Cooperative Minibatching in GNNs

Anonymous Authors[1]

Abstract

Training Graph Neural Networks at a large scale requires significant computational resources. One of the most effective ways to reduce resource requirements, especially in the distributed setting, is minibatch training coupled with graph sampling. However, existing approaches suffer from the Neighborhood Explosion Phenomenon (NEP), which seems to be the main bottleneck limiting scaling. To reduce the effects of NEP, we propose a new approach called Cooperative Minibatching. Our approach takes advantage of the fact that the sampled subgraph size is a sublinear and concave function of the batch size resulting into significant reductions in the amount of work performed per seed vertex with increasing batch sizes. Hence, it is favorable for processors to work on a large minibatch together instead of working on separate smaller minibatches. We also show how to take advantage of the same phenomenon in serial execution by generating dependent consecutive minibatches. Our experimental evaluations indicate up to $4\times$ bandwidth savings, for fetching vertex embeddings, by just increasing this dependency without harming the model convergence.

1. Introduction

Graph Neural Networks (GNNs) have become de facto deep learning models for unstructured data, achieving state-of-the-art results on various application domains involving graph data such as recommendation systems (Ying et al., 2018; Wu et al., 2020), fraud detection (Patel et al., 2022; Liu et al., 2022), identity resolution (Xu et al., 2019), quantum chemistry (Nakata & Shimazaki, 2017; Hu et al., 2021) and traffic forecasting (Jiang & Luo, 2022). However, as the usage of technology continues to increase, the amount of data generated by these applications is growing, exponentially, resulting in large and complex graphs that are infeasible or too time-consuming to train on a single accelerator device. For example, some social media graphs are reaching billions of vertices and trillions of interactions. Therefore, to address this issue, developing techniques for speeding up GNN training using distributed computing has become essential.

Before the GNNs' rise, a great line of research focused on distributing Deep Neural Network (DNN) training to lift scalability limitations. Techniques such as data parallelism (Ginsburg et al., 2017; Goyal et al., 2018), pipelining (Narayanan et al., 2019), and intra-layer parallelism (Dean et al., 2012) were employed. Following the success of traditional distributed DNN training, the same techniques have been also adapted to GNN training, such as data parallelism (Zheng et al., 2021; Gandhi & Iyer, 2021; Lin et al., 2020; Zhu et al., 2019) and intra-layer parallelism (Tripathy et al., 2020).

The parallelization techniques mentioned earlier are used to scale both full batch and minibatch training in a distributed setting. Minibatch training (Bertsekas, 1994) is the go-to method to train DNN models as it outperforms full batch training in general (Allen-Zhu & Hazan, 2016; Li et al., 2014). In the distributed setting, minibatch training for DNNs using data parallelism is straightforward. The training samples are partitioned across the Processing Elements (PE) and they compute the forward/backward operations on their minibatches. The only communication required is an all-reduce operation to synchronize the gradients across PEs. In addition to being easily parallelizable in the distributed setting, minibatch training has been shown to significantly reduce time to convergence in structured DNN models (Wilson & Martinez, 2003; Keskar et al., 2016), and more recently has been shown to also offer the same benefit for GNNs (Zheng et al., 2021).

Unfortunately, minibatch training a GNN model is more challenging than a usual DNN model. GNNs turns the given graph denoting relationships between the data points, into computational dependencies. Thus, in GNNs, the computations for each minibatch for a given data points, have different computational structure, as it is performed on their L-hop neighborhood, in an L layer GNN model. Real world graphs usually are power law graphs (Artico et al., 2020),

[1]Anonymous Institution, Anonymous City, Anonymous Region, Anonymous Country. Correspondence to: Anonymous Author <anon.email@domain.com>.

Preliminary work. Under review by the International Conference on Machine Learning (ICML). Do not distribute.

with very low diameter, thus, it is a challenge to train deep GNN models as the L-hop neighborhood grows exponentially with increasing L, easily reaching almost the whole graph within few hops.

A single epoch of full-batch GNN training requires computation proportional to the number of layers L and the size of the graph. However, minibatch training requires more operations to process a single epoch due to repeating calculations in the 2nd through Lth layers. As the batch size decreases, the number of repeated calculations increases. This is because the vertices and edges have to be processed each time appear in the L-hop neighborhood. The likelihood of repeated calculations increases as there are more minibatches in an epoch. Thus, it is natural to conclude that using larger batch sizes in GNNs reduces the runtime complexity of an epoch in contrast to regular DNN models. Our contributions in this work utilizing this important observation can be listed as follows:

- Investigate work vs batch size relationship and prove that the cost of processing a minibatch is a sublinear and concave function of the batch size (Sections 3.1 and 3.2).
- Utilize this sublinear relationship by combining data and intra-layer parallelism to process a minibatch across multiple PEs for reduced work and up to 22% faster runtimes (Section 3.3). We call this new approach *Cooperative Minibatching*.
- Use the same idea to generate consecutive dependent minibatches to increase temporal vertex embedding access locality (Section 3.4). This approach can reduce the transfer amount of vertex embeddings up to $4\times$, without harming model convergence.
- Show that the two approaches are orthogonal. Together, the overall work is reduced by $1.4\times$ and vertex embedding transfer amount is reduced by $3\times$ on mag240M.

2. Background

A graph $\mathcal{G} = (V, E)$ consists of vertices V and edges $E \subset V \times V$ along with optional edge weights $A_{ts} > 0, \forall (t \to s) \in E$. Given a vertex s the 1-hop neighborhood $N(s)$ is defined as $N(s) = \{t | (t \to s) \in E\}$, and it can be naturally expanded to a set of vertices S as $N(S) = \cup_{s \in S} N(s)$.

GNN models work by passing previous layer embeddings (H) from neighbors of s to s, and then combining them using a nonlinear function $f^{(l)}$ at layer l:

$$H_s^{(l+1)} = f^{(l)}(H_s^{(l)}, \{H_t^{(l)} \mid t \in N(s)\}) \tag{1}$$

If the GNN model has L layers, then the loss is computed by taking the layer L's embeddings and averaging them over the set of training vertices $V_t \subset V$ for *full-batch* training. In L-layer full-batch training, the total number of vertices that needs to be processed is $L|V|$.

2.1. Minibatching in GNNs

In minibatch training, a random subset of training vertices, called *Seed Vertices*, is selected, and training is done over the (sampled) subgraph composed of L layer neighborhood of the seed vertices. On each iteration, minibatch training computes the loss on seed vertices, that are random subsets of the training set V_t.

Given a set of vertices S, we define l-th layer expansion set S_l as:

$$S_0 = S, \quad S_{l+1} = S_l \cup N(S_l) \tag{2}$$

For GNN computations, S_l would also denote the set of the required vertices to compute (1) at each layer l. We will abuse this notation and use $\{s\}_l$ to denote l-layer expension starting from single vertex $s \in V$.

