GSplit: Scaling Graph Neural Network Training on Large Graphs via Probabilistic Splitting

Sandeep Polisetty [*1] **Juelin Liu** [*1] **Jacob Falus** [1] **Yi Ren Fung** [2] **Seung Hwan Lim** [3] **Hui Guan** [1]
Marco Serafini [1]

Abstract

Graph neural networks (GNNs), an emerging class of machine learning models for graphs, have gained popularity for their superior performance in various graph analytical tasks. Mini-batch training is commonly used to train GNNs on large graphs, and data parallelism is the standard approach to scale mini-batch training across multiple GPUs. Data parallel approaches contain redundant work as subgraphs sampled by different GPUs contain significant overlap. To address this issue, we introduce a hybrid parallel mini-batch training paradigm called split parallelism. Split parallelism avoids redundant work by splitting the sampling, loading, and training of each mini-batch across multiple GPUs. Split parallelism, however, introduces communication overheads that can be more than the savings from removing redundant work. We further present a lightweight partitioning algorithm that probabilistically minimizes these overheads. We implement split parallelism in GSplit and show that it outperforms state-of-the-art mini-batch training systems like DGL, Quiver, and P^3.

1 Introduction

Graph neural networks (GNNs) have demonstrated superior performance in various graph analytics tasks. Widely used systems like DGL (Wang et al., 2020) and PyTorch Geometric (Fey & Lenssen, 2019), as well as production systems like AliGraph (Zhu et al., 2019), employ mini-batch training, which is generally more effective than full-graph training at scaling to multiple GPUs (Bajaj et al., 2024). To accelerate GNN model training across multiple GPUs, these systems utilize *data parallelism*. At each iteration, they sample multiple independent *micro-batches*, one per GPU. Each GPU independently loads its micro-batch and computes gradients using a local replica of the GNN model. Each micro-batch consists of a partition of the target vertices in the mini-batch along with a sample of their k-hop neighbors.

One drawback of data-parallel training for GNNs is redundant data movements and computations (see Figure 1(a)). The k-hop neighbors of target vertices in different micro-batches overlap. Thus, the same vertices appear in multiple micro-batches. This redundancy leads to overheads across all stages of the graph neural network model training pipeline – sampling, loading, and training.

In this paper, we propose using a different hybrid parallelism approach tailored to mini-batch training that eliminates redundant data loading and computation. During each training iteration, the sampling step samples one mini-batch for all GPUs, divides it on the fly into non-overlapping partitions called *splits*, and assigns each split to a specific GPU (see Figure 1(c)). Now, only one GPU is responsible for the sampling, feature loading, and computation steps associated with a vertex. GPUs then cooperatively execute the training of an iteration on the same mini-batch. Each GPU operates only on the vertices within its assigned split. It shuffles intermediate results with other GPUs at each GNN layer. We refer to this parallelism technique as *split parallelism*. We implement split parallelism in GSplit, a scalable multi-GPU GNN training system. GSplit delivers state-of-the-art performance in mini-batch GNN training by eliminating redundancy. However, applying split parallelism to mini-batch GNN training requires solving several challenges.

The primary technical challenge in split parallelism is devising a *splitting algorithm* that satisfies three key requirements: (i) minimize the cost of shuffles while balancing load to avoid stragglers (ii) on-the-fly splitting at each iteration without becoming a performance bottleneck, (iii) Split vertices in alignment with the location of their cached input features on the GPU. A straightforward solution to the first requirement would be to run a min-cut graph partitioning algorithm online on each sampled mini-batch. However, this would not satisfy the second requirement as the partitioning problem is NP-hard and difficult to parallelize, nor

[*]Equal contribution [1]University of Massachusetts, Amherst [2]University of Illinois Urbana-Champaign [3]Oak Ridge National Laboratory. Correspondence to: Sandeep Polisetty <spolisetty@umass.edu>.

Proceedings of the 8th MLSys Conference, Santa Clara, CA, USA, 2025. Copyright 2025 by the author(s).

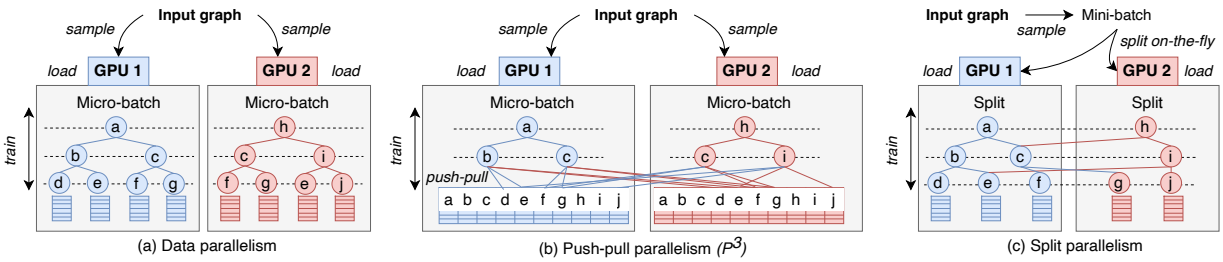

Figure 1. Comparison between data parallel, push-pull parallel, and the proposed split parallel training.

the third, since it would not consider where input features are cached. Many full-graph training systems shuffle intermediate data among GPUs at each GNN layer and use sophisticated scheduling algorithms to minimize communication and balance load (Cai et al., 2021; Jia et al., 2020; Ma et al., 2019; Md et al., 2021; Wan et al., 2022b; 2023; Wang et al., 2023b; Yang et al., 2023). However, these algorithms do not apply directly to splitting. In full-graph training, the work performed in each epoch is known in advance, since the batch is static and training occurs on the entire graph. In contrast, a splitting algorithm must dynamically split sampled mini-batches at each iteration, which operates on a much shorter time scale than a full-graph epoch.

To solve this problem, we propose a *probabilistic splitting algorithm* that achieves a negligible online overhead: on a randomly sampled mini-batch, it *provably* minimizes the *expected* communication cost among splits and balances the *expected* load per split. Compared to using a standard offline graph partitioning algorithm that does not provide probabilistic guarantees, our algorithm speeds up the end-to-end training time of our GSplit system by up to 1.7×.

Another challenge is to preserve the programming abstractions of data-parallel training systems. Besides simplifying development, preserving the programming abstractions of data-parallel training allows us to leverage work on optimizing single-GPU kernels for GNN sampling and training (Fan et al., 2023; Fu et al., 2022; Gong et al., 2023; Jangda et al., 2021; Wu et al., 2021; Ye et al., 2023; Zhou et al., 2023). In data-parallelism, vertices in a micro-batch layer are always local, whereas in split parallelism a layer may contain a mix of local and remote vertices. Further, GSplit provides a *split/shuffle* API that hides the low-level data shuffling details from the end users.

Our evaluation across multiple large graphs and GNN models shows that GSplit outperforms the state-of-the-art systems like DGL (Wang et al., 2020) and Quiver (Tan et al., 2023) by up to 4.4× and 1.9× respectively (2.4× and 1.4× on average). We also implement and evaluate the push-pull parallelism approach of P^3 (Gandhi & Iyer, 2021) in a single-host multi-GPU system and show that GSplit outperforms it by up to 4.1× (2.4× on average). GSplit's splitting

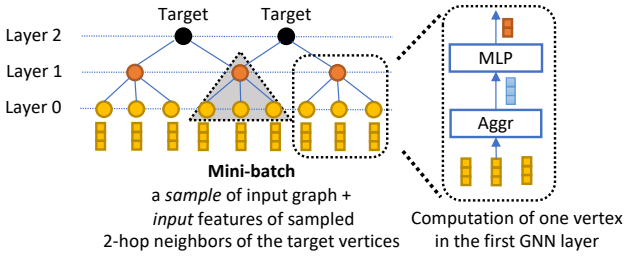

Figure 2. Example of a mini-batch.

algorithm is the key to achieving these speedups, and it is effective in balancing load among splits and reducing the cost of shuffles.

Overall, we make the following contributions:

- We propose split parallelism to eliminate redundant input feature loading and computation (Section 3)
- We propose a lightweight online splitting algorithm that uses a probabilistic approach to minimize the expected communication cost and balance the expected load per split at each iteration (Section 4).
- We develop GSplit's split/shuffle API, which supports optimized single-GPU kernels for sampling and training (Section 5).

2 Background and Motivation

This section first introduces mini-batch GNN training, and then elaborates on the limitations of existing optimizations.

Mini-Batch GNN Training and Data Parallelism A GNN model is defined as a sequence of *GNN layers*. During each mini-batch training iteration, there are three phases: *sampling*, *loading*, and *training*. Given a set of target vertices, the sampling phase randomly selects a subgraph from the k-hop neighborhood of the target vertices. A mini-batch with two target vertices is shown in Figure 1(a). In the loading phase, the input features of the vertices in the bottom layer of the mini-batch are loaded into the GPUs. During 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 2). The last GNN layer computes

the features of the target vertices, which are then used to calculate the loss. During backward propagation, the layers are executed in reverse order to compute gradients. Finally, all GPUs aggregate and apply the computed gradients.

