination vectors of each edge while the gather operator aggregates all messages to each destination vertex. The first approach can implement a source-to-destination scatter which sends the source vertices' features to the destination of each edge. In the backward pass, gradients flow in the opposite direction. The advantage of this approach is that all the subsequent SAGA operators can be executed locally until the end of the forward pass. This is also the default approach adopted in full-graph training systems [19].

The second approach can implement a destination-to-source scatter, which builds the edge tensor by sending the destination vertex vector to the source vertices. It can significantly reduce communication volume in mini-batch training since the total amount of destination vertices are usually much less than the source vertices. However, it requires a second shuffle for the gather operator since the gather function requires aggregating data by destination. Our empirical observation shows that more rounds of shuffles with less data are usually more costly than the a one-time shuffle with more data. Therefore, split parallelism implements the source-to-destination scatter in order to minimize the number of shuffles.

An example using the source-to-destination scatter schedule is shown in Figure 5. At the beginning of a GNN layer l, split parallelism has partitioned the hidden feature tensor $H^{(l-1)}$ by vertex across GPUs based on the splits. In this example, we compute the hidden features of a vertex v at layer l. Vertex v is in the split of GPU 1, but some of its neighbors are in GPU 2, so it is added to the split of GPU 2 as a reference vertex. The scatter operation builds an edge tensor that combines source and destination features for each edge. The tensor is partitioned by destination vertex: the scatter sends the source vertex features for all incoming edges incident to v at layer l to GPU 1, which is responsible for v. The edge-wise message function computes all messages to v locally to GPU 1, so the gather function only aggregates local messages. Finally, the update function computes the new hidden feature

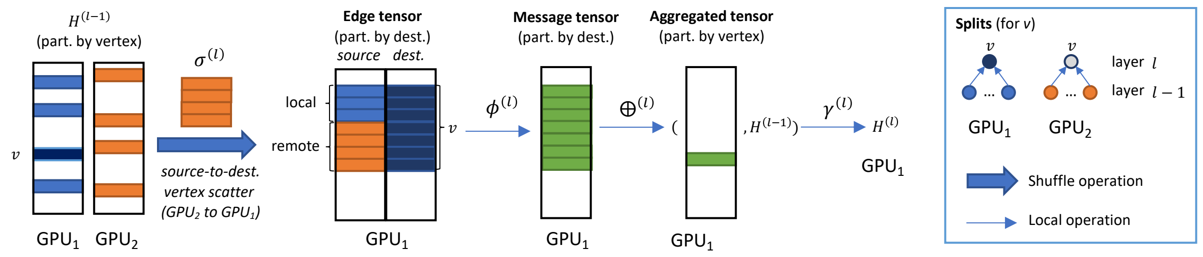


Figure 5: A SAGA computing the hidden features of vertex v at layer l over two GPUs using *source-to-destination* scatter.

vector of v. It is possible to optimize this schedule for models that do not need to build a full edge tensor to obtain messages, as we discuss shortly when considering GraphSage.

3.3.2 Cooperative Training Examples

We now discuss implementations of split-parallel training considering two popular and diverse GNN models: GraphSage [13] and Graph Attention Networks (GAT) [27]. Split parallelism runs the two models unmodified: it simply replaces local implementations of scatter with the split-aware implementation described previously.

GraphSage. A GraphSage layer builds messages directly from the input features of the source vertices of the edges. Its simplicity in the message function allows us to pre-aggregate feature vectors before the scatter operation in order to reduce the communication volume.

Figure 4 shows an example of cooperative training using a GraphSage model [13] with two layer. In the first layer, vertex v_1 is in split S_1 but has incoming edges from vertices in split S_2. GPU 2 computes the partial aggregated message $\hat{m}_{v_1}^{(1)}$ and then executes a shuffle (i.e., source-to-destination scatter) to send $\hat{m}_{v_1}^{(1)}$ to GPU 1 (see Step 3.1). GPU 1 computes the final aggregation to compute the hidden features of v_1 at layer 1. GPU 1 also locally compute the hidden features $h_{v_2}^{(1)}$. In the second layer, $h_{v_2}^{(1)}$ on GPU 1 is scattered to GPU 2 to compute messages for v_3 (see Step 3.2). At the end of layer 2, the GNN computes the loss and starts the backward propagation. The GPUs need to send the gradients for the activations they received back to the sender. In the example of Figure 4, GPU 1 sends $d\hat{m}_{v_1}^{(1)}$ and GPU 2 sends $dh_{v_2}^{(1)}$ (see Steps 3.3-4 in Figure 4).

GAT. The GAT model computes attention scores for each edge before computing the messages. The attention score of an edge $e(u,v)$ at layer l is computed as:

$$\alpha_{(u,v)}^{(l)} = \exp(e_{(u,v)}^{(l-1)}) / \sum_{(u',v)\in E^{(l)}} \exp(e_{(u',v)}^{(l-1)}).$$

where $e_e^{(l)}$ is an attention coefficient for the edge, computed using an attention function a over the features of the source and destination, $h_u^{(l-1)}$ and $h_v^{(l-1)}$.