That is, for a single minibatch, the total number of vertices that needs to be processed is $\sum_{l=1}^{L} |S_l|$. There are $\frac{|V|}{|S_0|}$ minibatches assuming $V_t = V$. Since each $|S_l| \geq |S_0|$, and a single epoch of minibatch training needs to go over the whole dataset, the work $W(|S_0|)$ for a single epoch is:

$$W(|S_0|) = \frac{|V|}{|S_0|} \sum_{l=1}^{L} E[|S_l|] \geq \frac{|V|}{|S_0|} \sum_{l=1}^{L} |S_0| = L|V|$$

where $E[|S_l|]$ is the expected number of sampled vertices in layer l and $|S_0|$ is the batch size. That is, the total amount of work to process a single epoch increases over full-batch training. The increase in work due to minibatch training is thus encoded in the ratios $\frac{E[|S_l|]}{|S_0|}, 1 \leq l \leq L$.

Next, we will briefly present some of the sampling techniques. When sampling is used with minibatching, the minibatch subgraph may potentially become random. However, the same argument of the increasing total amount of work holds for them too, as seen in Figure 3.

2.2. Graph Sampling

Below, we discuss 3 different sampling algorithms for minibatch training of GNNs. Our focus in this work in those whose expected number of sampled vertices is a function of batch size. All these methods are applied recursively for GNN models with multiple layers.

2.2.1. Neighbor Sampling (NS)

Given a fanout parameter k and a batch of seed vertices S_0, NS (Hamilton et al., 2017) samples the neighborhoods of vertices randomly. Given a batch of vertices S_0, a vertex $s \in S_0$ with degree $d_s = |N(s)|$, if $d_s \leq k$, NS uses the

full neighborhood $N(s)$, otherwise it samples k random neighbors for the vertex s.

One way to sample k edges from d_s options is to use the reservoir algorithm (Vitter, 1985). In this algorithm, the first k edges are included in the reservoir. Then, for the rest of the $d_s - k$ edges, the ith edge $(t \to s)$ rolls the uniform random integer $0 \leq r_{ts} \leq i$ and replaces the item in slot r_{ts} if $r_{ts} < k$.

2.2.2. LABOR Sampling

Given a fanout parameter k and a batch of seed vertices S_0, LABOR-0 (Balın & Çatalyürek, 2022) samples the neighborhoods of vertices as follows. First, each vertex rolls a uniform random number $0 \leq r_t \leq 1$. Given batch of vertices S_0, a vertex $s \in S_0$ with degree $d_s = |N(s)|$, the edge $(t \to s)$ is sampled if $r_t \leq \frac{k}{d_s}$. Because different seed vertices $\in S_0$ end up using the same random variate r_t for the same source vertex t, LABOR is expected to sample fewer vertices than NS, and this is experimentally demonstrated in the LABOR paper.

The LABOR-* algorithm is the importance sampling variant of LABOR-0 and samples an edge $(t \to s)$ if $r_t \leq c_s \pi_t$, where π is importance sampling probabilities optimized to minimize the expected number of sampled vertices and c_s is a normalization factor. LABOR-* is guaranteed to sample fewer vertices than LABOR-0.

Note that, choosing $k \geq \max_{s \in V} d_s$ corresponds to training with full neighborhoods for both NS and LABOR methods.

2.2.3. RandomWalk Sampling

Given a walk length o, a restart probability p, number of random walks a, and a fanout k, and a batch of vertices S_0, a vertex $s \in S_0$, a *Random Walk* (Ying et al., 2018) starts from s and each step picks a random neighbor s' from $N(s)$. For the remaining $o - 1$ steps, the next neighbor is picked from $N(s')$ with probability $1 - p$, otherwise it is picked from $N(s)$. This process is repeated a times for each seed vertex and at the end, we pick the top k most visited vertices as the *neighbors* of s for the current layer.

Notice that random walks correspond to weighted neighbor sampling from a graph with adjacency matrix $\tilde{A} = \sum_{i=1}^{o} A^i$, where the weights of $\tilde{A}$ depend on the parameters a, p and k. Random walks give us the ability to sample from $\tilde{A}$ without actually forming $\tilde{A}$.

2.3. Independent Minibatching

Independent minibatching is commonly used in multi-GPU, and distributed GNN training frameworks (Zheng et al., 2021; Gandhi & Iyer, 2021; Lin et al., 2020; Zhu et al., 2019) to parallelize the training and allows scaling to larger problems. Each Processing Element (PE, i.e., GPUs, CPUs, or cores of multi-core CPU), starts with their own S_0 as the seed vertices, and compute $S_1, \ldots, S_L$ along with the sampled edges to generate minibatches (see Figure 1). Computing $S_1, \ldots, S_L$ depends on the chosen sampling algorithm, such as the ones explained in Section 2.2.

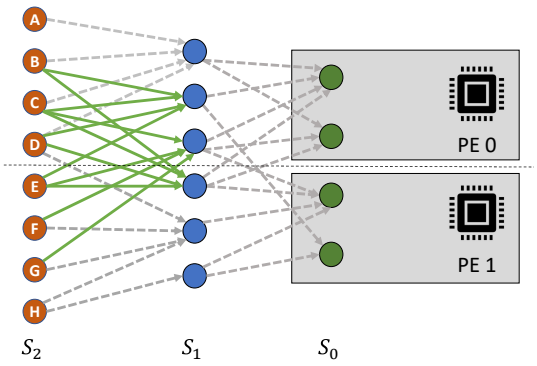


Figure 1. Minibatches of two processing elements may share edges in the second layer and vertices starting in the first layer. For independent minibatching, the solid green edges are shared by both processing elements represent duplicate work, and input nodes B through G are duplicate along with the directed endpoints of green edges. However for cooperative minibatching, the vertices and edges are partitioned across the PEs and the edges crossing the line between the two PEs necessitate communication.

Independent minibatching has the advantage that doing a forward/backward does not involve any communication with other PEs after the initial minibatch preparation stage at the expense of duplicate work (see Figures 1 and 2a).

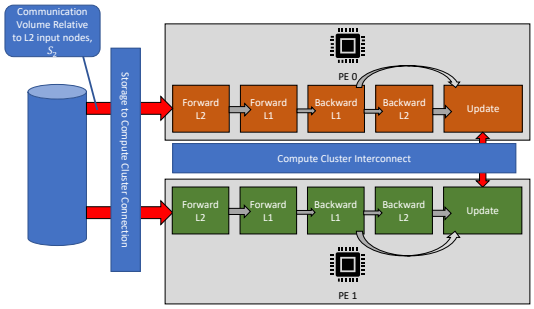
3. Cooperative Minibatching

In this section, we first prove that for any given graph, the work of an epoch will be monotonically nonincreasing with the increasing batch sizes. We will also quantify what affects the shape of the work vs batch size curve. After that, we will propose two algorithms that can take advantage of this monotonicity.

