Graph neural networks (GNN) are compute intensive; thus, they are attractive for acceleration on distributed platforms. We present DeepGalois, an efficient GNN framework targeting distributed CPUs. DeepGalois is designed for efficient communication of high-dimensional feature vectors used in GNN. The graph partitioning engine flexibly supports different partitioning policies and helps the user make tradeoffs among task division, memory usage, and communication overhead, leading to fast feature learning without compromising the accuracy. The communication engine minimizes communication overhead by exploiting partitioning invariants and communication bandwidth in modern clusters. Evaluation on a production cluster for the representative reddit and ogbn-products datasets demonstrates that DeepGalois on 32 machines is 2.5× and 2.3× faster than that on 1 machine in average epoch time and time to accuracy, respectively. On 32 machines, DeepGalois outperforms DistDGL by 4× and 8.9× in average epoch time and time to accuracy, respectively.

This work makes the following contributions:

• We present DeepGalois, a scalable GNN framework targeting distributed CPU clusters.
• We provide system support for efficient feature vector communication and flexible graph partitioning to improve scalability.
• Experimental results on up to 32 CPU machines demonstrate that DeepGalois scales well and outperforms the state-of-the-art.

2 BACKGROUND

2.1 Computation Patterns in Graph Neural Networks

Given an input graph $G(V, E)$ with vertices $V$ and edges $E$ and input features $h^0_v$ for each $v \in V$, a $k$-layer GNN learns an output feature vector $h^k_v$ for each vertex $v$ that can
be used for downstream tasks. In the i-th layer (0 \leq i < k), each vertex aggregates feature vectors from its neighbors and updates itself with the aggregated neighbor information and its own feature vector \( h^i_v \), to get its feature for layer \( i + 1 \), \( h^{i+1}_v \). The computation for vertex \( v \) in the l-th layer follows:

\[
\begin{align*}
    m^{l+1}_v &= \text{aggregate}(\{ h^l_u | u \in \mathcal{N}(v) \}) & (1) \\
    h^{l+1}_v &= \text{update}(m^{l+1}_v, h^l_v) & (2)
\end{align*}
\]

\( \mathcal{N}(v) \) is the set of \( v \)'s one-hop neighbors. Both functions can use trainable parameters that are learned during GNN training: for example, the update function typically has an associated set of weights that are used to transform the aggregated feature vector using a matrix multiply operation.

Many GNN models have been proposed, including graph convolutional networks (GCN) (Kipf & Welling, 2016), Graph Isomorphism Networks (GIN) (Xu et al., 2019), GraphSAGE (Hamilton et al., 2017), and graph attention networks (GAT) (Veličković et al., 2018). They differ in how they define the GNN’s aggregate and update functions.

2.2 Existing Graph Neural Network Work

Many techniques exist to support scalable GNN computing. FastGCN (Chen et al., 2018) and GraphSAGE (Hamilton et al., 2017) proposed mini-batching and neighbor sampling to reduce computation and memory footprints. LGCL (Gao et al., 2018) and GraphSAINT (Zeng et al., 2019; Zeng et al., 2020) proposed subgraph sampling which trains GNNs via random-walked sampled subgraphs. GPNN (Liao et al., 2018) adopts two-level (local-global) propagation approach to handle extremely large graphs. Other techniques remove redundant aggregation (Zeng & Prasanna, 2020; Jia et al., 2020b) from common neighbors.

These techniques can be applied on top of distributed CPU/GPU GNN implementations to further accelerate training. Distributed CPU systems include AltGraph (Yang, 2019), Euler (Eul, 2020), DistDGL (Zheng et al., 2020), and AGL (Zhang et al., 2020), and multi-GPU systems include Roc (Jia et al., 2020a), NeuGraph (Ma et al., 2019), and CAGNET (Tripathy et al., 2020). Each system has varying methods of distribution of the GNN computation.