A GAT layer l can be implemented using two SAGA operations. The first SAGA operation computes the denominator of $\alpha_{(u,v)}^{(l)}$ for all the incoming edges of each vertex v at layer l. It requires a shuffle operation using split-aware scattering. The second SAGA computes the features of destination vertices and is completely local. Specifically, in the first SAGA, a split-aware scatter first collects the source and destination features for all incoming edges for v, which requires a shuffle. The message function then computes the sum of the denominator over the local edges. Next, a gather aggregates the messages for each destination vertex to compute the denominator of $\alpha_{(u,v)}^{(l)}$. The GAT layer then proceeds with the second SAGA to compute the feature of each destination vertex v. A local scatter builds edge tensors that include the features of the source and destination and the denominator of $\alpha_{(u,v)}^{(l)}$. The message function calculates $\alpha_{(u,v)}^{(l)}$ for each edge, then computes the edges' attention score using the function a, and obtains the message. Finally, a gather aggregates the messages and the update function computes the hidden features of the vertices in layer l.

In summary, both GraphSage and GAT perform up to two shuffles per layer, one in the forward and one in the backward propagation.

4 Implementation

We implemented split-parallel training in GSplit, a multi-GPU GNN training system supporting synchronous training. GSplit is implemented on top of DGL 0.8.2 and Torch 1.8.0 with CUDA 11.1. The implementation consists of about 10k lines of code. The sampling and splitting code is in C++, the remaining code in Python.

Sampling and scheduling. GSplit runs sampling and splitting on the CPU to easily scale to large graphs that do not fit in GPU memory. It uses one thread to randomly shuffle the set of training nodes and create a set of target vertices for each mini-batch. Sampling and scheduling are performed using multiple *sampler* processes, each working independently in parallel on one mini-batch to sample it and split vertices and edges as they are sampled. It also uses one *trainer* process per GPU, each responsible for invoking kernels on that GPU.

Dataset	# Nodes	# Edges	# Feat
ogbn-products (PR)	2.4M	62M	100
ogbn-papers100M (PA)	111M	1.6B	128
Amazon (AM)	1.56M	168M	200

Table 3: Datasets used for the evaluation

In split-parallel training, all GPUs must work on splits of the same mini-batch at the same time. GSplit uses one trainer as a leader, which selects which mini-batch to process next among the ones currently complete. To avoid making the leader a bottleneck, GSplit separates the data and control paths. Workers write mini-batches into shared memory and send a handle to the leader. The leader picks one handle and instructs all other trainers to operate on that mini-batch.

Partitioning and caching policy. GSplit can use any offline partitioning and caching policy. By default, it uses Metis for offline graph partitioning [17]. After partitioning the graph offline, GSplit caches the highest degree vertices of each partition in the corresponding GPU, unless the user specifies a different policy. It allows the user to decide how much GPU memory should be dedicated to caching.

Cooperative training. The cooperative training we discussed requires a split-aware version of the scatter operator. It uses fast direct GPU-GPU buses such as NVLink if available in the system or the PCIe bus otherwise. Shuffles use data coalescing to have a GPU send at most one message to each other GPU. Vertices in the same local split have contiguous ids, so that the online scheduling algorithm and our split-aware operators can find the local split of a vertex using a simple range check.

Integration with DGL and Torch. GSplit implements sampling, split-parallel online scheduling, feature extraction, feature loading, and feature caching from scratch. It runs all GPU-local GNN operators by invoking unmodified DGL. The message and update operators are fully local, whereas the split-aware scatter and gather functions interleave local steps, executed by invoking DGL, and shuffle steps, which are implemented by GSplit. GSplit uses NCCL to implement inter-GPU peer-to-peer communication and exploits NVLink if available. It invokes NCCL through Torch. GSplit uses Torch also for gradient aggregation and model update.

5 Evaluation

5.1 Experiment Settings

Hardware setup. We run our experiments on two types of hosts, both with 4 GPUs but with different GPU-GPU interconnects. The first host type, which we call `PCIe`, has four NVIDIA GeForce RTX 3090 Ti GPUs, each with 24 GB memory, and four Intel(R) Xeon(R) Silver 4214R CPU @ 2.40GHz, each with 12 cores and 192 GB RAM. The GPUs in host are connected using a PCIe 3.0 bus.

The second host type, which we call `NVLink`, is an AWS EC2 P3.8xlarge instance. It has four NVIDIA V100 GPUs with 16GB memory and Xeon E5-2686 v4 @ 2.70GHz, with 18 cores and 244 GB RAM. GPUs are connected to the CPU with a PCIe 3.0 bus and with each other via NVLink.

Datasets. We use three datasets listed in Table 3. Two of the datasets are from the Open Graph Benchmark (OGB), a standard benchmark for GNN training [1]. We use the two largest graphs in the benchmark: products and papers100M. We also use the Amazon dataset from [36].