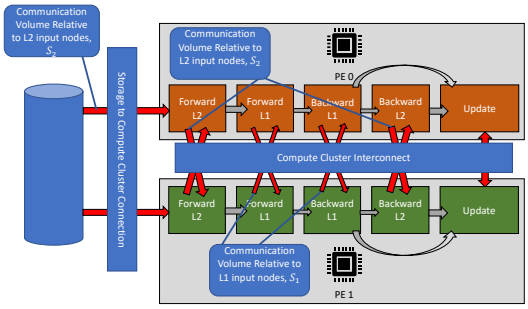
3.1. Work Monotonicity Theorem

Theorem 3.1. *The work per epoch required to train a GNN model using minibatch training is monotonically nonincreasing as the batch size increases.*

Proof. Given any $n \geq 2$, let's say we uniform randomly sample without replacement $S_0 \subset V$, where $n = |S_0|$. Now note that for any $S_0' \subset S_0$, using the definition in (2), we have $S_l' \subset S_l, \forall l$. We will take advantage of that and define $S_0' = S_0 \setminus \{s\}$ in following expression.



(a) Independent minibatching via duplication. Each PE has an independent minibatch to process, and fetches it from the Storage independently. This leads to a waste of Storage bandwidth as duplicate vertices in Figure 1 need to be transferred multiple times.



(b) Cooperative minibatching without duplication. Multiple PEs cooperate to process a single minibatch, and fetches it from the Storage cooperatively. The communication through the Compute Cluster Interconnect is performed for the edges that cross the partition boundary as shown in Figure 1.

Figure 2. The main bottleneck of distributed training systems is the Storage Connection that is used to fetch the data required for minibatch iterations. For example, in a multi-GPU system, PEs would refer to GPUs, storage to system RAM, storage connection to PCI-e, and Compute Cluster Interconnect to GPU interconnect. Cooperative minibatching reduces Storage bandwidth requirements as the thickness of the red arrows indicate.

$$\begin{aligned}
\sum_{\substack{s\in S_0\\ S_0'=S_0\setminus\{s\}}} |S_l| - |S_l'| &= \sum_{\substack{s\in S_0\\ S_0'=S_0\setminus\{s\}}} \sum_{w\in S_l} \mathbb{1}[w\notin S_l'] \\
&= \sum_{\substack{s\in S_0\\ S_0'=S_0\setminus\{s\}}} \sum_{w\in\{s\}_l} \mathbb{1}[w\notin S_l'] \\
&= \sum_{\substack{s\in S_0\\ S_0'=S_0\setminus\{s\}}} \sum_{w\in\{s\}_l} \mathbb{1}[w\notin \{s'\}_l, \forall s'\in S_0'] \\
&= \sum_{w\in S_l} \sum_{\substack{s\in S_0\\ w\in\{s\}_l}} \mathbb{1}[w\notin \{s'\}_l, \forall s'\in S_0\setminus\{s\}]
\end{aligned} \tag{3}$$

In the last expression, for a given $w \in S_l$, if there are two different elements $s, s' \in S_0$ such that $w \in \{s\}_l$ and $w \in \{s'\}_l$, then the indicator expression will be 0. It will be 1 only if $w \in \{s\}_l$ for a unique $s \in S_0$. So:

$$\begin{aligned}
\sum_{w\in S_l} \sum_{\substack{s\in S_0\\ w\in\{s\}_l}} \mathbb{1}[w\notin \{s'\}_l, \forall s'\in S_0\setminus\{s\}] &= \sum_{\substack{w\in S_l\\ \exists! s\in S_0, w\in\{s\}_l}} 1 \\
&= |\{w\in S_l \mid w\in\{s\}_l, \exists! s\in S_0\}| \le |S_l|
\end{aligned} \tag{4}$$

Using this, we can get:

$$\begin{aligned}
&\sum_{\substack{S_0'\subset S_0\\ |S_0'|+1=|S_0|}} |S_l| - |S_l'| \le |S_l| \\
\iff & |S_0||S_l| - \sum_{\substack{S_0'\subset S_0\\ |S_0'|+1=|S_0|}} |S_l'| \le |S_l| \\
\iff & |S_l|(|S_0|-1) \le \sum_{\substack{S_0'\subset S_0\\ |S_0'|+1=|S_0|}} |S_l'| \\
\iff & |S_l|(|S_0|-1) \le |S_0|E[|S_l'|] \\
\iff & \frac{|S_l|}{|S_0|} \le \frac{E[S_l']}{|S_0'|}
\end{aligned}$$

Since S_0 was uniformly randomly sampled from V, its potential subsets S_0' are also uniformly randomly picked from V as a result. Then, taking an expectation for the random sampling of $S_0 \subset V$, we conclude that $\frac{E[|S_l|]}{|S_0|} \le \frac{E[|S_l'|]}{|S_0'|}$, i.e., the expected work of batch size n is not greater than the work of batch of size $n-1$. This implies that the work with respect to batch size is a monotonically nonincreasing function. □

Empirical evidence can be seen in Figure 3. In fact, the decrease is related to the cardinality of the following set:

$$T_l(S) = \{w \in S_l \mid w \in \{s\}_l, \exists! s \in S_0\}$$

When $T(S_0)$ is equal to S_l, the work is equal as well. In the next section, we further investigate the effect of $|T(S_0)|$ on $E[|S_l|]$.

3.2. Overlap monotonicity

In addition to the definition of $T(S)$ above, if we define the following set $T^2(S)$:

$$T_l^2(S) = \{w \in S_l \mid w \in \{s\}_l \cap \{s'\}_l, \exists!\{s, s'\} \subset S_0\}$$

Theorem 3.2. *The expected subgraph size $E[|S_l|]$ required to train a GNN model using minibatch training is a sublinear and concave function of batch size, $|S_0|$, sublinear here means that $E[|S_l|] = o(|S_0|)$.*

Proof. Given any $n \geq 2$, let's say we uniformly randomly sample without replacement $S_0 \subset V$ of size n.

$$\begin{aligned}
|T_l(S_0)| - 2|T_l^2(S_0)| &= \sum_{\substack{S_0' \subset S_0 \\ |S_0'|+1=|S_0|}} |T_l(S_0)| - |T_l(S_0')| \\
&= |S_0||T_l(S_0)| - |S_0|E[|T_l(S_0')|] \\
\iff (|S_0|-1)|T_l(S_0)| &= |S_0|E[|T_l(S_0')|] - 2|T_l^2(S_0)| \\
\iff \frac{|T_l(S_0)|}{|S_0|} &= \frac{E[|T_l(S_0')|]}{|S_0'|} - \frac{2|T_l^2(S_0)|}{|S_0'||S_0|} \\
\implies \frac{|T_l(S_0)|}{|S_0|} &\leq \frac{E[|T_l(S_0')|]}{|S_0'|}
\end{aligned} \tag{5}$$

where the first equality above is derived similar to Equations (3) and (4). Overall, this means that the overlap between vertices increases as the batch size increases. Utilizing our finding from Equations (3) and (4), we have:

$$\begin{aligned}
&\sum_{\substack{S_0' \subset S_0 \\ |S_0'|+1=|S_0|}} |S_l| - |S_l'| = |T_l(S_0)| \\
\implies &|S_0||S_l| - |S_0|E[|S_l'|] = |T_l(S_0)| \\
\implies &|S_l| = E[|S_l'|] + \frac{|T_l(S_0)|}{|S_0|} \\
\implies &E[|S_l|] = E[|S_l'|] + \frac{E[|T_l(S_0)|]}{|S_0|}
\end{aligned} \tag{6}$$

Note that the last step involved taking expectations for the random sampling of S_0. See the recursion embedded in the equation above, the expected size of the subgraph S_l with batch size $|S_0|$ depends on the expected size of the subgraph S_l' with batch size $|S_0| - 1$. Expanding the recursion, we get:

$$E[|S_l|] = \sum_{i=1}^{|S_0|} \frac{E[|T_l(V_0^i)|]}{i} \tag{7}$$

where V_0^i is a random subset of V of size i. Since $\frac{E[|T_l(V_0^i)|]}{i}$ is monotonically nonincreasing as i increases as we showed in (5), we conclude that $E[|S_l|]$ is a sublinear and concave function of the batch size, $|S_0|$. □

So, the slope of the expected number of sampled vertices flattens as batch size increases, see the last row in Figure 3. Note that this implies work monotonicity as well.