Redundant loading and computation Data parallelism is the most commonly used training strategy for mini-batch GNN training. In data parallel training, the target vertices are partitioned among GPUs, where each partition corresponds to a separate *micro-batch* (see Figure 1(a)). Each GPU independently samples the neighborhood of the target vertices, loads the input features of all the vertices in the bottom layer of its micro-batch, and computes the hidden representation for the sampled nodes. This approach has a high degree of sampling, data loading, and computational redundancy. For example, in Figure 1(a), vertex c is sampled in two micro-batches, the features of its neighbors are loaded by two GPUs, and its hidden representation is computed redundantly.

Table 1 further reports the degree of computational and data loading redundancy in data-parallel training. With 4 GPUs, data parallelism creates 4 separate micro-batches ("Micro"), causing up to $1.2\times$ compute and $2.5\times$ feature loading compared to having only a single mini-batch ("Mini").

Graph	# Edges Computed			# Features Loaded		
	Micro	Mini	Ratio	Micro	Mini	Ratio
Orkut	926M	751M	1.2x	422M	169M	2.5x
Papers100M	452M	389M	1.2x	231M	154M	1.5x
Friendster	13.4B	13.1B	1.0x	11.4B	9.4B	1.2x

Table 1. Redundant computation and data loading. The total number of edges computed and feature data loaded over one epoch when each mini-batch is sampled as 4 micro-batches of size 1024 (Micro) vs. 1 mini-batch of size 4096 (Mini).

Limitations of Existing Optimizations. The P3 system introduced a hybrid parallelism approach called *push-pull parallelism* to partially address the redundancy problem of data parallelism (Gandhi & Iyer, 2021). It avoids transferring input features among hosts by proposing an alternative to data parallelism called *push-pull parallelism*. The feature data for each vertex is partitioned across multiple GPUs. As shown in Figure 1(b), each GPU computes the first layer for *all* the micro-batches on its local feature slice. Then each GPU exchanges partial activations and continues the iteration for the remaining layers in a data-parallel fashion. Figure 3 shows that push-pull parallelism (described in Section 6) reduces data loading cost, but it also introduces an expensive shuffle that increases the overall training time. This shuffle is expensive because $P3*$ still uses data parallelism for the upper layers.

Previous work also explored using GPU caching to reduce data-loading overhead in the context of data parallel training (Kaler et al., 2023; Lin et al., 2020; Tan et al., 2023; Yang et al., 2022a;b). These systems populate a static cache offline with frequently accessed input features. Some systems use a distributed shared memory to enable GPUs to fetch features from other GPUs' memory using fast GPU-to-GPU interconnects like NVLink (Cai et al., 2023; Song et al., 2023a; Tan et al., 2023). As shown in Figure 3, the distributed shared-memory caching mechanism in Quiver (Tan et al., 2023) can reduce loading time for the Orkut graph, especially since its whose feature data can be fully cached across multiple GPUs.

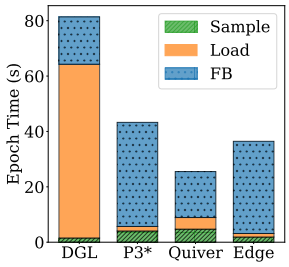

Figure 3. Breakdown of epoch time into sampling, loading, and training time for DGL, P3*, Quiver, and Edge (naive implementation of split parallelism) on Orkut with the GAT model.

Challenges of avoiding redundancy. Despite these improvements, none of the above approaches avoids the computational and loading redundancies of mini-batch GNN training. In this work, we propose using split parallelism to avoid these redundancies by splitting each mini-batch sample on the fly at each iteration. Splitting maps work associated with a vertex to only one GPU and shuffles vertex features across GPUs (see Figure 1(c)).

Splitting is on the critical path, so it is necessary to minimize its overhead. Splitting must also minimize communication costs during shuffles and balance the load across GPUs. A naïve approach would be to run a min-edge-cut graph partitioning algorithm on each mini-batch we sample. However, this would be too computationally expensive since splitting must be executed during the sampling step of each iteration, and it is hard to parallelize across multiple GPUs. A more practical approach would be to use graph partitions computed offline using a min-edge-cut graph partitioner like Metis (Karypis & Kumar, 1997). We evaluate this approach in Figure 3 and call it Edge, and find that it is not sufficient to outperform data parallelism since the partitioning is done on the entire graph, not on the specific mini-batch that is being split. Edge performs worse than Quiver as the overhead of shuffling during training and load imbalance exceeds the savings from eliminating redundancy. This observation motivates us to design an optimal split assignment (Section 4) that addresses these challenges.

3 GSplit: Training pipeline

To eliminate redundant work *and* reduce training time, we introduce a hybrid parallelism approach called *split parallelism*. Instead of sampling micro-batches with overlapping input and hidden vertices, one micro-batch for each GPU,

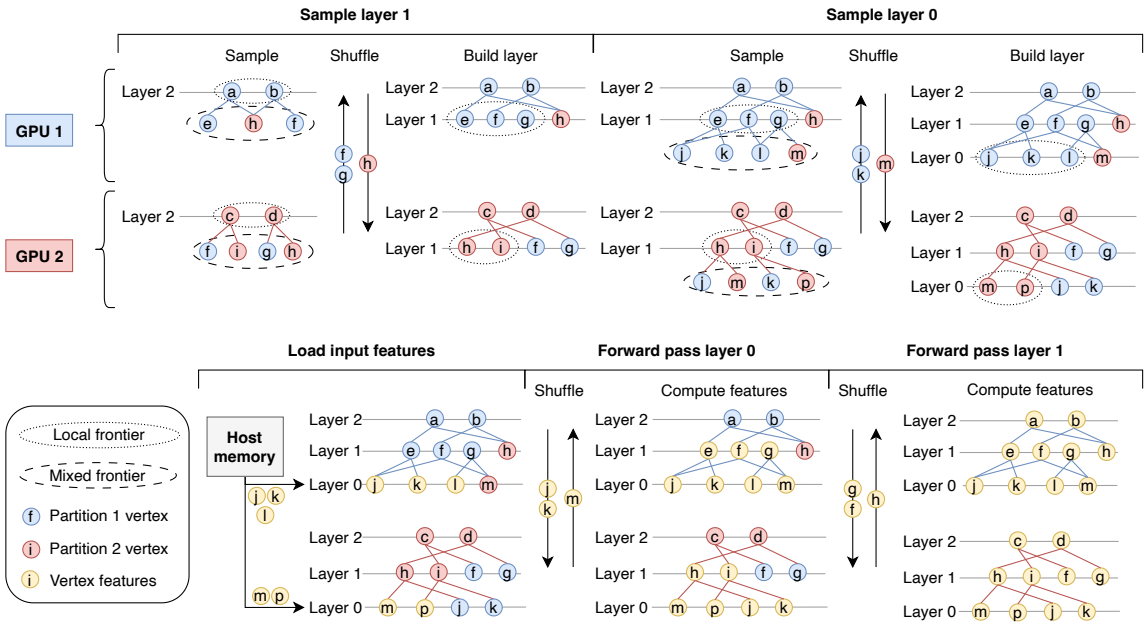

Figure 4. Overview of the GSplit training pipeline.

the sampling phase now produces non-overlapping sets of vertices called *splits*, each assigned to a different GPU. The key to effective split parallelism is that the splits are obtained by a lightweight online *splitting algorithm*, which provides probabilistic performance guarantees described in Section 4, while sampling each mini-batch at each iteration. Each GPU loads only the features of the input vertices within its assigned splits, taking into account each GPU's local cache. During training, only one GPU computes the hidden features of any given vertex and then shuffles the computed feature to other GPUs.

Example of a GSplit iteration We present a running example to describe one training iteration in GSplit (see Figure 4) and introduce GSplit's API, such as the local and mixed frontier and the split index that supports existing efficient single-GPU kernels for sampling and training. Although GSplit can be combined with distributed GPU caching, the example assumes no caching to simplify the description.

The first phase of each iteration is sampling. GSplit pushes sampling to the GPU for performance reasons, in line with recent work (Gong et al., 2023; Jangda et al., 2021; Pandey et al., 2020; Wang et al., 2021). Sampling proceeds layer-by-layer as each GPU samples its local split of the same mini-batch, rather than a separate micro-batch. At each layer, a GPU starts from a set of vertices called the *local frontier*, samples its neighbors, and obtains what we call the *mixed frontier*. Unlike the local frontier, a mixed frontier can include remote vertices. For example, in Figure 4, GPU 1 starts the iteration with the layer-2 local frontier $\{a, b\}$ and samples the mixed frontier $\{e, h, f\}$, which includes the remote vertex h. Similarly, GPU 2 starts from the local frontier $\{c, d\}$ and samples the mixed frontier $\{f, g, h, i\}$, which includes the two remote vertices f and g. Each GPU uses the splitting algorithm to separate the local and remote vertices and then shuffles the remote vertices to their partition. GPUs then build the local frontier for the next layer based on the vertices they receive. For example, the new local frontier of GPU 1 is $\{e, f, g\}$, where $\{g\}$ was sampled by GPU 2. Analogously, the new local frontier of GPU 2 at layer-1 is $\{h, i\}$. The next layer is the union of the new local frontier and the previous mixed frontier: $\{e, f, g, h\}$ for GPU 1 and $\{h, i, f, g\}$ for GPU 2. Sampling for layer-1 is carried out in a similar way from its local frontier. The splitting step creates an auxiliary data structure, called the *shuffle index*, that consists of gather and scatter indexes to efficiently send and receive sparse vertex data during the shuffle rounds at each layer.