GNN models. We consider two popular and diverse GNN models, which we described in Section 3.3.2: GraphSage [13] and GAT [27]. Both GraphSage and GAT perform up to two shuffles per layer. We use the standard neighbour sampling algorithm, with a fanout of 20 and three hops. We use a default hidden size of 16, as used in [13,27], and a batch size of 4096.

Baselines. All state-of-the-art systems for mini-batch multi-GPU GNN training use data-parallel training. We use three systems as baselines: DGL, PaGraph, and Quiver.

- **DGL** (Deep Graph Library) is a standard library for GNN training [2]. We use DGL version 0.8.2, the same we use as a component of GSplit. DGL does not use GPU caching.

- **PaGraph** uses GPU caching to reduce data transfer overheads [18]. PaGraph always loads missing input features from the host memory through the CPU-GPU bus (PCIe). We run the publicly-available implementation of PaGraph, updated to run with DGL 0.8.2.

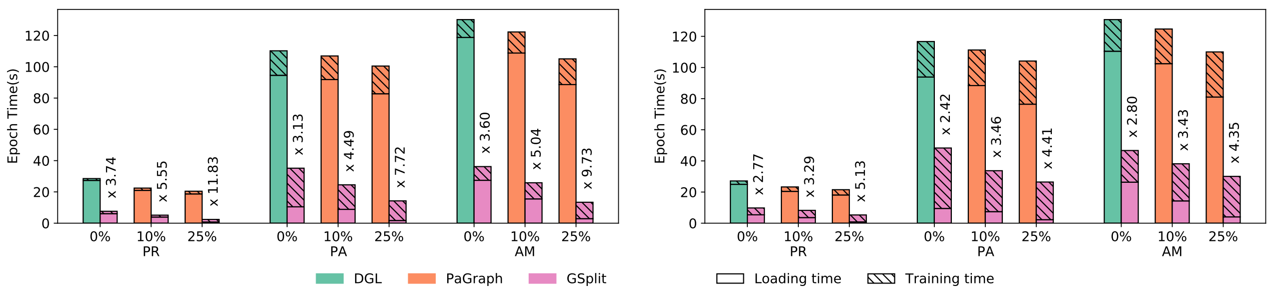
- **Quiver** is a recent GNN training system that leverages fast direct GPU-GPU buses like NVLink [4] to reduce data loading overhead. Quiver partitions the input features across GPU caches. It loads missing features from other GPUs' caches whenever possible.

All systems use CPU-based sampling to scale to graphs that do not fit in GPU memory. They all run 32 sampler processes in parallel with training. GSplit uses the same degree-based caching policy as PaGraph and Quiver for consistency.

5.2 Running Time Comparison

We compare the running times per-epoch for all systems. All systems except DGL make use of the local cache. Sampling, and slicing for GSplit, happen in parallel with training. Therefore, we break down the epoch time of a trainer process into two sequential steps: loading time and training time.

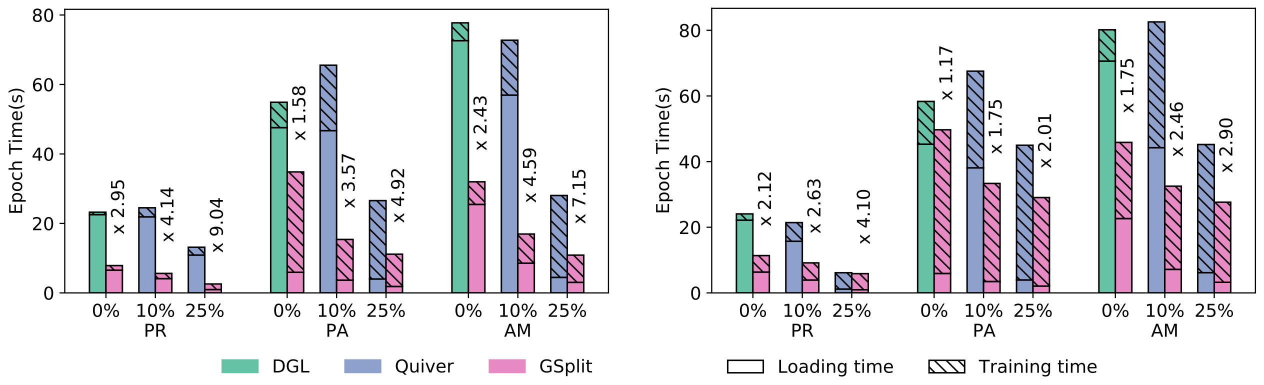
We compare the performance of different GNN training systems on our two hosts, which use different buses.



(a) GraphSage

(b) GAT

Figure 6: Epoch time (in seconds) on the `PCIe` host for different GPU cache sizes. Speedups are over DGL.