3.3. Cooperative Minibatching

As explained in Section 2, independent distributed minibatch training (Figure 2a), can not take advantage of the reduction in work with increasing batch sizes, because practically, it uses separate small batches on each PE for each iteration of training.

Here, we propose *Cooperative Minibatching* method that will take advantage of the reduction with increasing batch sizes in multi PE settings. In Cooperative Minibatching, a single batch will be processed by all the PEs in parallel as shown in Figure 2b.

We achieve this as follows: we first partition the graph in 1D fashion by logically assigning each vertex and their incoming edges to PEs. Next, each PE samples their batch seed vertices from the training vertices they own. Then using any sampling algorithm, each PE samples the incoming edges, which they already own, for their seed vertices. Note that, some of these edges will have source vertices that reside on a different PE. The PEs exchange the vertex ids so that each PE receives vertices they own. This process is repeated recursively for GNN models with multiple layers. The vertex exchange information is cached to be used during the forward/backward passes.

For the forward/backward passes, the same communication pattern used during cooperative sampling is used to send and receive input and intermediate layer embeddings before each graph convolution operation.

3.4. Cooperative Dependent Minibatch

In light of Theorems 3.1 and 3.2, consider doing the following: choose $\kappa > 1$, then sample a batch S_0 of size $\kappa\beta$, i.e., $\kappa\beta = |S_0|$ to get $S_0, \ldots, S_L$. Then sample κ minibatches S_0^i, of size $\beta = |S_0^i|$ from this batch of size $\kappa\beta$ to get $S_0^i, \ldots, S_L^i, \forall i \in \{0, \ldots, \kappa - 1\}$. In the end, all of the input features required for these minibatches will be a subset of the input features of the large batch, i.e. $S_j^i \subset S_j, \forall i, j$. This means that the collective input feature requirement of these κ batches will be $|S_L|$, the same as our batch of size $\kappa\beta$. Hence, we can now take advantage of the sublinear growth of the work of Theorem 3.2.

Note that, if one does not use any sampling algorithms and

proceeds to use the full neighborhoods, this technique will not give any benefits as by definition the l-hop neighborhood of a batch of size $\kappa\beta$ will be equal to the union of the l-hop neighborhoods of batches of size β. However for sampling algorithms, any overlapping vertex sampled by any two of batches of size β might end up with different random neighborhoods resulting into a larger number of sampled vertices. Thus, having a single large batch ensures that only a single random set of neighborhood is used for any vertex processed over a period of κ batches.

The approach described above has a nested iteration structure and the minibatches part of one κ group will be significantly different than another group and this might slightly affect convergence. We propose an alternative approach that does not require recursion and still take advantage of the same phenomenon for the NS and LABOR sampling algorithms.

As described in Section 2, NS algorithm works by using the random variate r_{ts} for each edge $(t \to s)$. Being part of the same minibatch means that a single random variate r_{ts} will be used for each edge. If we use a Pseudo Random Number Generator (PRNG) to generate these random variates, by initializing the PRNG with a random seed z along with t and s, we can ensure that the first rolled random number r_{ts} from the PRNG stays fixed when the random seed z is fixed. We propose to do the following, given random seeds z_1 and z_2, let's say we wanted to use z_1 for the first κ iterations and would later switch to the seed z_2. If we could make this switch in a continuous fashion, that would solve our problem. This switch can be made continuous as follows, if we are processing the batch number $i < \kappa$ in the group of κ batches, then we want the contribution of z_2 to be $c = \frac{i}{\kappa}$, while the contribution of z_1 is $1 - c$. We can sample $n^1_{ts} \sim \mathcal{N}(0,1)$ with seed z_1 and $n^2_{ts} \sim \mathcal{N}(0,1)$ with seed s_2. Then we combine them as

$$n_{ts}(c) = \cos(\frac{c\pi}{2})n^1_{ts} + \sin(\frac{c\pi}{2})n^2_{ts},$$

note that $n_{ts}(0) = n^1_{ts}$, $n_{ts}(1) = n^2_{ts}$ and $n_{ts}(c) \sim \mathcal{N}(0,1), \forall c \in \mathbb{R}$ also, then we can set $r_{ts} = \Phi(n_{ts}(c))$, where $\Phi(x) = \mathbb{P}(Z \leq x)$ for $Z \sim \mathcal{N}(0,1)$, to get $r_{ts} \sim U(0,1)$ that the NS algorithm can use. Dropping s from all the notation above gives the version for LABOR. In this way, the random variates slowly change as we are switching from one group of κ batches to another. We will experimentally evaluate the locality benefits and the overall effect of this algorithm on convergence in Section 4.2.

4. Experiments

Setup: In our experiments, we use the following datasets: reddit (Hamilton et al., 2017), papers100M (Hu et al., 2020), mag240M (Hu et al., 2021), yelp. flickr (Zeng et al., 2020). Details about these datasets are given in Table 1. We use Neighbor Sampling (NS) (Hamilton et al., 2017), LABOR Sampling (Balın & Çatalyürek, 2022) and Random Walks (RW) (Ying et al., 2018) to form sampled minibatches. We used a fanout of $k = 10$ for the samplers. In addition, Random Walks used length of $o = 3$, restart probability $p = 0.5$ and number of random walks from each seed $a = 100$. All our experiments involve a GCN model with $L = 3$ layers (Hamilton et al., 2017), with 1024 hidden dimension for mag240M and papers100M and 256 for the rest. We use the Adam optimizer (Kingma & Ba, 2014) with 0.001 learning rate.

Implementation: We implemented our experiments using C++ and Python on a fork of the DGL framework (Wang et al., 2019) with the Pytorch backend (Paszke et al., 2019). All our experiments were repeated 50 times and averages are presented. Early stopping was used during model training runs. So, the variance of our convergence plots as we go to the right along the x-axis increases because the number of runs that were ongoing keeps decreasing.