After sampling finishes, the loading step loads the input vertex features from the host memory into the GPUs. When split parallelism is combined with caching, a GPU can skip loading an input feature if it is already cached locally. Unlike data-parallel systems that could load redundant input features, split parallelism eliminates the redundant CPU-GPU data loading as splits do not have overlapping vertices. In the example of Figure 4, we consider no caching, so GPU 1 loads the input features of the vertices in layer-0 local frontier $\{j, k, l\}$ from the host memory. Similarly, GPU 2 loads the input features of the vertices $\{m, p\}$.

The training phase begins after loading and proceeds bottom-up. At each layer, each GPU is responsible for computing the hidden features of the vertices in its local frontier by aggregating the features of the neighbors in the layer below.

These neighbor vertices are the same vertices that were in the mixed frontier for that layer during sampling, so we can reuse the shuffle index generated at that step. In the example, GPU 1 must compute the layer-1 hidden features of vertices $\{e, f, g\}$. To do that, it needs features of vertices $\{j, k, l, m\}$, which constitute the mixed frontier of layer-0 constructed during sampling and include the remote vertex h. GPU 1 sends the vertex features in its local frontier $\{j, k\}$ to GPU 2 and receives the features of the remote vertex $\{m\}$ by performing an all-to-all shuffle. The shuffle rebuilds the mixed frontier, which now holds all the hidden features required for the computation of all the vertices in the local frontier of layer-1 $\{e, f, g\}$. GPU 1 proceeds to construct the mixed frontier of layer-1 by sending $\{g, f\}$ and receiving $\{h\}$ by performing an all-to-all shuffle. Finally, GPU-1 uses the mixed frontier $\{e, f, g, h\}$ of layer-1 to compute the hidden features of the target vertices $\{a, b\}$. The backward pass works in a similar way, moving from the top layer down to the bottom, but in the opposite direction. Importantly, the data follows the same paths as during the sampling step, so we can reuse the shuffle index.

4 THE SPLITTING ALGORITHM

Online splitting is challenging because it must be efficient, balance load across splits, and minimize communication cost, as discussed in Section 2. GSplit uses an embarrassingly parallel online splitting algorithm that maps each vertex to a split independently of each other in constant time. It does so by providing *probabilistic* guarantees, rather than deterministic ones: given a random mini-batch sample, it minimizes the *expected* communication cost while balancing the *expected* load per split. Formally, the splitting algorithm solves the following problem:

Problem definition. (Mini-batch splitting problem) *Let $M(V_M, E_M)$ be a sampled mini-batch of a graph $G(V_G, E_G)$, S a set of splits, and $f_M : V_M \rightarrow S$ a splitting function that assigns each vertex in V_M to a split. Let S_i be the set of vertices assigned to split i by f_M, X_i a random variable expressing the number of vertices at layer $l > 0$ in S_i, Y a random variable expressing the number of edges in E_M having endpoints in two different splits, and $\epsilon \geq 0$ a tunable constant. The mini-batch splitting problem involves finding the splitting function f_M that solves the following minimization problem:*

$$\begin{aligned} \min_{f_M} \quad & \mathbb{E}[Y] \\ \text{s.t.} \quad & \forall i : \mathbb{E}[X_i] \leq (1+\epsilon) \cdot \sum_{i \in [1,|S|]} \mathbb{E}[X_i]/|S|. \end{aligned} \tag{1}$$

The problem is to find a *splitting function* f_M that can be used online by the splitting algorithm to map each sampled vertex to a split, each corresponding to a different GPU device. The random variables X_i and Y represent the computation and communication cost of the splits, respectively. When the algorithm assigns a vertex at layer $l > 0$ to a split, the corresponding GPU must sample its neighbors during the sampling phase and compute its hidden feature during the training phase. Edges connecting vertices in different splits induce communication costs during the shuffle phases of both sampling and training. This problem is NP-hard since it can be reduced to min-edge-cut graph partitioning by selecting an appropriate sampling function.

Splitting algorithm To avoid solving this NP-hard problem online, our approach is to reduce it to a problem that can be solved offline. Therefore, we propose using a splitting algorithm that has an offline and an online part. Offline, the algorithm finds a *global partitioning function* $f_G : V_G \rightarrow D$ that statically maps each vertex in the whole input graph to a GPU device. Then, online, the splitting algorithm uses f_G as a substitute for f_M to map each vertex to a device and thus to its corresponding split at each iteration. Online splitting is embarrassingly parallel since f_G maps each vertex to a split independently and does it in constant time. GSplit uses the global partitioning f_G also to determine the GPU where it statically caches input features of a vertex. This ensures that the caches are consistent with the splits.

We now describe the details of the offline part of the splitting algorithm, which finds the global partitioning function. The first stage of the offline algorithm is pre-sampling, which weighs the vertices and edges of the input graph. Weights represent the computational and communication costs incurred by GPUs during split-parallel sampling and training. The second stage uses a weighted min-edge-cut graph partitioning algorithm to find the global partitioning function.

Given an input graph G, the pre-sampling stage assigns weights to the vertices and edges of G to obtain a weighted graph G_w. It runs the same sampling algorithm used during training for a fixed number of epochs. At each iteration, the algorithm samples the k-hop neighborhood of the training (target) vertices in the mini-batch. It then assigns to each vertex v a weight $w_V(v) = k_v/N$, where k_v is the total number of times v is sampled at a layer $l > 0$ across all samples and N is the number of samples. The weight of each edge e is $w_E(e) = k_e/N$, where k_e is the total number of times e is sampled across all samples.

After completing the pre-sampling stage and obtaining the weighted graph G_w, the offline algorithm runs a weighted min-edge-cut graph partitioning algorithm on G_w to obtain partitions. The algorithm outputs partitions that minimize the sum of the weights of the edges in the cut and ensure that the load per partition, which is the sum of the weights of its vertices, is balanced. Formally, given a graph $G_w(V, E, w_V, w_E)$ and the number of partitions $d = |D|$, where D is the set of GPU devices, a weighted min-edge-cut graph partition algorithm outputs the set of partitions

$P = \{P_1, \ldots, P_d\}$ of V that solves the following problem:

$$\begin{aligned} \min_{P} \quad & \sum_{e \in C} w_E(e) \\ \text{s.t.} \quad & \forall i: \; L_i \leq (1+\epsilon) \cdot L/d \end{aligned} \tag{2}$$

where $w_E(e) = k_e/N$ is the edge weight of e, $C = \{\langle u, v\rangle \in E : u \in P_i, v \in P_j, i \neq j\}$ is the edge cut set, $L_i = \sum_{v \in P_i} w_V(v) = \sum_{v \in P_i} k_v/N$ is the load of each partition P_i, $L = \sum_{v \in V} w_V(v)$ is the total load across all partitions, and $\epsilon \geq 0$ is a tunable constant. Note that since the minimization problem of Eq. 2 is NP-hard, we use heuristics to solve it in practice, for example, using Metis (Karypis & Kumar, 1997). The partitions P found by the graph partitioning algorithm determine the global partitioning function f_G from vertices to GPUs.

Analysis We show that our splitting algorithm finds a solution to the mini-batch splitting problem of Eq. 1 by reducing it to the optimization problem of Eq. 2 with $d = |S|$, which we can solve with a heuristic. We show that $\mathbb{E}[X_i] = L_i$ and $\mathbb{E}[Y] = \sum_{e \in C} k_e/N$. After doing that, we can conclude that Eq. 2 minimizes $\mathbb{E}[Y]$ and constraints $\mathbb{E}[X_i]$ as in Eq. 1.

We start by showing that $\mathbb{E}[X_i] = L_i$. Given a vertex $v \in V$, let the random variable Z_v be the number of layers $l > 0$ where the vertex appears in the sample S. We have that $X_i = \sum_{v \in P_i} Z_v$, which implies that $\mathbb{E}[X_i] = \sum_{v \in P_i} \mathbb{E}[Z_v]$. If we assign vertex weights using a sufficiently large number of samples N, according to the law of large numbers, we have $\mathbb{E}[Z_v] = k_v/N$, which implies that $\mathbb{E}[X_i] = L_i$.

We use a similar argument to show that $\mathbb{E}[Y] = \sum_{e \in C} k_e/N$. Given an edge $e \in E$, let the random variable Z_e be the number of times e is sampled (at different layers) in S. The set of cross-split edges in the sample S is a subset of the cross-partition edges C in the whole graph, so $Y = \sum_{e \in C} Z_e$. The expected value of Y is given by $\mathbb{E}[Y] = \sum_{e \in C} \mathbb{E}[Z_e]$. If we use a sufficiently large number of samples N to assign edge weights, according to the law of large numbers, we have that $\mathbb{E}[Z_e] = k_e/N$, so $\mathbb{E}[Y] = \sum_{e \in C} k_e/N$.

5 GSplit's API

GSplit offers a *layer-centric* programming API that simplifies the development of GNN sampling and training code and kernels by hiding cross-GPU coordination. Besides simplicity, this approach supports the reuse of optimized single-GPU kernels for GNN sampling and training proposed by recent research (Fan et al., 2023; Fu et al., 2022; Gong et al., 2023; Jangda et al., 2021; Wang et al., 2023c; Wu et al., 2021; Ye et al., 2023; Zhou et al., 2023).