(a) GraphSage

(b) GAT

Figure 7: Epoch time (in seconds) on the `NVLink` host for different GPU cache sizes. Speedups are over DGL.

5.2.1 Performance using PCIe

We first consider the `PCIe` host. We use DGL as the baseline in configurations without a GPU cache a PaGraph in configurations with a cache. We do not run Quiver in this setup since its distributed caching strategy requires NVLink. Figure 6 reports the epoch time comparison. Overall, GSplit outperforms both baselines significantly, sometimes by one order of magnitude. We now discuss the impact of split parallelism in terms of loading and training time.

Loading time. The most important factor in this speedup is the reduction in loading time, which is a significant bottleneck in baselines. When each GPU caches 25% of the graph or more, the combined distributed cache contains all the features. The scheduler schedules computation based on the placement of the cached features, so there are no cache misses. GSplit only has to load the computation graph and this results in the lowest loading time.

DGL has the highest loading time because it does not use caching. PaGraph's loading time is lower than DGL thanks to caching. However, it still has a much higher loading overhead than GSplit for two reasons. First, PaGraph maintains an independent cache at each GPU, with large overlaps among the caches. Unlike GSplit, it can only cache a subset of the input features. Second, it loads data redundantly because it uses data parallelism, an overhead GSplit avoids.

Figure 6 also shows the effect of using different cache sizes. We consider a cache size of up to 25%, which is where GSplit achieves full caching. GSplit can reduce loading time compared to both baselines even when the distributed cache does not contain all the features. Consider for example the 0% case, where caching is not used. In this case, all the input features of the mini-batch need to be loaded from the CPU memory. However, GSplit loads each feature vector to only one GPU, whereas the other data parallel systems need to perform redundant loads.

Training time. GSplit does not show large speedups in terms of training time on the `PCIe` host. On the one hand, it has a lower computation cost than the other systems because it eliminates redundant computation. The effect is particularly visible for the PA and AM datasets, which have larger overlaps among micro-batches (see Table 2). On the other hand, it has the additional communication and synchronization cost of cooperative training, which the baseline systems do not have. Eliminating redundant computation does not always entirely offset this cost in the `PCIe` host, which has a slower inter-GPU communication. However, the training time of GSplit is still in line with the baseline systems. As expected, the cache size has no significant impact on training time, as shown in Figure 6.

5.2.2 Performance using NVLink

In the experiments on the `NVLink` host, we include Quiver since its distributed cache is designed to leverage the NVLink bus and is a superior baseline than PaGraph. Figure 7 reports the epoch time comparison. GSplit shows large speedups of up to one order of magnitude in this configuration too, this time for both loading and training.

Loading time. As before, the loading time is strongly influenced by the size of the GPU caches. In the 25% cache case, both Quiver and GSplit can cache the input features of the entire graph thanks to their distributed cache. Neither system needs to load input features from the CPU memory. However, Quiver replaces loads from the CPU memory with loads from other GPUs' memory, as shown in Table 1. These data transfers are faster thanks to NVLink, so data loading time is significantly reduced compared to DGL. However, they are still not free and represent a low but not negligible cost. GSplit can take advantage of the caches much more effectively than Quiver since it does not require loading input features at all.

The drawback of redundant data transfers is particularly evident when the cache size is lower than 25%. In this case, both Quiver and GSplit need to load features from the CPU memory. The loading time of Quiver grows significantly and becomes a major bottleneck. In contrast, GSplit's loading time also grows, but much less significantly.

Training time. GSplit shows significant speedups compared to DGL and Quiver in terms of training time because it eliminates redundant computation. The additional communication cost of cooperative training is greatly reduced thanks to NVLink, leading to a clearly positive net gain, particularly for PA and AM, the two datasets with larger computational redundancy. As in the previous case, the training time of GSplit is not affected by the size of the cache, as shown in Figure 7.

5.2.3 Effect of Mini-Batch Size

We now consider the effect of the mini-batch size on the running time. For these experiments, we consider a cache size of 25%. The results are shown in Figures 8 and 9.

As the size of the mini-batch grows, the epoch time for all systems decreases since the system performs fewer training iterations. GSplit achieves consistent speedups regardless of the mini-batch size and on both hosts. It tends to do slightly better on larger batch sizes since they present larger overlaps among micro-batches.

5.3 Online Scheduling Evaluation

Cost of Online Splitting. Split parallelism requires an additional step in the training pipeline executed at each iteration: online splitting, which occurs after a mini-batch sample is produced and before training starts. It is important to ensure that this step does not become a performance bottleneck. We now perform a micro-benchmark to evaluate the throughput of sampling and splitting in GSplit compared to DGL.

We run a single thread for both DGL and GSplit on the `PCIe` host and measure the time it takes to produce all the data required for a training iteration, considering a system with four GPUs. A data-parallel system like DGL consumes four micro-batch samples per training iteration, each for one fourth of the target vertices in the mini-batch. A split-parallel system like GSplit consumes four splits of a single mini-batch. Even though different samples are sampled in parallel, our single-thread experiment estimates the throughput at which sampling can operate.