We first compare how the work to process an epoch changes for different graph sampling algorithms. Next, we will show how dependent batches introduced in Section 3.4 benefits GNN training. We will also show the runtime benefits of cooperative minibatching compared to independent minibatching in the multi-GPU setting. Finally, we will show that these two techniques are orthogonal, can be combined to get multiplicative savings. Finally, we investigate whether there is any practical difference between independent minibatching vs cooperative minibatching when it comes model convergence behavior in Appendix A.2.

4.1. Demonstrating monotonicity of work

We use three sampling approaches, NS, LABOR, and RW, to demonstrate that the work to process an epoch decreases as the batch size increases for the $L = 3$ layer case. We carried out evaluation in two problem settings: node and edge prediction. For node prediction, a batch of training vertices is sampled with a given batch size. Then, the graph sampling algorithms described in Section 2.2 are applied to sample the neighborhood of this batch. The top row of Figure 3 shows how many input vertices is required on average to process an epoch, specifically $\frac{E[|S_3|]}{|S_0|}$. For edge prediction, we make the graph into an undirected one and sample a batch of edges. For each of these edges a random negative edge (an edge that is not part of E) with one endpoint coinciding with the positive edge is sampled. Then, all of the endpoints of these positive and negative edges are used as seed vertices to sample their neighborhoods. The bottom row Figure 3 shows $E[|S_3|]$.

We can see that in all use cases, datasets and sampling algorithms, the work to process an epoch is monotonically

Table 1. Properties of the datasets used in experiments: numbers of vertices, edges, avg. degree, features, sampling budget used, and training, validation and test vertex split.

Dataset	$\lvert V\rvert$	$\lvert E\rvert$	$\frac{\lvert E\rvert}{\lvert V\rvert}$	# feats.	cache size	train - val - test (%)
flickr	89.2K	900K	10.09	500	70k	50.00 - 25.00 - 25.00
yelp	717K	14.0M	19.52	300	200k	75.00 - 10.00 - 15.00
reddit	233K	115M	493.56	602	60k	66.00 - 10.00 - 24.00
papers100M	111M	1.6B	14.55	128	2M	1.09 - 0.11 - 0.19
mag240M	244M	1.72B	7.08	768	2M	0.45 - 0.06 - 0.04

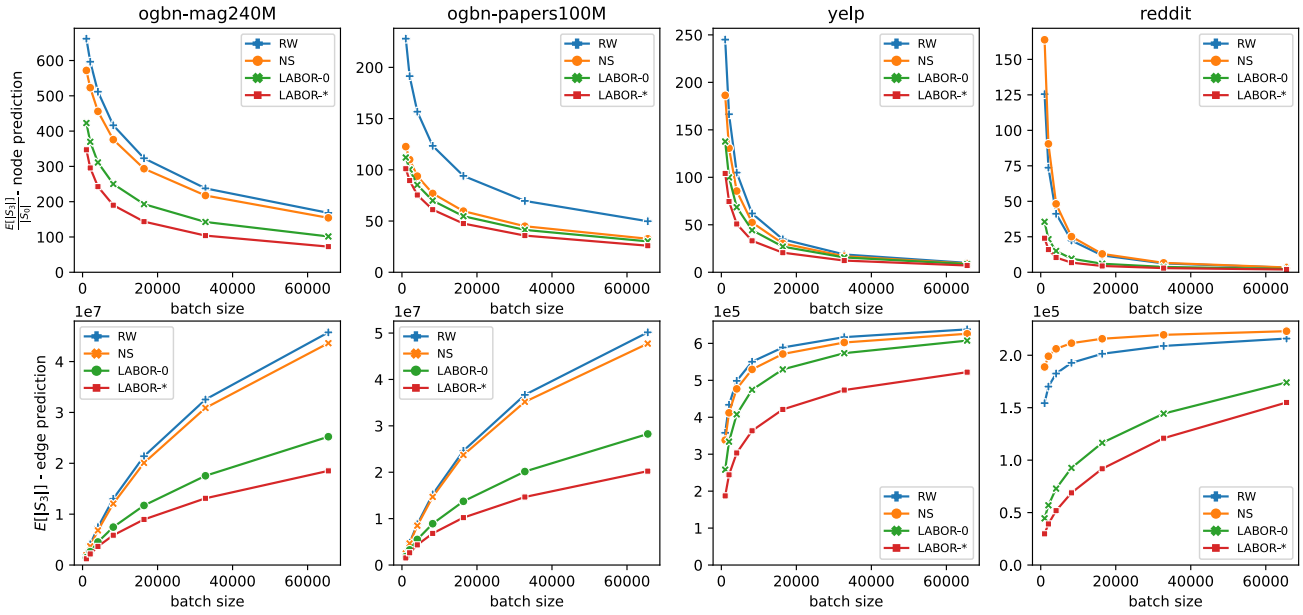


Figure 3. Monotonicity of the work. x-axis shows the batch size, y-axis shows $\frac{E[|S_3|]}{|S_0|}$ for node prediction (top row) and $E[|S_3|]$ for edge prediction (bottom row), where $E[|S_3|]$ denotes the expected number of sampled vertices in the 3rd layer and $|S_0|$ denotes the batch size. RW stands for Random Walks, NS stands for Neighbor Sampling, and LABOR-0/* stand for the two different variants of the LABOR sampling algorithm described in Section 2.2.

decreasing as we have proved in Section 3.1. Not only that, we also see the plot of the expected number of vertices sampled, $E[|S_3|]$, is sublinear and concave with respect to batch size, which we have proved in Section 3.2.

Another observation is that the concavity characteristic of $E[|S_3|]$ seems to differ for different sampling algorithms. If we were to order them in increasing order of concavity, we would have RW, NS, LABOR-0 and LABOR-*. The more concave a sampling algorithm's $E[|S_L|]$ curve is, the less it is affected from the NEP and more savings are available through the use of the proposed methods in Sections 3.3 and 3.4. Note that the differences would grow with larger choice of layer count L.

4.2. Dependent Minibatches

We vary the batch dependency parameter κ introduced in Section 3.4 for the LABOR-0 sampler with a batch size of 1024. Our expectation is that as consecutive batches become more dependent on each other, the subgraphs used during consecutive steps of training would start overlapping with each other, in which case, the vertex embedding accesses would become more localized. We attempted to capture this increase in temporal locality in vertex embedding accesses by implementing an LRU cache to fetch them, the cache sizes used for different datasets is given in Table 1. Note that the cache miss rate is proportional to the amount of data that needs to be copied from the vertex embedding storage. The Figure 4 shows that as κ increases, the cache miss rate across all datasets drop, e.g., from 64% to as low as 16% on reddit. They also show that training is not negatively affected across all datasets up to $\kappa = 256$, after which point the validation F1-score with w/o sampling might slightly start to diverge from the $\kappa = 1$ case. Appendix A.1 has additional discussion about the effect of varing κ.