Motivation for a layer-centric API Single-GPU kernels used by data-parallel systems are layer-centric: they assume that at each GNN layer, the source vertex, destination vertex, and edge features for all incoming edges incident to the same destination vertex are locally available. GSplit's API supports this property even after it splits destination vertices of the same mini-batch among different GPUs. The alternative hybrid mini-batch parallelism approach, which is P^3, breaks this assumption to offer an *edge-centric* API to enable pushing part of the computation of each micro-batch to multiple hosts and GPUs in a fine-grained manner. For example, implementing state-of-the-art models such as Graph Attention Networks (GAT) requires a custom implementation to ensure correctness (Gandhi & Iyer, 2021).

Algorithm 1 Split-parallel sampling.

```
def sp_sample(local_targets):
   local_front[L] = local_targets
   for l in [L-1, 0]:
      mixed_front, edges[l] =
      sample_layer(local_front[l+1])
      local_front[l], edges[l],
      shuffle_idx[l] = split(mixed_front,
      edges[l])
   return edges, shuffle_idx
```

Sampling Algorithm 1 shows the pseudocode of sampling using GSplit's API.

The `sp_sample` function executed by each GPU takes as input the subset of target vertices in the mini-batch that are local to the GPU according to the splitting algorithm (`local_targets`) and outputs a *split*, which consists of the set of `edges` at each layer that are used by the local training kernels to aggregate features. It also outputs a *shuffle index* (`shuffle_idx`), which consists of indices to efficiently send and receive vertex data in the frontiers during the sparse all-to-all shuffling at each layer.

The `sample_layer` function outputs a new mixed frontier (`mixed_front`) and the edges from the input local frontier to the output mixed frontier (`edges[l]`). The mixed frontier includes both local and remote vertices. The sampling code must invoke the `split` function to perform a shuffle and obtain a local frontier (Line 1).

The benefit of using this layer-wise sampling API is that we can directly use the sampling implementations in DGL (Wang et al., 2020) as well as other GPU sampling kernels (Gong et al., 2023; Jangda et al., 2021; Pandey et al., 2020; Wang et al., 2021) that sample graph layer by layer.

Training We now explain how to implement cooperative split-parallel training. Algorithm 2 shows the pseudocode for the forward propagation. The backward propagation works similarly, except that the computation happens from the top layer to the bottom layer.

The `sp_forward` function takes as input the GNN model (`model`) the input features of the vertices in the local split

Algorithm 2 Forward pass of split-parallel training.

```
def sp_forward(model, feat, edges,
shuffle_idx):
   hidden = feat
    for l in [L-1, 0]:
      gnn_layer = model.layer(l)
      mixed_hidden = shuffle(shuffle_idx
      [l], local_hidden)
      local_hidden = gnn_layer(edges[l],
      mixed_hidden)
   return hidden;
```

(`feat`), the structure of the split (`edges`), and the shuffle index (`shuffle_idx`) required for each layer. The latter two inputs are produced by the sampling code of Algorithm 1. It outputs the hidden features of the target vertices.

At each layer l, each GPU starts by shuffling the features/activations of its local vertices to other GPUs using the `shuffle` function (Line 2). The output of this function is the `mixed_hidden` tensor, which contains all the features required to compute the hidden features at layer $l+1$, including the features of remote vertices. The GNN layer then computes the next-layer hidden features for the vertices in the local partition of the GPU (Line 2).

During the backward pass, each GPU computes the gradients for both local and remote vertices at each layer l. The shuffle index (`shuffle_idx`) is then used to push gradients of remote vertices back to the GPU, storing the vertices in its local partition. The backward pass proceeds for each layer, reversing the direction of communication in the forward pass. Our abstraction thus ensures that gradients are computed accurately across layers when using split parallelism between GPUs and is agnostic to the specific GNN model being used.

Similar to the case of sampling, using a layer-wise API for training allows us to reuse the optimized single-GPU training kernels (Fan et al., 2023; Fu et al., 2022; Wu et al., 2021; Ye et al., 2023; Zhou et al., 2023) in `gnn_layer`.

6 Evaluation

In this section, we evaluate GSplit by answering the following questions: What are the end-to-end speedups that can be achieved by GSplit relative to the baselines (§ 6.2)? What is the impact of using GSplit's splitting algorithm, which provides probabilistic performance guarantees (§ 6.3)? How does GSplit scale to a larger number of GPUs within one host and across hosts (§ 6.4)? How does performance vary when we vary the hyperparameters (§ 6.5)? How does GSplit's split parallelism impact accuracy (§ 6.6)?

6.1 Experiment Settings

GNN models and datasets We consider two popular and diverse GNN models: GraphSage (Hamilton et al., 2017) and GAT (Veličković et al., 2018). We use the standard neighborhood sampling algorithm. Its low computational complexity makes it less likely to hide the cost of shuffling during split-parallel sampling. By default, we use a sampling fanout of 15, 3 GNN layers, a default hidden size of 256 as used in (Hamilton et al., 2017), and a batch size of 1024. GSplit's split-parallel implementations use the same sampling and training kernels as DGL's data-parallel one.

Dataset	# Nodes	# Edges	# Feat	# Type
Orkut	3.1M	120M	512	Undirected
Papers100M	111M	1.6B	128	Directed
Friendster	65M	1.9B	128	Undirected

Table 2. Datasets used for the evaluation

We use three large datasets listed in Table 2. The Paper100M dataset, derived from the directed citation graph, is the largest homogeneous dataset from the Open Graph Benchmark (OGB), a standard benchmark for GNN training (Hu et al., 2021). Additionally, we utilize two other large synthetic graphs from the SNAP repository (Leskovec & Krevl, 2014), consisting of undirected social networks that are frequently used for evaluating GNN training performance.

Hardware setup By default, our experiments use an AWS EC2 p3.8xlarge instance with 4 NVIDIA V100 GPUs (16GB memory) and Xeon E5-2686 v4 @ 2.70GHz, with 32 CPU cores and 244 GB RAM. GPUs are connected to the CPU with a PCIe 3.0 16 bus and to each other via NVLink. For experiments with 8 GPUs, we use a similar p3.16xlarge instance having 64 CPU cores and 488 GB of RAM.

Baseline systems We consider the following systems as baselines. All systems perform synchronous training to avoid biasing model accuracy and use GPU-based sampling.

- **DGL** is a standard production library for data-parallel GNN training (Wang et al., 2020). We use DGL version 1.1.3, the same one we use as a component of GSplit. DGL only supports caching input features and the graph topology when they fully fit into one GPU.
- **Quiver** is a recent data-parallel GNN training system that uses distributed caches and leverages fast direct GPU-GPU buses like NVLink (Tan et al., 2023). We use version 0.1.1. Quiver supports distributed and partial caching across multiple GPUs.
- $\boldsymbol{P^3}$ is a distributed GNN training system that uses hybrid push-pull parallelism. Its source code is not publicly available, so we adapt the push-pull parallelism approach to a single-host multi-GPU system and refer to our implementation as **P3***.
- **Edge** is a variant of GSplit used to investigate the impact of using a naïve offline splitting algorithm that

Graph	System	GraphSAGE					GAT				
		S	L	FB	Total(s)	Speedup	S	L	FB	Total(s)	Speedup
Orkut	DGL	1.5	62.7	9.2	73.4	4.4×	1.5	62.8	17.1	81.4	3.6×
	P3*	4.0	1.5	8.5	14.1	0.8×	4.0	1.7	37.6	43.3	1.9×
	Quiver	4.9	4.3	8.7	17.8	1.1×	4.7	4.2	16.4	25.5	1.1×
	Edge	1.9	1.3	25.1	28.3	1.7×	1.9	1.3	33.3	36.5	1.6×
	GSplit	1.9	0.1	14.8	16.7		1.9	0.1	20.5	22.5	
Papers100M	DGL	4.6	9.5	11.3	25.4	1.4×	4.7	9.0	31.7	45.4	1.2×
	P3*	3.3	11.5	25.8	40.6	2.2×	3.3	11.3	65.7	80.4	2.2×
	Quiver	11.8	10.4	11.7	34.7	1.9×	11.0	11.0	30.5	53.5	1.4×
	Edge	11.7	0.1	16.3	28.1	1.5×	11.5	0.1	40.1	51.7	1.4×
	GSplit	3.9	2.6	11.8	18.3		3.8	2.3	31.1	37.2	
Friendster	DGL	62.7	283.4	61.1	407.2	2.9×	62.6	284.8	245.9	593.3	1.7×
	P3*	85.9	350.8	151.5	588.1	4.1×	76.5	351.4	613.8	1041	3.0×
	Quiver	132.5	24.9	63.6	223.9	1.6×	135.5	24.8	243.5	404.2	1.2×
	Edge	65.7	1.0	121.8	188.5	1.3×	106.1	0.7	368.2	475.0	1.4×
	GSplit	41.2	2.2	98.9	142.3		62.1	2.2	283.5	347.8	

Table 3. Epoch time (in seconds). S = Sampling, L = Loading, FB = Forward and backward pass. The speedups are the total epoch time of other systems relative to GSplit.

does not weigh vertices and edges using pre-sampling (see Section 6.3). It uses min-cut partitioning and balances the number of edges and target vertices in each partition, as commonly done in data-parallel GNN training systems (Zheng et al., 2020), while minimizing the number of edges across partitions.