The results are shown in Table 4. Splitting adds some overhead to sampling: for each vertex, we need to check which offline partition it belongs to and whether it is cached. DGL does not perform splitting or caching, so it does not have this overhead. However, sampling for split-parallel training also benefits from the elimination of redundancy. Instead of sampling four independent and potentially overlapping micro-batches, a split-parallel sampler only needs to sample one larger mini-batch. The combined effect of these two factors is that the overhead of online splitting is at most 74% of the time spent on sampling.

Offline-Online Split Quality. GSplit uses a combination of offline partitioning and online splitting to reduce the edge cut. Previous experiments have shown that this solution achieves both high training performance and low splitting time. Here, we examine the split quality by measuring *edge skew* and *percentage of local edges*.

Specifically, *edge skew* is the maximum number of edges assigned to a split in an iteration minus the minimum divided by the average. It aims to capture whether the computation workloads are balanced across GPUs. A large edge skew means the local splits are not balanced. Splitting a mini-batch in a random uniform way (called *random uniform partition*) would result in zero edge skew and thus perfect load balancing. However, it could lead to many edge cuts and thus high inter-GPU communication overhead.

A *local edge* is an edge whose source and destination vertex are in the same split. Otherwise it is remote and introduce a reference vertex. *Percentage of local edges* is the ratio between the number of local edges and the total edges in a mini-batch. Higher percentage of local edges are better since

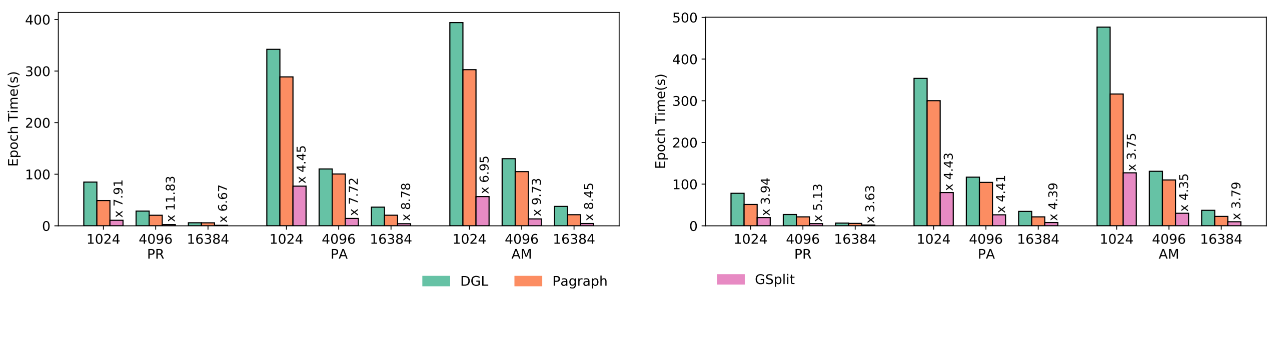


Figure 8: Epoch time (in seconds) on the `PCIe` host for different mini-batch sizes.

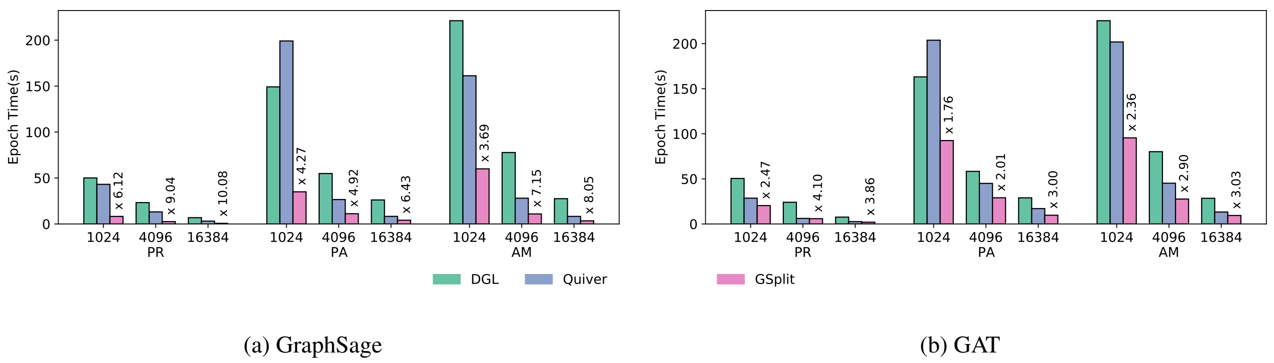


Figure 9: Epoch time (in seconds) on the `NVLink` host for different bacth sizes. Speedups are over DGL.

it indicates lower inter-GPU communication costs. A *random uniform partition* would result in low percentage of local edges (e.g., an average of 25% when splitting a mini-batch into four local splits), as explained in Section 3.2.