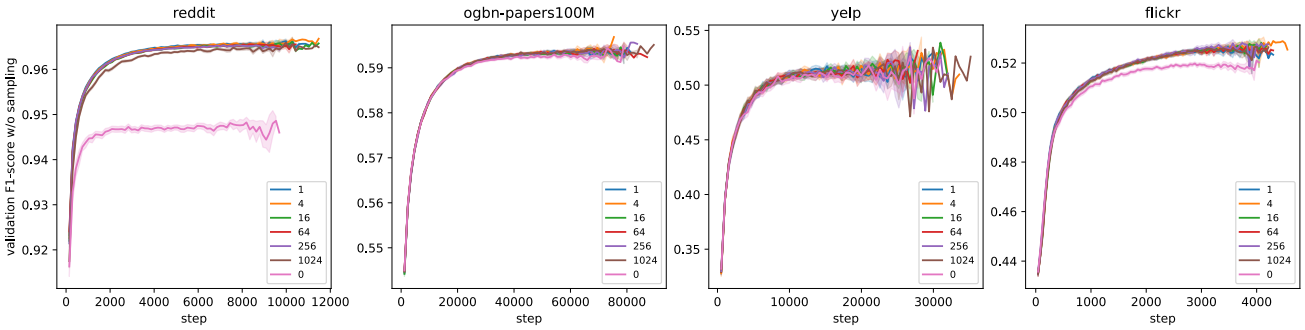


Figure 4. The validation F1-score with the full neighborhoods for LABOR-0 sampling algorithm with 1024 batch size and varying κ dependent minibatches, $\kappa = 0$ denotes infinite dependency, meaning the neighborhood sampled for a vertex stays static during training. See Figure 5 for cache miss rates. See Figure 7 for the validation F1-score with the dependent sampler and the training loss curve.

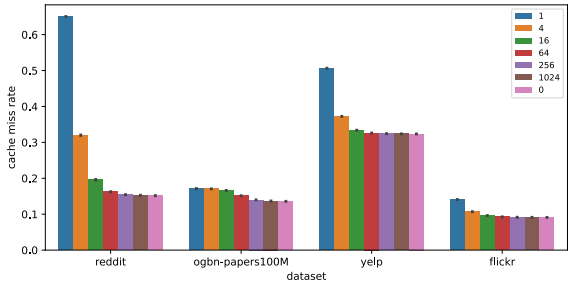


Figure 5. LRU-cache miss rates for LABOR-0 sampling algorithm with 1024 batch size and varying κ dependent minibatches, $\kappa = 0$ denotes infinite dependency.

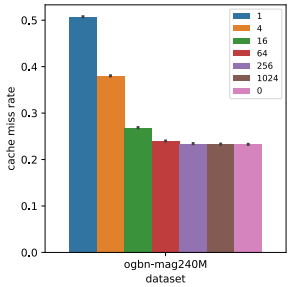


Figure 6. LRU-cache miss rates for LABOR-0 sampling algorithm with 1024 batch size per PE and varying κ dependent minibatches, $\kappa = 0$ denotes infinite dependency.

4.3. Cooperative Minibatching

We use our largest datasets, mag240M, as distributed training is motivated by large scale graph datasets. We present our runtime results on a NVIDIA DGX Station A100 4 GPU machine with NVLink interconnect between the GPUs. We use a batch size of 1024 per PE and the LABOR-0 algorithm as the sampler. In cooperative minibatching, both data size and computational cost is shrinking. To highlight the benefit one could get from reduced computational cost, for the mag240M dataset, we use the R-GAT model (Busbridge et al., 2019), since it is a heterogenous dataset with multiple vertex and edge types. As seen in Table 2, cooperative minibatching reduces the forward/backward pass runtime on mag240M by 22%.

Table 2. Cooperative minibatching forward/backward pass runtimes after minibatches were transferred to the PEs.

Dataset	# PEs	Runtime
mag240M	1	298.56ms
mag240M	4	244.74ms

4.4. Cooperative-Dependent Minibatching

We use the same experimental setup as Section 4.3, but this time vary the κ parameter to show that cooperative minibatching can be used with dependent batches (Figure 6). We use a cache size of 1M per PE. We see that for our largest dataset mag240M, on top $1.4\times$ reduced work coming from cooperative minibatching alone, the cache miss rates were reduced more than $2\times$, making the total improvement $3\times$.

5. Conclusions

In this paper, we investigated a striking difference between DNN and GNN minibatch training. We observed that the cost of processing a minibatch is a sublinear and concave function of batch size in GNNs, unlike DNNs where the cost scales linearly. We then mathematically proved that this is indeed the case for every graph and then proceeded to propose two approaches to take advantage of cost sublinearity. The first approach, which we call cooperative minibatching proposes to partition a minibatch between multiple PEs and process it cooperatively. This is in contrast to existing practice of having independent minibatches per PE, and avoids duplicate work that is a result of vertex and edge repetition across PEs. The second approach proposes the use of consecutive dependent minibatches, through which the temporal locality of vertex and edge accesses is manipulated. As batches get more dependent, the locality increases. We demonstrate this increase in locality by employing an LRU-cache for vertex embeddings. Finally, we show that these approaches can be combined, and greatly benefits large scale GNN training.

References

Allen-Zhu, Z. and Hazan, E. Variance reduction for faster non-convex optimization. In *Proceedings of The 33rd International Conference on Machine Learning*, pp. 699–707. PMLR, 20–22 Jun 2016. URL https://proceedings.mlr.press/v48/allen-zhua16.html.

Artico, I., Smolyarenko, I., Vinciotti, V., and Wit, E. C. How rare are power-law networks really? In *Royal Society*, volume 476, 2020. URL http://doi.org/10.1098/rspa.2019.0742.

Balın, M. F. and Çatalyürek, U. V. (la)yer-neigh(bor) sampling: Defusing neighborhood explosion in gnns. Technical Report arXiv:2210.13339, ArXiv, October 2022. URL http://arxiv.org/abs/2210.13339.

Bertsekas, D. Incremental least squares methods and the extended kalman filter. In *Proceedings of 1994 33rd IEEE Conference on Decision and Control*, volume 2, pp. 1211–1214 vol.2, 1994. doi: 10.1109/CDC.1994.411166.

Busbridge, D., Sherburn, D., Cavallo, P., and Hammerla, N. Y. Relational graph attention networks, 2019.

Dean, J., Corrado, G., Monga, R., Chen, K., Devin, M., Mao, M., Ranzato, M. a., Senior, A., Tucker, P., Yang, K., Le, Q., and Ng, A. Large scale distributed deep networks. In Pereira, F., Burges, C., Bottou, L., and Weinberger, K. (eds.), *Advances in Neural Information Processing Systems*, volume 25. Curran Associates, Inc., 2012. URL https://proceedings.neurips.cc/paper/2012/file/6aca97005c68f1206823815f66102863-Paper.pdf.

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.

Ginsburg, B., Gitman, I., and You, Y. Large batch training of convolutional networks with layer-wise adaptive rate scaling. Technical Report arXiv:1708.03888, ArXiv, September 2017. URL http://arxiv.org/abs/1708.03888.

Goyal, P., Dollár, P., Girshick, R., Noordhuis, P., Wesolowski, L., Kyrola, A., Tulloch, A., Jia, Y., and He, K. Accurate, large minibatch sgd: Training imagenet in 1 hour. Technical Report arXiv:1706.02677, ArXiv, April 2018. URL http://arxiv.org/abs/1706.02677.