We configure all systems to maximize the memory available for caching the graph structure and input features while allocating sufficient memory to sample and train without going out of memory. We configure Quiver and GSplit to use the same sampling frequency criterion to rank the input features to cache as proposed in (Yang et al., 2022b). P3* cannot cache input features for only a subset of the vertices, so it only uses caching for the Orkut graph.

6.2 End-to-End Performance

We now compare the performance of GSplit with existing work: DGL, Quiver, and P3*. The comparison is based on epoch time only, as none of these systems biases the GNN model accuracy they train. We measure the total epoch time and break it down for the three steps of the mini-batch training iterations: sampling a subgraph, which also includes splitting for GSplit, loading the input features into each GPU, and performing the forward and backward pass.

We report the results in Table 3. Overall, GSplit outperforms DGL by up to 4.4x (2.5x on average), P3* by up to 4.1x (2.4x on average), and Quiver by up to 1.9x (1.4x on average) by eliminating redundant loading and computation. By avoiding redundant computation, GSplit can reduce its sampling and training costs, mitigate the additional cost of shuffling, and in some cases be even faster than some data-parallel systems in those steps.

Sampling time comparison The sampling step in GSplit entails not only sampling the mini-batch, as in the other systems, but also splitting the mini-batch, constructing the shuffle indexes, and shuffling vertices. The evaluation shows that these additional costs are balanced, and sometimes offset, by the elimination of redundant work and by the use of distributed caching for the graph structure. GSplit's online splitting is not a performance bottleneck because it is embarrassingly parallel and fast.

Loading time comparison Compared to data parallel systems GSplit reduces the input feature loading time by reducing redundant loads. P3* and Quiver perform much better than DGL in the Orkut graph because its input features can be fully cached across GPUs. The input features for the Papers100M and Friendster graphs cannot be fully cached, even when using a distributed cache. Quiver has lower loading times than DGL and P3* for Friendster because it supports caching only a subset of the features in the GPUs, but it cannot leverage its cache effectively for Papers100M because of the high cost of loading feature cache misses from the host memory.

Forward/backward pass time comparison In contrast to data-parallel systems, hybrid-parallel approaches such as P3* and GSplit introduce communication overhead in the form of shuffles during training, which generally leads to increased forward-backward (FB) times. When training the GraphSage model, P3* partially offsets the overhead of push-pull shuffling by distributing some computation of each micro-batch across all GPUs. This optimization allows P3* to achieve the shortest training time among all systems on the Orkut graph. However, in all other scenarios, P3* exhibits the longest FB times due to the need to shuffle all partial activations across micro-batches, a cost that becomes particularly pronounced with complex models like GAT, which generate larger intermediate activations.

GSplit addresses the shuffling overhead by eliminating redundant computation of hidden features, thereby improving

overall efficiency. Nonetheless, its FB times remain higher than those of data-parallel systems. This gap is less pronounced for computationally intensive models such as GAT, where the benefits of avoiding redundant computation are more significant. Compared to P3*, GSplit consistently achieves lower training times across all cases except for the Orkut-GraphSage setting, attributed to its reduced shuffle costs and more efficient use of computation.

6.3 Evaluation of the Splitting Algorithm

End-to-end impact of probabilistic splitting GSplit relies on an offline graph partitioning algorithm to provide probabilistic performance guarantees: balancing the expected load across splits and minimizing the expected communication costs across partitions. We evaluate the impact of using this algorithm by combining GSplit's online splitting with three alternative offline partitioning algorithms that do not provide these guarantees.

The *GSplit* is the pre-sampling-based algorithm with probabilistic guarantees described in Section 4. The *Node* algorithm partitions the graph using only the pre-sampled node weights. Comparing it to *GSplit* shows the impact of using edge weights during graph partitioning. The *Edge* algorithm uses min-cut partitioning, but it does not assign weights to vertices and edges using pre-sampling. It balances the number of edges and target vertices in each partition, as commonly done in data-parallel GNN training systems (Zheng et al., 2020) while minimizing the number of edge cuts across partitions. Finally, *Rand* partitioning algorithm randomly assigns each vertex to a partition.

Table 3 shows the end-to-end performance benefit of using GSplit's splitting algorithm compared to the Edge baseline. GSplit helps improve the end-to-end training performance by up to 1.5× on Orkut, 1.7× on Papers100M, and 1.4× on Friendster. We analyze the reasons for these speedups

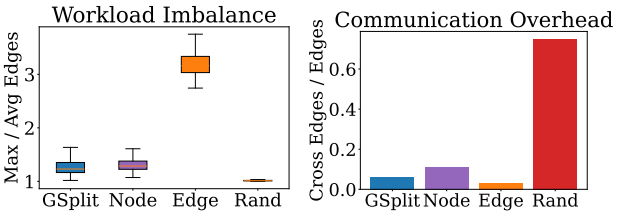

Figure 5. GSplit vs. other offline partitioning algorithms.

more in-depth in Figure 5, which compares the workload imbalance and communication costs of different partition strategies using the Papers100M graph. We quantify the *workload imbalance* among the splits in each iteration as the maximum number of edges at layer $l > 0$ per split divided by the average, and the *communication cost* as the percentage of cross-edges among splits over all the edges in the mini-batch.

As shown in Figure 5, the *Rand* baseline leads to the most evenly distributed computation cost across partitions. Yet, it results in a high communication overhead with 75% of the edges crossing two partitions in most iterations. The *Edge* baseline achieves a much lower edge cut and reduces the communication overhead. However, balancing the target vertices alone does not guarantee that the splits of the sampled mini-batches will be balanced.

The splitting algorithm of GSplit achieves the benefit of both approaches. It has a lower communication overhead compared to the random partition algorithm and a more balanced workload than simply balancing the number of target vertices in each partition. This is thanks to its offline pre-sampling approach.

In addition, we observe that assigning weights to edges using GSplit effectively reduces the communication overhead in a mini-batch. Compared to Node, which does not weigh edges, the average ratio of cross edges over total edges is reduced from 9% to 5% for Papers100M as shown in Figure 5. Better yet, the reduction in communication costs does not significantly impact workload imbalance. We observe a similar trend in Orkut and Friendster.

Offline pre-processing costs. The splitting algorithm has two offline steps: pre-sampling and graph partitioning. Empirically, we found that running ten epochs of sampling during the pre-sampling stage is sufficient. Using a larger number of sampling epochs has little impact on load balancing and communication costs. When using 30 and 100 pre-sampling epochs, the difference in average load imbalance per mini-batch remains within 2% for all the graphs, while the percentage of cross edges over the total number of edges per mini-batch remains within 7% for Orkut, 2% for Papers100M, and Friendster. Pre-sampling is fast relative to the overall training time. The pre-sampling time is 19s for Orkut, 20s for Papers100M, and 288s for Friendster when using a machine with four RTX 3090 GPUs.

The final offline step of GSplit is graph partitioning, which is commonly used in many distributed mini-batch GNN training systems. We use METIS (LaSalle & Karypis, 2013) to partition the graphs on an AWS r7a.x24large instance, which has 48 cores (96 threads) and 768GB of memory. The partitioning time is 14s for Orkut, 78s for Papers100M, and 534s for Friendster. Both pre-sampling and partitioning are one-time costs that can be amortized by training over the same dataset multiple times.

6.4 Scalablity

We now evaluate the performance of GSplit against baseline systems across different configurations of GPUs and hosts.

Single-host. We first evaluate using a single host and varying the number of GPUs in Figure 6(a). GSplit scales better

than the other systems with a larger number of GPUs because it has more opportunities to avoid redundant loads and computation. It can also make more efficient use of the GPU caches thanks to its use of collective communication primitives. Quiver relies on direct remote memory access to transfer cached input features across GPUs efficiently. This, however, is only possible between GPUs that have direct NVLink connections. In our 8-GPU host, not all GPUs are directly connected with each other. Quiver circumvents this problem by replicating cached features across GPUs that have no direct links. GSplit, by contrast, does not need to cache features redundantly.

Multi-hosts. We also run distributed multi-host experiments where each host has 4 GPUs and show the results in Figure 6(b). GSplit can be implemented across multiple hosts by partitioning the graph and node structure across hosts. Its shuffle function can operate across a diverse set of interconnects as it completely abstracts data collection into contiguous memory and utilizes low-level libraries such as NCCL for communication. As NCCL utilizes the best available interconnect, GSplit can transparently use high-speed interconnects such as the fifth-generation NVLink Switch technology. However, in the absence of high-speed interconnects across hosts, as in our evaluation setting, GSplit uses a hybrid approach to scale to multiple hosts, using data parallelism across hosts and split parallelism within each host. We observe that GSplit shows consistent speedups in all configurations and models.

6.5 Ablation Study

We now estimate how consistent GSplit's speedups are when the model and training hyperparameters change. The results for the Friendster graph are reported in Figure 6.

Hidden size. Increasing the hidden feature size impacts the FB time of GSplit negatively, increasing the overall volume of data shuffled. However, it also increases the gains of avoiding redundant computation, especially for complex models such as GAT. The two factors balance out and GSplit shows consistent speedups over the baselines, as shown in Figure 6(c).