2.3 Distributed Vertex Programs

In graph analytics applications, a computation rule called an operator (Pingali et al., 2011) is applied repeatedly to a vertex \( v \) and its label until some termination condition is reached. The operator uses \( v \)'s label as well as the labels of \( v \)'s neighborhood, which the operator can define arbitrarily (e.g., beyond single-hop neighbors). A vertex program is an application where the neighborhood of an operator applied to vertex \( v \) consists only of \( v \)'s immediate neighbors. Many distributed graph analytics systems have been proposed to efficiently distribute the computation of vertex programs across machines such as D-Galois (Dathathri et al., 2018), Lux (Jia et al., 2017), and Gemini (Zhu et al., 2016). In particular, D-Galois is composed of two components that can be reused in any system to distribute vertex programs: the Gluon communication substrate which optimizes communication across machines using structural and temporal invariants of partitioning and the Customizable Streaming Partitioner (CuSP) (Hoang et al., 2019), a library which partitions the graph.

We observe that GNN computation is a vertex program: aggregation followed by update is an operator that reads features (i.e., labels) from immediate neighbors and applies the update function update its own label. The termination condition is the number of layers in the network: this operator will be applied \( k \) times for a \( k \)-layer network.

3 DeepGalois

3.1 Components of DeepGalois

Graph Computation: For computation on each machine, DeepGalois uses Galois (Nguyen et al., 2013) which is the state-of-the-art shared memory framework that provides parallel constructs and efficient abstract data structures for graph computations. For matrix multiplications required in GNN computations, DeepGalois uses the BLAS library. DeepGalois’s modular design allows users to use any such library of their choice.

Graph Partitioning: DeepGalois uses a customizable streaming graph partitioner called CuSP (Hoang et al., 2019) that provides a simple API to design custom partitioning policies for graphs. CuSP allows the flexibility to customize domain specific partitioning policies for GNN applications; this is especially important for efficient communication (Section 3.3). CuSP provides many graph partitioning policies including edge-cuts, vertex-cuts, and hybrid-cuts, so users of DeepGalois can pick any of these policies for their specific GNN, dataset, and cluster (Gill et al., 2018).

Synchronization: Finally, for synchronization of vertex labels (aggregated features in the case of GNNs) among machines in the distributed setting, DeepGalois leverages Gluon (Dathathri et al., 2018) which is a partition-aware communication substrate for vertex programs.

The combination of these three components has been shown useful in the past in D-Galois (Dathathri et al., 2018), a state-of-the-art distributed graph analytics framework; DeepGalois applies it to the machine learning domain in this work which brings its own set of design challenges.
Abstractly, all partitioning policies used by DeepGalois can be expressed with the definition of two heuristics (Hoang et al., 2019): (1) assignment of master proxies and (2) assignment of edges to machines. For example, a simple way to assign master proxies is to split them evenly among all machines in a blocked fashion. An outgoing edge-cut (OEC) would assign edge \((u, v)\) to the machine that has \(v\)'s master proxy: this results in a partition where the machine with the master proxy ends up with all its outgoing edges. Another example of a more complex policy is a 2D Cartesian vertex cut (CVC) (Boman et al., 2013) which distributes edges according to a 2D block-cyclic distribution of the blocks of the adjacency matrix representing the edges of the graph. Fig. 1 illustrates OEC and CVC. The way the graph is partitioned affects computational load balance as well as the communication patterns during synchronization.

DeepGalois is the first distributed GNN implementation to allow for arbitrary partitioning of the graph via CuSP: this allows us to use partitions that scale better than the edge-cuts used in most distributed GNN systems (e.g., DistDGL) as we illustrate in our experimental results.

### 3.3 Synchronization of Data

**Aggregation** After the graph is partitioned among machines, each partition is a self-contained subgraph with local proxies. The local application of the aggregation function on a given machine performs local aggregation for vertices in the partition on that machine. This does not involve any communication due to the cached copies in the local proxies. In other distributed GNN frameworks like DistDGL (Zheng et al., 2020), vertex features are queried from a server if they are not present locally, so server communication may be required to access vertex features. DeepGalois reduces the communication overhead by performing local aggregation without communication.