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, pp. 1025–1035, 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. *Advances in Neural Information Processing Systems*, 2020-Decem (NeurIPS):1–34, 2020.

Hu, W., Fey, M., Ren, H., Nakata, M., Dong, Y., and Leskovec, J. Ogb-lsc: A large-scale challenge for machine learning on graphs, 2021. URL https://arxiv.org/abs/2103.09430.

Jiang, W. and Luo, J. Graph neural network for traffic forecasting: A survey. *Expert Systems with Applications*, 207: 117921, nov 2022. doi: 10.1016/j.eswa.2022.117921.

Keskar, N. S., Mudigere, D., Nocedal, J., Smelyanskiy, M., and Tang, P. T. P. On large-batch training for deep learning: Generalization gap and sharp minima. *arXiv preprint arXiv:1609.04836*, 2016.

Kingma, D. P. and Ba, J. Adam: A method for stochastic optimization, 2014. URL https://arxiv.org/abs/1412.6980.

Li, M., Zhang, T., Chen, Y., and Smola, A. J. Efficient mini-batch training for stochastic optimization. In *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '14, pp. 661–670, 2014. doi: 10.1145/2623330.2623612.

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, 2020. doi: 10.1145/3419111.3421281.

Liu, Y., Sun, Z., and Zhang, W. Improving fraud detection via hierarchical attention-based graph neural network, 2022.

Nakata, M. and Shimazaki, T. Pubchemqc project: A large-scale first-principles electronic structure database for data-driven chemistry. *Journal of Chemical Information and Modeling*, 57(6):1300–1308, 2017. doi: 10.1021/acs.jcim.7b00083. URL https://doi.org/10.1021/acs.jcim.7b00083. PMID: 28481528.

Narayanan, D., Harlap, A., Phanishayee, A., Seshadri, V., Devanur, N. R., Ganger, G. R., Gibbons, P. B., and Zaharia, M. Pipedream: Generalized pipeline parallelism for dnn training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, pp. 1–15, 2019. doi: 10.1145/3341301.3359646.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Köpf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., and Chintala, S. Pytorch: An imperative style, high-performance deep learning library. In *NeurIPS*, 2019.

Patel, V., Rajasegarar, S., Pan, L., Liu, J., and Zhu, L. Evangcn: Evolving graph deep neural network based anomaly detection in blockchain. In Chen, W., Yao, L., Cai, T., Pan, S., Shen, T., and Li, X. (eds.), *Advanced Data Mining and Applications*, pp. 444–456, 2022.

Tripathy, A., Yelick, K., and Buluç, A. Reducing communication in graph neural network training. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2020. ISBN 9781728199986.

Vitter, J. S. Random sampling with a reservoir. *ACM Trans. Math. Softw.*, 11(1):37–57, mar 1985. ISSN 0098-3500. doi: 10.1145/3147.3165.

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, 2019. URL https://arxiv.org/abs/1909.01315.

Wilson, D. R. and Martinez, T. R. The general inefficiency of batch training for gradient descent learning. *Neural networks*, 16(10):1429–1451, 2003.

Wu, S., Sun, F., Zhang, W., Xie, X., and Cui, B. Graph neural networks in recommender systems: A survey, 2020.

Xu, N., Wang, P., Chen, L., Tao, J., and Zhao, J. MR-GNN: Multi-resolution and dual graph neural network for predicting structured entity interactions. In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence*, aug 2019. doi: 10.24963/ijcai.2019/551.

Ying, R., He, R., Chen, K., Eksombatchai, P., Hamilton, W. L., and Leskovec, J. Graph convolutional neural networks for web-scale recommender systems. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '18, pp. 974–983, 2018. doi: 10.1145/3219819.3219890.

Zeng, H., Zhou, H., Srivastava, A., Kannan, R., and Prasanna, V. GraphSAINT: Graph sampling based inductive learning method. In *International Conference on Learning Representations*, 2020. URL https://openreview.net/forum?id=BJe8pkHFwS.

Zheng, D., Song, X., Yang, C., LaSalle, D., Su, Q., Wang, M., Ma, C., and Karypis, G. Distributed hybrid cpu and gpu training for graph neural networks on billion-scale graphs. *arXiv preprint arXiv:2112.15345*, 2021.

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, aug 2019. doi: 10.14778/3352063.3352127.

A. Appendix

A.1. Dependent batches (cont.)

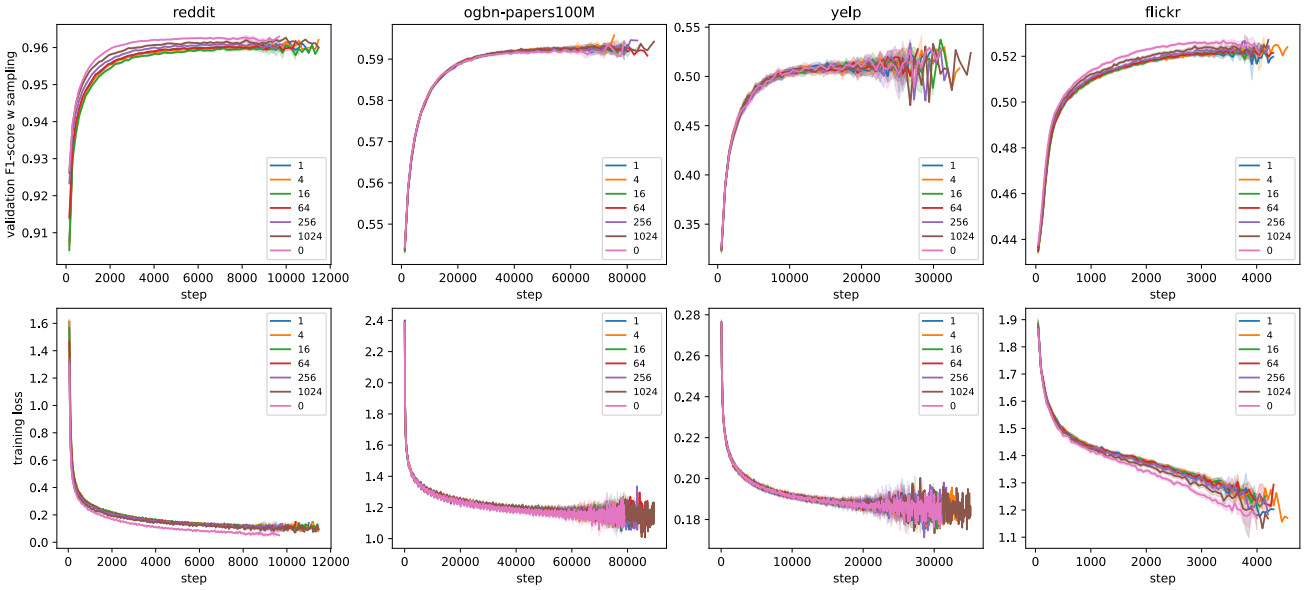


Figure 7. LABOR-0 sampling algorithm with 1024 batch size and varying κ dependent minibatches, $\kappa = 0$ denotes infinite dependency, meaning the neighborhood sampled for a vertex stays static during training. The first row shows the validation F1-score with the dependent sampler. The second row shows the training loss curve. Completesfor Figure 4.