Batch size. We vary the mini-batch size while keeping a hidden size of 128 to avoid going out of memory. Larger mini-batches increase the relative cost of shuffling during the FB phase but also offer more opportunities to save on redundant data loading. Overall, GSplit always outperforms the data-parallel baselines, as shown in Figure 6(d).

Number of GNN Layers. In this experiment, we use a hidden size of 128 and pick the largest sampling fanout that avoids going out of memory for each number of layers. The results are reported in Figure 6(e). The GNNs are most commonly trained with 2 or 3 layers, i.e., 2 or 3 hops from the target vertices. GSplit consistently outperforms the baselines in these settings. Adding more layers in split parallelism increases the number of shuffles but also increases redundancy across minibatches.

6.6 Accuracy

GSplit is a general GNN training system that introduces no system-level algorithmic bias that could affect training accuracy. To validate this, we compare the test accuracy after each training epoch on the Papers100M dataset using the GraphSAGE and GAT models, trained with GSplit and the DGL data-parallel baseline. Figure 7 shows that GSplit closely mirrors the baseline's accuracy across all epochs for both models, as expected. These curves confirm the correctness of our split/shuffle abstraction, which automatically computes gradients across splits during the backward pass in a model-agnostic way.

7 Related Work

Mini-batch training systems DistDGL (Zheng et al., 2020) and AliGraph (Zhu et al., 2019) use data-parallel mini-batch training to scale to large graphs. ByteGNN optimizes distributed sampling for CPU-based data-parallel training (Zheng et al., 2022). Another research direction is taken by Marius++, which runs data-parallel GNN training on large graphs using a single GPU and an out-of-core approach rather than a distributed system (Waleffe et al., 2022). Pipelining multiple batches is another orthogonal technique to increase resource utilization and reduce training time (Cai et al., 2023; Kaler et al., 2022; Min et al., 2021).

Prior work has focused on efficient single-GPU sampling with high-level programming APIs (Gong et al., 2023; Jangda et al., 2021; Pandey et al., 2020; Tripathy et al., 2024; Wang et al., 2021). GNNLab reduces the communication costs in distributed sampling using a factorized approach, where sampler GPUs cache the entire graph topology while other trainer GPUs only cache input features (Yang et al., 2022b). Other systems propose caching both topological and feature data on each GPU (Cai et al., 2023; Sun et al., 2023; Zhang et al., 2023b). DSP supports distributed sampling on a partitioned graph structure, but it performs two all-to-all shuffles for each sampled layer (Cai et al., 2023). Split-parallel sampling requires only one all-to-all shuffle per sampled layer, since training is split-parallel. Split parallelism and cooperative training were introduced in a preliminary version of this paper (Polisetty et al., 2023). Dedicated sampling algorithms can increase feature access locality (Balin et al., 2023). APT adaptively chooses between data, push-pull, and split parallelism, but it does not consider probabilistic splitting (Ma et al., 2025).

Sampling algorithms In our work, we consider general systems that support arbitrary sampling algorithms rather than imposing a specific algorithm, with its specific performance

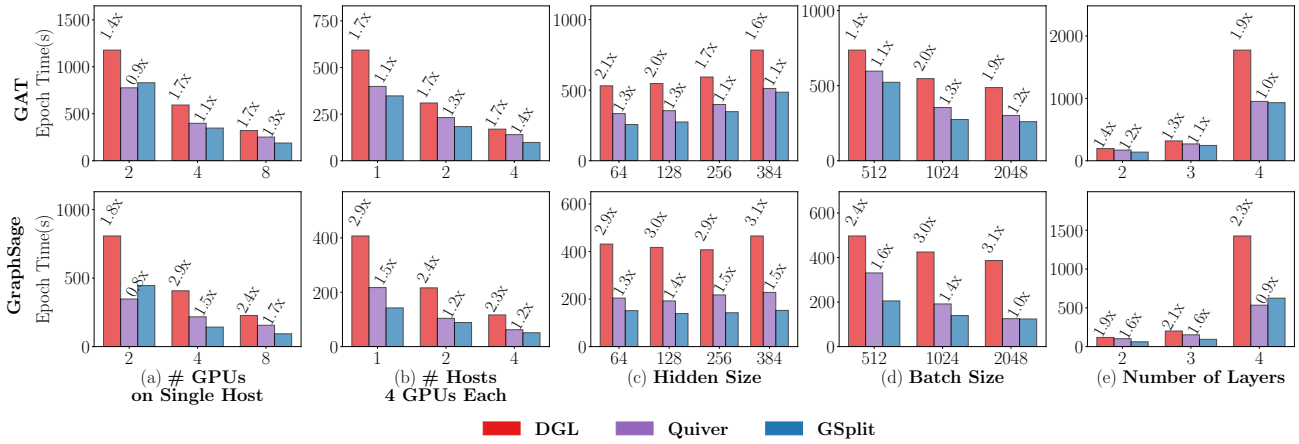

Figure 6. Scalability and ablation study. The reported speedups are the epoch time of other systems relative to GSplit.

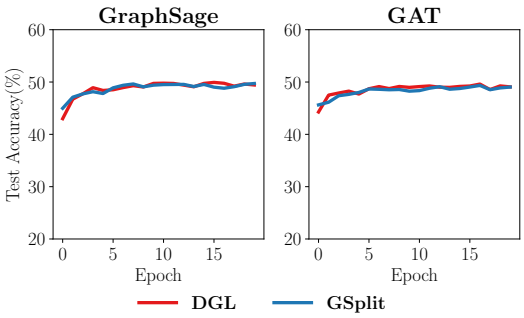

Figure 7. Accuracy study

and accuracy tradeoffs, to the user. A different line of work has focused on designing specific sampling algorithms that speed up GNN training and reduce data transfers at the cost of potentially biasing accuracy (Dong et al., 2021; Liu et al., 2023; Ramezani et al., 2020; Song et al., 2024; 2023b; Wan et al., 2022a; Zhang et al., 2023a;c).

Hybrid parallelism in full-graph training Workload characteristics of full graph training systems are different from data parallel minibatch training as the graph structure used in training does not change between iterations. Thus, overlapping optimized computation and communication schedules can be constructed during preprocessing as a fixed cost which is amortized during training (Cai et al., 2021; Jia et al., 2020; Ma et al., 2019; Md et al., 2021; Wan et al., 2022b; 2023; Wang et al., 2022; 2023a;b; Yang et al., 2023). Recent work has found mini-batch training to be generally more effective than full-graph training when using multiple GPUs (Bajaj et al., 2024).

8 Conclusion

This paper introduces split parallelism, a novel hybrid parallelism strategy for mini-batch training that mitigates redundant sampling, data loading, and computation commonly encountered in data parallelism and push-pull parallelism. A key technical challenge in using split parallelism is to develop a lightweight splitting algorithms that can balance work and minimize communication. This paper proposes a probabilistic splitting algorithm that can push most of the computational overhead offline while achieving probabilistic guarantees. It also shows that split parallelism can be applied transparently using a simple programming API.

Acknowledgements

This material is based upon work supported by the National Science Foundation under Grant No. CNS-2224054. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation. The work was also supported by Adobe Research grants and an Amazon Research Award. We would like to thank our anonymous reviewers and our shepherd Yuke Wang for their valuable feedback.

This manuscript has been authored by UT-Battelle, LLC, under contract DE-AC05-00OR22725 with the US Department of Energy (DOE). The US government retains and the publisher, by accepting the article for publication, acknowledges that the US government retains a nonexclusive, paid-up, irrevocable, worldwide license to publish or reproduce the published form of this manuscript, or allow others to do so, for US government purposes. DOE will provide public access to these results of federally sponsored research in accordance with the DOE Public Access Plan (https://www.energy.gov/doe-public-access-plan). This research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.

REFERENCES

Bajaj, S., Son, H., Liu, J., Guan, H., and Serafini, M. Graph neural network training systems: A performance comparison of full-graph and mini-batch. *Proceedings of the VLDB Endowment*, 18(4):1196 – 1209, 2024.

Balin, M. F., LaSalle, D., and Çatalyürek, Ü. V. Cooperative minibatching in graph neural networks. *arXiv preprint arXiv:2310.12403*, Oct. 2023.

Cai, Z., Yan, X., Wu, Y., Ma, K., Cheng, J., and Yu, F. Dgcl: An efficient communication library for distributed gnn training. In *Proceedings of the Sixteenth European Conference on Computer Systems*, EuroSys '21, pp. 130–144, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450383349. doi: 10.1145/3447786.3456233.

Cai, Z., Zhou, Q., Yan, X., Zheng, D., Song, X., Zheng, C., Cheng, J., and Karypis, G. Dsp: Efficient gnn training with multiple gpus. In *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, PPoPP '23, pp. 392–404, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400700156. doi: 10.1145/3572848.3577528. URL https://doi.org/10.1145/3572848.3577528.

Dong, J., Zheng, D., Yang, L. F., and Karypis, G. Global neighbor sampling for mixed cpu-gpu training on giant graphs. In *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining*, KDD '21, pp. 289–299, 2021.

Fan, R., Wang, W., and Chu, X. Fast sparse gpu kernels for accelerated training of graph neural networks. In *2023 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2023.

Fey, M. and Lenssen, J. E. Fast graph representation learning with pytorch geometric. *arXiv preprint arXiv:1903.02428*, 2019.