To get the full aggregated result after local aggregation, a logical all-reduce operation is required in which all partial values on proxies of a given vertex \(v\) are synchronized. This requires that the aggregation operation must be commutative and associative; most GNN aggregation functions satisfy this property. Synchronization using Gluon is a logical reduction on the master proxy to produce the final canonical value by reducing the contributions from mirror proxies followed by a logical broadcast of the canonical value from the master to mirror proxies. As they may be millions of vertices whose partial values may need to be synchronized,
DeepGalois uses Gluon to combine partial values (or messages) for different vertices into a single message per host in each logical reduction or broadcast phases. Such message aggregation reduces the communication overhead by better utilizing the communication bandwidth (Dang et al., 2018).

It is important to note that the partitioning policy plays a key role in determining the communication volume during synchronization. Reduction will occur from machine $a$ to machine $b$ if $a$ has a proxy for a vertex $v$ that host $b$ owns the master proxy of, and a broadcast occurs from $b$ to all machines that have a proxy for $v$. It is possible for $a$ to need to broadcast to all machines in the system if a proxy exists for a vertex it owns on every machine: this does not scale well as the number of machines increases. Therefore, to avoid this problem, it is useful to use a 2D partitioning policy such as CVC which is more structured: due to the way edges are placed in CVC, a machine’s communication partners are strictly limited to only a subset of hosts, which allows communication to scale for a higher number of hosts. These communication optimizations based on partitioning policies have been detailed and studied extensively (Datathathri et al., 2018; Gill et al., 2018), and we leverage these findings here for efficient GNN synchronization.

### Model Gradients
During the backward pass of GNN training, each machine calculates only the gradient contributions of its master proxies. A simple sum all-reduce of all gradients among all machines (as all gradients are replicated on all machines) reconstructs the same gradients that would have been derived on a single machine. This ensures that distributed execution does not degrade accuracy.

## 4 Evaluation

### 4.1 Experimental Setup
We use two platforms for evaluation. The first is a machine with Intel Xeon Gold 5120 2.2 GHz CPUs with 56 cores on 4 sockets and 187GB of DRAM. We refer to this machine as Ghostwheel (GW). The second is the Stampede2 supercomputer (Stanzione et al., 2017) at the Texas Advanced Computing Center (TAC); each machine has 48 Intel SkyLake cores on 2 sockets and 192GB of DRAM. The machines are connected via a 100 Gb/s Intel Omni-Path network. We run on up to 32 of these machines.

We evaluate three GNN systems. Deep Graph Library (DGL) (Wang et al., 2019) is a library that provides many GNN models as well as data structures to easily build new models. DistDGL (Zheng et al., 2020) is the distributed variant of DGL: it has a subset of the models in DGL. DeepGalois is the system presented in this paper. For our experiments, we evaluate the Graph Convolutional Network model (Kipf & Welling, 2016) and GraphSAGE model (Hamilton et al., 2017) of GNNs on these systems (DistDGL does not have GCN). We evaluate DGL only on Ghostwheel, and we evaluate DistDGL only on Stampede2. We perform vertex classification for these datasets, and accuracy is for single-label vertex classification.

Table 1 shows the input graphs we use for evaluation. `reddit` is a representative graph used in previous works (Jia et al., 2020a; Wang et al., 2019), and the other two datasets are from Open Graph Benchmark (OGB) (Hu et al., 2020a). DistDGL supports sampling, which significantly changes computation for large graphs such as ogbn-papers100M, so we do not include DistDGL results for it here; we use this graph only to show scaling for DeepGalois. We plan to add support for sampling in DeepGalois in the future. We use a 2-layer GNN with a hidden feature dimension size of 16. Feature dropout and mini-batching are disabled; note that DistDGL is optimized for mini-batching, but it is disabled in order to examine full-batch behavior compared to DeepGalois. We average runtimes over 200 epochs except for ogbn-papers100M where we only run for 5 epochs. More experimental setup details are in the appendix.

### 4.2 Single Machine: DGL and DeepGalois
To show that DeepGalois matches a baseline for performance, we compare runtime with DGL on single-machine. Table 2 shows average epoch time of running 200 epochs. DeepGalois is competitive with DGL, showing that it can match the performance of existing GNN implementations.