Table 5 reports the edge skew (column "Edge Skew") and percentage of local edges (column "Local Edges") of our offline-partitioning online-splitting approach using different datasets and batch sizes, when producing 4 splits per mini-batch. Compared to a uniform random partition approach, our approach achieves higher skew but much higher percentage of local edges (83% - 96%), indicating much lower inter-GPU communication cost.

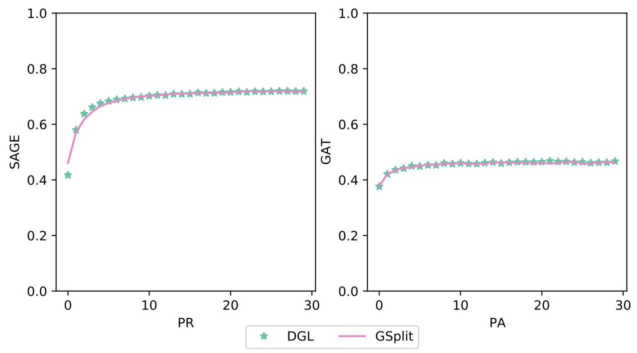


Figure 10: Test Accuracy

5.4 Accuracy Evaluation

To validate the correctness of GSplit, we compare its test accuracy with DGL. Test accuracy is the accuracy of the GNN model on the test dataset, which is not seen during training.

Figure 10 compares the accuracy at different epochs for the two OGB datasets, PR, and PA, using GraphSage and GAT respectively. GSplit's accuracy at each epoch matches DGL's accuracy, as shown in the Figure. GSplit's convergence speed, however, is much faster since each epoch takes a much shorter time to complete.

6 Related Work

Our work focuses on single-host multi-GPU mini-batch training. Distributed mini-batch training is another important area of research for GNN training systems. Systems like DistDGL [37] and AliGraph [38] use data-parallel mini-batch training to scale to large graphs. P^3 introduces pipelined push-pull hybrid parallelism for distributed mini-batch training [9]. Marius++ runs data-parallel GNN training on large graphs using a single GPU and out-of-core approach rather than a

Algorithm	PR 1024	PR 4096	PR 16384	PA 1024	PA 4096	PA 16384	AM 1024	AM 4096	AM 16384
DGL sample	101.701	98.230	62.723	88.844	71.453	59.916	190.542	128.012	64.928
GSplit sample+split	141.485	91.448	44.958	139.557	123.972	94.477	266.657	143.772	66.641

Table 4: Average time (in seconds) to produce the mini-batch, or the micro-batches, for one training epoch using a single sampling thread, with different datasets and batch sizes (i.e., 1024, 4086, and 16384).

Graph	Batch Size	Edge Skew	Local Edges
PR	1024	0.997	0.93
	4096	0.999	0.93
	16384	1.012	0.94
PA	1024	1.326	0.96
	4096	1.323	0.96
	16384	1.315	0.95
AM	1024	0.514	0.83
	4096	0.517	0.84
	16384	0.541	0.83

Table 5: Edge skew and percentage of local edges using the offline partitioning/online splitting approach in GSplit.

distributed system [28]. For a survey on distributed GNN training, we refer the reader to [25]

Many systems focus on full-graph GNN training, which is a different problem than mini-batch training as discussed in Sections 1 and 3.2. Other notable but less directly related work in full-graph training includes Dorylus [26], which uses serverless functions, DGCL [5], which optimizes communication in distributed training, and FlexGraph, which support aggregation from indirect neighbors [29]. NeutronStar is a distributed full-graph training system that uses standard full-graph training for some target vertices and fetches the k-hop neighborhood similar to data parallelism for others [31].

Other work has explored other types of bottlenecks than the ones discussed in this paper. Mini-batch sampling and extraction can become bottlenecks in some cases. Existing work has explored using in-GPU sampling to alleviate this bottleneck. NextDoor proposes a programming API and a runtime to speed up in-GPU sampling [15]. C-SAW is another system with similar goals [22]. GPU sampling in these systems does not scale to large graphs that cannot be stored in the GPU memory. GNNLab is a multi-GPU mini-batch training system that uses a factorized approach to share GPU resources between sampling and training to leave more GPU memory for sampling [34]. Adding support for in-GPU sampling and splitting to GSplit is an interesting avenue of future work. GNNLab also proposes a pre-sampling technique to select vertices to cache based on sampling probability, using an independent per-GPU cache. GSplit can leverage this and other caching policies to increase its cache hit rate, also thanks to its distributed cache design. Finally, some work proposes dedicating a subset of the GPU threads to load data directly from the CPU memory concurrently with training [21]. This speeds up loading but it requires a double in-GPU buffer to load features. None of these systems explores different paradigms to parallelize GNN training.

Work on optimizing GPU kernels for GNN training (for example SeaStar [32]) is complementary to the work described in this work. CPU-GPU data transfers can also be limited by biasing the sampling algorithms to prioritize sampling from graph partitions that store on GPUs [6, 24]. However, this approach is not transparent as it relies on ML practitioners to change the model design to be cache-aware, which affects training convergence and accuracy.