Fu, Q., Ji, Y., and Huang, H. H. Tlpgnn: A lightweight two-level parallelism paradigm for graph neural network computation on gpu. In *Proceedings of the 31st International Symposium on High-Performance Parallel and Distributed Computing*, pp. 122–134, 2022.

Gandhi, S. and Iyer, A. P. P3: Distributed deep graph learning at scale. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pp. 551–568, 2021.

Gong, P., Liu, R., Mao, Z., Cai, Z., Yan, X., Li, C., Wang, M., and Li, Z. gsampler: General and efficient gpu-based graph sampling for graph learning. In *ACM 2023 Symposium on Operating Systems Principles (SOSP)*, 2023.

Hamilton, W. L., Ying, R., and Leskovec, J. Inductive Representation Learning on Large Graphs. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS'17, 2017.

Hu, W., Fey, M., Zitnik, M., Dong, Y., Ren, H., Liu, B., Catasta, M., and Leskovec, J. Open graph benchmark: Datasets for machine learning on graphs, 2021.

Jangda, A., Polisetty, S., Guha, A., and Serafini, M. Accelerating graph sampling for graph machine learning using gpus. In *European Conference on Computer Systems (EuroSys)*, 2021.

Jia, Z., Lin, S., Gao, M., Zaharia, M., and Aiken, A. Improving the accuracy, scalability, and performance of graph neural networks with roc. *Proceedings of Machine Learning and Systems*, 2:187–198, 2020.

Kaler, T., Stathas, N., Ouyang, A., Iliopoulos, A.-S., Schardl, T., Leiserson, C. E., and Chen, J. Accelerating training and inference of graph neural networks with fast sampling and pipelining. In Marculescu, D., Chi, Y., and Wu, C. (eds.), *Proceedings of Machine Learning and Systems*, volume 4, pp. 172–189, 2022.

Kaler, T., Iliopoulos, A., Murzynowski, P., Schardl, T., Leiserson, C. E., and Chen, J. Communication-efficient graph neural networks with probabilistic neighborhood expansion analysis and caching. In Song, D., Carbin, M., and Chen, T. (eds.), *Proceedings of Machine Learning and Systems*, volume 5, pp. 477–494. Curan, 2023.

Karypis, G. and Kumar, V. Metis: A software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices. 1997.

LaSalle, D. and Karypis, G. Multi-threaded graph partitioning. In *Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*, pp. 225–236. IEEE, 2013.

Leskovec, J. and Krevl, A. SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data, 2014.

Lin, Z., Li, C., Miao, Y., Liu, Y., and Xu, Y. Pagraph: Scaling gnn training on large graphs via computation-aware caching. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, SoCC '20, pp. 401–415, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450381376. doi: 10.1145/3419111.3421281.

Liu, T., Chen, Y., Li, D., Wu, C., Zhu, Y., He, J., Peng, Y., Chen, H., Chen, H., and Guo, C. BGL: GPU-Efficient

GNN training by optimizing graph data I/O and preprocessing. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pp. 103–118, 2023.

Ma, K., Liu, R., Yan, X., Cai, Z., Song, X., Wang, M., Li, Y., and Cheng, J. Adaptive parallel training for graph neural networks. In *Proceedings of the 30th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, pp. 29–42, 2025.

Ma, L., Yang, Z., Miao, Y., Xue, J., Wu, M., Zhou, L., and Dai, Y. Neugraph: parallel deep neural network computation on large graphs. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pp. 443–458, 2019.

Md, V., Misra, S., Ma, G., Mohanty, R., Georganas, E., Heinecke, A., Kalamkar, D., Ahmed, N. K., and Avancha, S. Distgnn: Scalable distributed training for large-scale graph neural networks. In *SC'21: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2021.

Min, S. W., Wu, K., Huang, S., Hidayetoğlu, M., Xiong, J., Ebrahimi, E., Chen, D., and Hwu, W.-m. Large graph convolutional network training with gpu-oriented data communication architecture. *Proc. VLDB Endow.*, 14 (11):2087–2100, oct 2021.

Pandey, S., Li, L., Hoisie, A., Li, X. S., and Liu, H. C-saw: a framework for graph sampling and random walk on gpus. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, 2020.

Polisetty, S., Liu, J., Falus, K., Fung, Y. R., Lim, S.-H., Guan, H., and Serafini, M. GSplit: Scaling graph neural network training on large graphs via split-parallelism. *arXiv preprint arXiv:2303.13775v1*, Mar. 2023.

Ramezani, M., Cong, W., Mahdavi, M., Sivasubramaniam, A., and Kandemir, M. T. Gcn meets gpu: Decoupling" when to sample" from" how to sample". In *NeurIPS*, 2020.

Song, J., Jang, H., Lim, H., Jung, J., Kim, Y., and Lee, J. Granndis: Fast distributed graph neural network training framework for multi-server clusters. In *Proceedings of the 2024 International Conference on Parallel Architectures and Compilation Techniques*, PACT '24, pp. 91–107, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400706318. doi: 10.1145/3656019.3676892. URL https://doi.org/10.1145/3656019.3676892.

Song, X., Zhang, Y., Chen, R., and Chen, H. Ugache: A unified gpu cache for embedding-based deep learning. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, pp. 627–641, New York, NY, USA, 2023a. Association for Computing Machinery. ISBN 9798400702297. doi: 10.1145/3600006.3613169. URL https://doi.org/10.1145/3600006.3613169.

Song, Z., Gu, Y., Li, T., Sun, Q., Zhang, Y., Jensen, C. S., and Yu, G. Adgnn: Towards scalable gnn training with aggregation-difference aware sampling. *Proceedings of the ACM on Management of Data*, 1(4):1–26, 2023b.

Sun, J., Su, L., Shi, Z., Shen, W., Wang, Z., Wang, L., Zhang, J., Li, Y., Yu, W., Zhou, J., and Wu, F. Legion: Automatically pushing the envelope of Multi-GPU system for Billion-Scale GNN training. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, pp. 165–179, Boston, MA, July 2023. USENIX Association. ISBN 978-1-939133-35-9. URL https://www.usenix.org/conference/atc23/presentation/sun.

Tan, Z., Yuan, X., He, C., Sit, M.-K., Li, G., Liu, X., Ai, B., Zeng, K., Pietzuch, P., and Mai, L. Quiver: Supporting gpus for low-latency, high-throughput gnn serving with workload awareness. *arXiv preprint arXiv:2305.10863*, 2023.

Tripathy, A., Yelick, K., and Buluç, A. n. Distributed matrix-based sampling for graph neural network training. In Gibbons, P., Pekhimenko, G., and Sa, C. D. (eds.), *Proceedings of Machine Learning and Systems*, volume 6, pp. 253–265, 2024.

Veličković, P., Cucurull, G., Casanova, A., Romero, A., Lio, P., and Bengio, Y. Graph attention networks. In *International Conference on Learning Representations*, 2018.

Waleffe, R., Mohoney, J., Rekatsinas, T., and Venkataraman, S. Marius++: Large-scale training of graph neural networks on a single machine. *arXiv preprint arXiv:2202.02365*, 2022.

Wan, C., Li, Y., Li, A., Kim, N. S., and Lin, Y. Bns-gcn: Efficient full-graph training of graph convolutional networks with partition-parallelism and random boundary node sampling. *Proceedings of Machine Learning and Systems*, 4:673–693, 2022a.

Wan, C., Li, Y., Wolfe, C. R., Kyrillidis, A., Kim, N. S., and Lin, Y. PipeGCN: Efficient full-graph training of graph convolutional networks with pipelined feature communication. In *The Tenth International Conference on Learning Representations (ICLR 2022)*, 2022b.

Wan, X., Xu, K., Liao, X., Jin, Y., Chen, K., and Jin, X. Scalable and efficient full-graph gnn training for large graphs. *Proc. ACM Manag. Data*, 1(2), jun 2023. doi: 10.1145/3589288. URL https://doi.org/10.1145/3589288.

Wang, M., Zheng, D., Ye, Z., Gan, Q., Li, M., Song, X., Zhou, J., Ma, C., Yu, L., Gai, Y., Xiao, T., He, T., Karypis, G., Li, J., and Zhang, Z. Deep graph library: A graph-centric, highly-performant package for graph neural networks. 2020.

Wang, P., Li, C., Wang, J., Wang, T., Zhang, L., Leng, J., Chen, Q., and Guo, M. Skywalker: Efficient alias-method-based graph sampling and random walk on gpus. In *2021 30th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pp. 304–317. IEEE, 2021.

Wang, Q., Zhang, Y., Wang, H., Chen, C., Zhang, X., and Yu, G. Neutronstar: Distributed gnn training with hybrid dependency management. In *Proceedings of the 2022 International Conference on Management of Data*, SIGMOD '22, pp. 1301–1315, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450392495. doi: 10.1145/3514221.3526134. URL https://doi.org/10.1145/3514221.3526134.

Wang, Q., Chen, Y., Wong, W.-F., and He, B. Hongtu: Scalable full-graph gnn training on multiple gpus. *Proceedings of the ACM on Management of Data*, 1(4):1–27, 2023a.

Wang, Y., Feng, B., Wang, Z., Geng, T., Barker, K., Li, A., and Ding, Y. MGG: Accelerating graph neural networks with Fine-Grained Intra-Kernel Communication-Computation pipelining on Multi-GPU platforms. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pp. 779–795, Boston, MA, July 2023b. USENIX Association. ISBN 978-1-939133-34-2. URL https://www.usenix.org/conference/osdi23/presentation/wang-yuke.