Looking at the training loss and validation F1-score with sampling curves in Figure 7, we notice that the performance gets better as κ is increased. This is due to the fact that a vertex's neighborhood changes slower and slower as κ is increased, the limiting case being $\kappa = 0$, in which case the sampled neighborhoods are unchanging. This makes training easier so $\kappa = 0$ case leads the pack in the training loss and validation F1- score with sampling curves.

A.2. Comparing a single batch vs P independent batches convergence

We investigate whether training with a single large batch in P-GPU training shows any convergence differences to the current approach of using P separate batches for each of the GPUs. We use a global batch size of 4096 and divide a batch into $P \leq 8$ independent batches, with each batch having a size of $\frac{4096}{P}$. We use NS and LABOR-0 samplers with fanouts of $k = 10$ for each of the hoods 3 layers. Figure 8 shows that there are no visible differences between the two approaches, we present the results averaged over the samplers to save space.

A.3. Monotonicity of work (cont.)

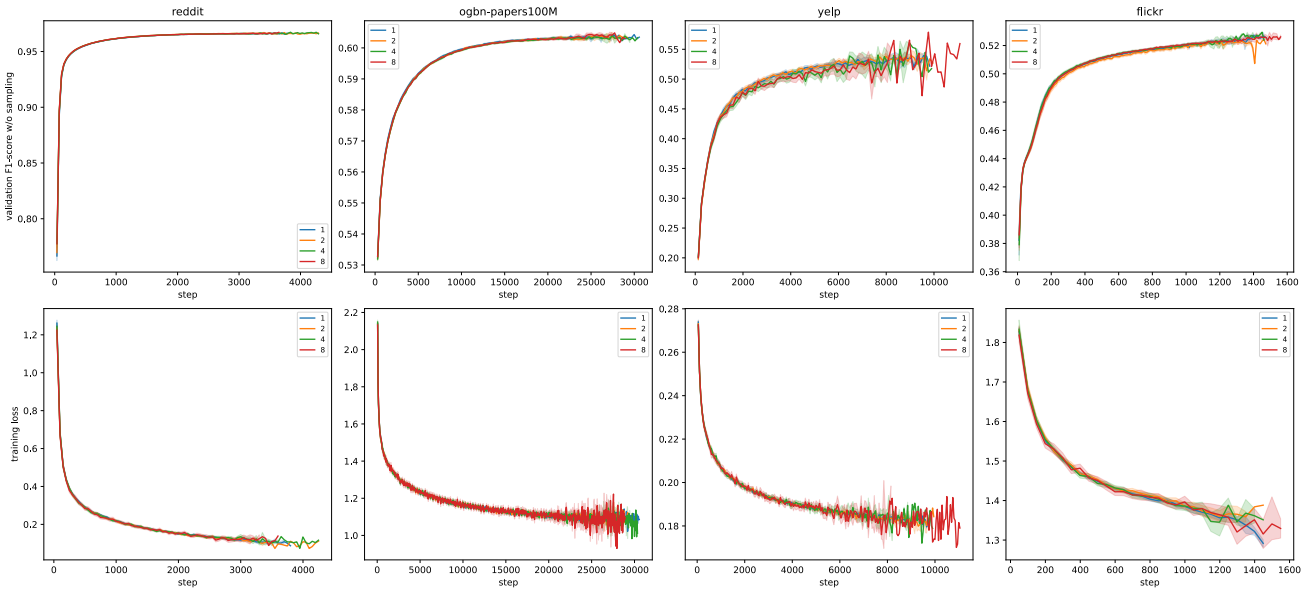


Figure 8. Convergence difference between cooperative vs independent minibatching with a global batch size of 4096 averaged over Neighbor and LABOR-0 samplers.

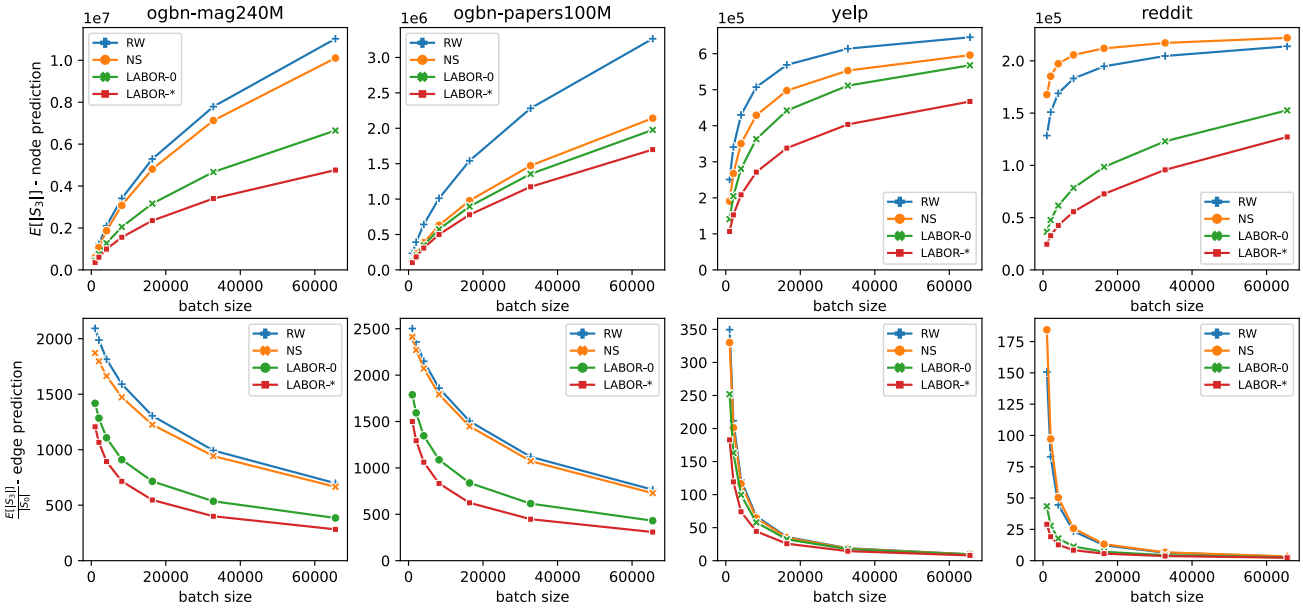


Figure 9. Monotonicity of the work. x axis shows the batch size, y axis shows $E[|S_3|]$ for node prediction (top row) and $\frac{E[|S_3|]}{|S_0|}$ for edge prediction (bottom row), where $E[|S_3|]$ denotes the expected number of vertices sampled in the 3rd layer and $|S_0|$ denotes the batch size. RW stands for Random Walks, NS stands for Neighbor Sampling, and LABOR-0/* stand for the two different variants of the LABOR sampling algorithm described in Section 2.2.