7 Conclusions

This paper discusses some inherent performance limitations of the common data-parallel approach to mini-batch GNN training. It proposes split parallelism, a novel parallel training paradigm that eliminates the computation and data transfers redundancies of data parallelism. Split parallelism requires new approaches for online scheduling and cooperative training. We implement GSplit, a multi-GPU mini-batch training system based on split parallelism, and show that it outperforms state-of-the-art systems in a variety of scenarios.

By moving computation to data, split parallelism significantly reduces data loading time. While this result was expected given the premise of split-parallel training, showing that cooperative training can match and sometimes surpass the performance of data parallel training is an interesting empirical result. Similarly, we show that offline pre-processing can effectively support fast online splitting while achieving good training performance. We believe that split parallelism represents an interesting new avenue to scale GNN training to large graphs and systems.

References

[1] Open graph benchmark - node property prediction. https://ogb.stanford.edu/docs/nodeprop/, 2021.

[2] Deep graph library. https://www.dgl.ai, 2022.

[3] Nvlink. https://www.nvidia.com/en-us/data-center/nvlink/, 2022.

[4] Quiver project. https://github.com/quiver-team/torch-quiver, 2022.

[5] Zhenkun Cai, Xiao Yan, Yidi Wu, Kaihao Ma, James Cheng, and Fan Yu. Dgcl: an efficient communication library for distributed gnn training. In *Proceedings of the Sixteenth European Conference on Computer Systems*, pages 130–144, 2021.

[6] Jialin Dong, Da Zheng, Lin F Yang, and Geroge Karypis. Global neighbor sampling for mixed cpu-gpu training on giant graphs. *arXiv preprint arXiv:2106.06150*, 2021.

[7] Matthias Fey and Jan Eric Lenssen. Fast graph representation learning with pytorch geometric. *arXiv preprint arXiv:1903.02428*, 2019.

[8] Fabrizio Frasca, Emanuele Rossi, Davide Eynard, Benjamin Chamberlain, Michael Bronstein, and Federico Monti. Sign: Scalable inception graph neural networks. In *ICML 2020 Workshop on Graph Representation Learning and Beyond*, 2020.

[9] Swapnil Gandhi and Anand Padmanabha Iyer. P3: Distributed deep graph learning at scale. In *15th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 21)*, pages 551–568, 2021.

[10] Thomas Gaudelet, Ben Day, Arian R Jamasb, Jyothish Soman, Cristian Regep, Gertrude Liu, Jeremy BR Hayter, Richard Vickers, Charles Roberts, Jian Tang, et al. Utilising graph machine learning within drug discovery and development. *arXiv preprint arXiv:2012.05716*, 2020.

[11] Justin Gilmer, Samuel S Schoenholz, Patrick F Riley, Oriol Vinyals, and George E Dahl. Neural message passing for quantum chemistry. In *International conference on machine learning*, pages 1263–1272. PMLR, 2017.

[12] Yoav Goldberg. Neural network methods for natural language processing. *Synthesis lectures on human language technologies*, 10(1):1–309, 2017.

[13] William L. Hamilton, Rex Ying, and Jure Leskovec. Inductive Representation Learning on Large Graphs. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS'17, 2017.

[14] Weihua Hu, Matthias Fey, Hongyu Ren, Maho Nakata, Yuxiao Dong, and Jure Leskovec. Ogb-lsc: A large-scale challenge for machine learning on graphs. *arXiv preprint arXiv:2103.09430*, 2021.

[15] Abhinav Jangda, Sandeep Polisetty, Arjun Guha, and Marco Serafini. Accelerating graph sampling for graph machine learning using gpus. In *European Conference on Computer Systems (EuroSys)*, 2021.

[16] Zhihao Jia, Sina Lin, Mingyu Gao, Matei Zaharia, and Alex Aiken. Improving the accuracy, scalability, and performance of graph neural networks with roc. *Proceedings of Machine Learning and Systems*, 2:187–198, 2020.

[17] George Karypis and Vipin Kumar. Metis: A software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices. 1997.

[18] 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*, pages 401–415, 2020.

[19] 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, 2019.

[20] Vasimuddin Md, Sanchit Misra, Guixiang Ma, Ramanarayan Mohanty, Evangelos Georganas, Alexander Heinecke, Dhiraj Kalamkar, Nesreen K Ahmed, and Sasikanth Avancha. Distgnn: Scalable distributed training for large-scale graph neural networks. *arXiv preprint arXiv:2104.06700*, 2021.

[21] Seung Won Min, Kun Wu, Sitao Huang, Mert Hidayetoğlu, Jinjun Xiong, Eiman Ebrahimi, Deming Chen, and Wen-mei Hwu. Large graph convolutional network training with gpu-oriented data communication architecture. *Proc. VLDB Endow.*, 14(11):2087–2100, oct 2021.