Wang, Y., Feng, B., Wang, Z., Huang, G., and Ding, Y. TC-GNN: Bridging sparse GNN computation and dense tensor cores on GPUs. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, pp. 149–164, Boston, MA, July 2023c. USENIX Association. ISBN 978-1-939133-35-9. URL https://www.usenix.org/conference/atc23/presentation/wang-yuke.

Wu, Y., Ma, K., Cai, Z., Jin, T., Li, B., Zheng, C., Cheng, J., and Yu, F. Seastar: vertex-centric programming for graph neural networks. In *Proceedings of the Sixteenth European Conference on Computer Systems*, pp. 359–375, 2021.

Yang, D., Liu, J., Qi, J., and Lai, J. Wholegraph: A fast graph neural network training framework with multi-gpu distributed shared memory architecture. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '22. IEEE Press, 2022a. ISBN 9784665454445.

Yang, J., Tang, D., Song, X., Wang, L., Yin, Q., Chen, R., Yu, W., and Zhou, J. Gnnlab: a factored system for sample-based gnn training over gpus. In *Proceedings of the Seventeenth European Conference on Computer Systems*, pp. 417–434, 2022b.

Yang, S., Zhang, M., Dong, W., and Li, D. Betty: Enabling large-scale gnn training with batch-level graph partitioning. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS 2023, pp. 103–117, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9781450399166. doi: 10.1145/3575693.3575725. URL https://doi.org/10.1145/3575693.3575725.

Ye, Z., Lai, R., Shao, J., Chen, T., and Ceze, L. Sparsetir: Composable abstractions for sparse compilation in deep learning. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, pp. 660–678, 2023.

Zhang, M., Hu, Q., Sun, P., Wen, Y., and Zhang, T. Boosting distributed full-graph gnn training with asynchronous one-bit communication, 2023a. URL https://arxiv.org/abs/2303.01277.

Zhang, X., Shen, Y., Shao, Y., and Chen, L. Ducati: A dual-cache training system for graph neural networks on giant graphs with the gpu. *Proc. ACM Manag. Data*, 1(2), jun 2023b. doi: 10.1145/3589311. URL https://doi.org/10.1145/3589311.

Zhang, Z., Luo, Z., and Wu, C. Two-level graph caching for expediting distributed gnn training. In *IEEE INFOCOM 2023 - IEEE Conference on Computer Communications*, pp. 1–10, 2023c. doi: 10.1109/INFOCOM53939.2023.10228911.

Zheng, C., Chen, H., Cheng, Y., Song, Z., Wu, Y., Li, C., Cheng, J., Yang, H., and Zhang, S. Bytegnn: efficient graph neural network training at large scale. *Proc. VLDB Endow.*, 15(6):1228–1242, feb 2022. ISSN 2150-8097. doi: 10.14778/3514061.3514069. URL https://doi.org/10.14778/3514061.3514069.

Zheng, D., Ma, C., Wang, M., Zhou, J., Su, Q., Song, X., Gan, Q., Zhang, Z., and Karypis, G. Distdgl: Distributed graph neural network training for billion-scale graphs. *arXiv preprint arXiv:2010.05337*, 2020.

Zhou, Y., Song, Y., Leng, J., Liu, Z., Cui, W., Zhang, Z., Guo, C., Chen, Q., Li, L., and Guo, M. Adaptgear: Accelerating gnn training via adaptive subgraph-level kernels on gpus. *Proceedings of the 20th ACM International Conference on Computing Frontiers*, 2023. URL https://api.semanticscholar.org/CorpusID:258960480.

Zhu, R., Zhao, K., Yang, H., Lin, W., Zhou, C., Ai, B., Li, Y., and Zhou, J. Aligraph: A comprehensive graph neural network platform. *Proc. VLDB Endow.*, 12(12): 2094–2105, 2019.

A Artifact Appendix

A.1 Abstract

The artifact contains the source code of SPA and scripts to run experiments on the baselines to reproduce the results in our paper. The artifact primarily requires a machine with four NVIDIA V100 GPUs, fully connected via NVLink. The artifact supports the key results in Table 3, demonstrating SPA's effectiveness in improving end-to-end training time relative to the baselines by eliminating redundant work while minimizing load imbalance and communication costs.

A.2 Artifact check-list (meta-information)

- **Algorithm:** Graph Neural Networks
- **Program:** Pytorch, CUDA and C/C++ Code
- **Dataset:** Preprocessed datasets are provided.
- **Hardware:** AWS EC2 p3.8xlarge instance consisting of at least 244 GB RAM with 4 NVIDIA V100 GPUs (16GB memory) and GPUs are connected to the CPU with a PCIe 3.0 x 16 bus and with each other via NVLink.
- **Metrics:** Execution time
- **Output:** Tables and graphs
- **Experiments:** We provide a README in the artifact containing instructions to set up, run experiments, and post-process the results to generate tables and figures. The maximum variation is 5%.
- **How much disk space is required (approximately)?:** The artifact requires approximately 300GB to save all the data and graphs.
- **How much time is needed to prepare workflow (approximately)?:** We provide a compile script (build.sh) which takes 10 min and download script (download.sh) which takes 50 min.
- **How much time is needed to complete experiments (approximately)?:** The total execution time takes approximately 12 hours.
- **Publicly available?:** Yes
- **Code licenses (if publicly available)?:** Apache-2.0
- **Archived (provide DOI)?:** Zenedo DOI will be provided after evaluation.

A.3 Description

All computational artifacts are in a single repository, in which the README contains the detailed instructions to set up and reproduce the figures and tables in the paper:

https://github.com/Juelin-Liu/dgl.git
(Branch: spa-mlsys-ae)

A.3.1 Hardware dependencies

By default, our experiments use an AWS EC2 p3.8xlarge instance with 4 NVIDIA V100 GPUs (16GB memory) and Xeon E5-2686 v4 @ 2.70GHz, with 32 CPU cores and 244 GB RAM. GPUs are connected to the CPU with a PCIe 3.0 x 16 bus and to each other via NVLink. For experiments with 8 GPUs, we use a similar p3.16xlarge instance having 64 CPU cores and 488 GB of RAM.

A.3.2 Software dependencies

The artifact uses CUDA v11.8 and GCC version 11.3 (-O2) to compile all the tested systems. We use PyTorch v2.0 as the backend for all the tested systems. We recommend using Docker to setup the software dependencies and provide instructions for the setup in our README file.

A.3.3 Datasets

We provide pre-processed datasets, including the graph topology and partition maps that can be downloaded from Amazon S3. Alternatively, you can use the scripts in the repository to generate the prepared datasets. Notice that this would take several days on a single machine to generate all partition maps for the graphs.

A.4 Installation

We provide a Dockerfile for setting up the environment for compiling the source code. The instructions for running the Dockerfile are provided in the README. We also provide scripts used for downloading the dependencies if you prefer not to use Docker. Please note to set the branch to spa-mlsys-ae and recursively pull all the submodules, before running the Docker container. After installing all the dependencies, and pulling all submodules, simply execute the build.sh script to install Spara into your Conda environment.

A.5 Experiment workflow

S_1 The first step is to obtain the input datasets, which include the graph topology data and partition maps. The script *download.sh* can be used to download these pre-processed files automatically.

S_2 After obtaining the prepared datasets, you can run the main experiment by executing the bash script *experiment/script/main.sh*. This script runs all the baselines and generates the log file. (Expected time: 120 min, depends on S_1.)

S_3 Run the script $experiment/partition_ablation$ to collect the training logs for varying partitioning strategies. (depends S_1)

S_4 We postprocess the logs from S_2 and S_3 to generate Figures 3 using the notebook $plot/time_breakdown$. (depends on S_2 and S_3).

S_5 Postprocess the logs from S_2 and S_3 using the notebook $plot/main$ to generate Table 3 showing the relative performance of spara against the baselines. (depends on S_2 and S_3).

S_6 Run the sampling simulation $experiment/sample_main$, to generate the varying edges computed and features loaded for varying batch sizes (1024,256) and graphs(papers100M, orkut, friendster) to generate the datapoints in Table 1. (Expected time: 30min, depends on S_1)

S_7 Run the script $experiment/scripts/ablation.sh$ to run all the ablation experiments on papers100M graph. (depends on S_1)

S_8 Post-process the logs generated in the previous step with the Jupyter notebook $plot/final_ablation$ (Expected time: 3 hours, depends on S_6)

S_9 : Run the script $experiment/script/simulate$ to generate the workload characteristics for various partitioning schemes (depends on S_1)

S_{10} : Post process the workloads generated in step S_8 with the notebook $plot/simulation_plot$ to generate Figures 5. (depends on S_8)

A.6 Evaluation and expected result

The key results of the paper can be demonstrated at S_5 in our workflow, where we generate the results in Table 3, demonstrating the effectiveness of spa's split parallelism relative to the baselines.

A.7 Notes

The artifact can be checked for functionality on a machine with four GPUs with or without NVLink connections.

A.8 Methodology

Submission, reviewing and badging methodology:

- http://cTuning.org/ae/submission-20190109.html
- http://cTuning.org/ae/reviewing-20190109.html
- https://www.acm.org/publications/policies/artifact-review-badging