[22] Santosh Pandey, Lingda Li, Adolfy Hoisie, Xiaoye S Li, and Hang Liu. C-saw: a framework for graph sampling and random walk on gpus. *arXiv preprint arXiv:2009.09103*, 2020.

[23] Xiaojuan Qi, Renjie Liao, Jiaya Jia, Sanja Fidler, and Raquel Urtasun. 3d graph neural networks for rgbd semantic segmentation. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 5199–5208, 2017.

[24] Morteza Ramezani, Weilin Cong, Mehrdad Mahdavi, Anand Sivasubramaniam, and Mahmut T Kandemir. Gcn meets gpu: Decoupling" when to sample" from" how to sample". In *NeurIPS*, 2020.

[25] Yingxia Shao, Hongzheng Li, Xizhi Gu, Hongbo Yin, Yawen Li, Xupeng Miao, Wentao Zhang, Bin Cui, and Lei Chen. Distributed graph neural network training: A survey. *arXiv preprint arXiv:2211.00216*, 2022.

[26] John Thorpe, Yifan Qiao, Jonathan Eyolfson, Shen Teng, Guanzhou Hu, Zhihao Jia, Jinliang Wei, Keval Vora, Ravi Netravali, Miryung Kim, et al. Dorylus: affordable, scalable, and accurate gnn training with distributed cpu servers and serverless threads. In *15th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 21)*, pages 495–514, 2021.

[27] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio. Graph attention networks. *arXiv preprint arXiv:1710.10903*, 2017.

[28] Roger Waleffe, Jason Mohoney, Theodoros Rekatsinas, and Shivaram Venkataraman. Marius++: Large-scale training of graph neural networks on a single machine. *arXiv preprint arXiv:2202.02365*, 2022.

[29] Lei Wang, Qiang Yin, Chao Tian, Jianbang Yang, Rong Chen, Wenyuan Yu, Zihang Yao, and Jingren Zhou. Flexgraph: a flexible and efficient distributed framework for gnn training. In *Proceedings of the Sixteenth European Conference on Computer Systems*, pages 67–82, 2021.

[30] Minjie Wang, Lingfan Yu, Da Zheng, Quan Gan, Yu Gai, Zihao Ye, Mufei Li, Jinjing Zhou, Qi Huang, Chao Ma, et al. Deep graph library: Towards efficient and scalable deep learning on graphs. 2019.

[31] Qiange Wang, Yanfeng Zhang, Hao Wang, Chaoyi Chen, Xiaodong Zhang, and Ge Yu. Neutronstar: distributed gnn training with hybrid dependency management. In *Proceedings of the 2022 International Conference on Management of Data*, pages 1301–1315, 2022.

[32] Yidi Wu, Kaihao Ma, Zhenkun Cai, Tatiana Jin, Boyang Li, Chenguang Zheng, James Cheng, and Fan Yu. Seastar: vertex-centric programming for graph neural networks. In *Proceedings of the Sixteenth European Conference on Computer Systems*, pages 359–375, 2021.

[33] Dongxu Yang, Junhong Liu, Jiaxing Qi, and Junjie Lai. Wholegraph: a fast graph neural network training framework with multi-gpu distributed shared memory architecture. In *2022 SC22: International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 767–780. IEEE Computer Society, 2022.

[34] Jianbang Yang, Dahai Tang, Xiaoniu Song, Lei Wang, Qiang Yin, Rong Chen, Wenyuan Yu, and Jingren Zhou. Gnnlab: a factored system for sample-based gnn training over gpus. In *Proceedings of the Seventeenth European Conference on Computer Systems*, pages 417–434, 2022.

[35] 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*, pages 974–983, 2018.

[36] Hanqing Zeng, Hongkuan Zhou, Ajitesh Srivastava, Rajgopal Kannan, and Viktor Prasanna. Graphsaint: Graph sampling based inductive learning method. In *International Conference on Learning Representations*, ICLR '20, 2020.

[37] Da Zheng, Chao Ma, Minjie Wang, Jinjing Zhou, Qidong Su, Xiang Song, Quan Gan, Zheng Zhang, and George Karypis. Distdgl: Distributed graph neural network training for billion-scale graphs. *arXiv preprint arXiv:2010.05337*, 2020.

[38] Rong Zhu, Kun Zhao, Hongxia Yang, Wei Lin, Chang Zhou, Baole Ai, Yong Li, and Jingren Zhou. Aligraph: A comprehensive graph neural network platform. *arXiv preprint arXiv:1902.08730*, 2019.

[39] C Lawrence Zitnick, Lowik Chanussot, Abhishek Das, Siddharth Goyal, Javier Heras-Domingo, Caleb Ho, Weihua Hu, Thibaut Lavril, Aini Palizhati, Morgane Riviere, et al. An introduction to electrocatalyst design using machine learning for renewable energy storage. *arXiv preprint arXiv:2010.09435*, 2020.