Note that DeepGalois is much faster than DGL for GraphSAGE. This is due to an important optimization which or-ders aggregation and update depending on the dimensions of input embedding $D_i$ and output embedding $D_o$. When $D_i > D_o$, update is performed before aggregation. Otherwise, aggregation is performed before update. In this way,
aggregation (which is more expensive than update) is always performed on the shorter embeddings. In DGL, this optimization is applied to GCN but not to GraphSAGE because GraphSAGE must support multiple different aggregators, some of which (e.g., LSTM) prevent this optimization from being applied. This can be fixed in DGL by checking the constraint compile time to see if this optimization is applicable. For example, with a mean aggregator, it can be applied.

4.3 Distributed Execution: DistDGL vs. DeepGalois

We compare DeepGalois with the state-of-the-art distributed GNN system, DistDGL. Unless specified otherwise, we report results of DeepGalois using CVC partitioning policy, which performs better than OEC.

4.3.1 Performance Overview

Fig. 2 compares average training epoch time of DistDGL and DeepGalois on 1-machine to 32-machine for GraphSAGE. We observe that for **reddit** and **products**, DeepGalois is constantly faster than full-batch DistDGL even though DistDGL does neighbor sampling to reduce computation, likely due to DeepGalois avoiding the data copying over the network done by DistDGL. Another observation is that DistDGL does not scale well, particularly for **reddit**. With 32-machine, DistDGL is 2.6× slower than it is on single-machine, which defeats the purpose of using a distributed cluster. This is also likely due to the significant data copying overhead, which would be non-trivial especially medium-sized graphs like **reddit** where the reduction of computation time can not amortize the communication overhead. In comparison, DeepGalois achieves 2.3× speedup with 32-machine over 1-machine for **reddit**.

4.3.2 Performance Breakdown

Fig. 3a and Fig. 3b compares average epoch time distribution of DistDGL and DeepGalois on different number of machines. We observe that there is a non-trivial overhead in DistDGL to sample and copy the data. For DeepGalois, “other” includes the communication time.
Efficient Distribution for Deep Learning

Fig. 4. OEC vs. CVC partitioning policy in DeepGalois using ogbn-papers100M on 8-, 16-, 32-machines.

Fig. 5. Speed of convergence for DeepGalois and DistDGL on 1-machine and 32-machine using products. Each two consecutive points in the figure cover 10 epochs.

Fig. 4 illustrates the time distribution of DeepGalois on the large graph papers-100M with OEC and CVC. Due to the limited memory capacity on each machine, we can only run it with at least 8 machines. We observe that DeepGalois scales well to 32 machines with CVC. Computation time is reduced due to additional compute power. Notice that the communication time also decreases as number of machines increases: even though the total amount of communication increases, communication is overlapped with more machines. OEC has higher computation time than CVC due to load imbalance since OEC assigns all edges of a vertex to the same machine: this may not balance the workload well due to the high-degree vertices. Overall, CVC executes for less time, which is why we choose to use it in practice.

Fig. 4 shows that the graph partitioning policy matters: the flexibility provided by CuSP is beneficial for the training speed. It is more important for GNN applications than it is for graph analytics applications to explore the best graph partitioning policy, as there is much more computation and communication involved in GNN due to vector in GNN instead of scalar in graph analytics. Therefore, carefully balancing workload and reducing communication overhead is critical to the training speed.

4.4 Accuracy Convergence Speed

Fig. 5 shows how the test accuracy changes over time for DistDGL and DeepGalois. We notice that with more machines, both DistDGL and DeepGalois converge faster than single-machine due to reduced epoch time. We observe that DeepGalois converges faster than DistDGL. There are two reasons. First, with the same number of epochs, DeepGalois’s accuracy increases faster, likely because DistDGL uses neighbor sampling but DeepGalois does not. Second, since DeepGalois runs faster than DistDGL, the convergence time is reduced.

5 Conclusion

This paper presents DeepGalois, a distributed GNN system built using proven distributed graph analytics systems and techniques applied to GNNs. DeepGalois scales as the number of machines grows due to efficient communication that leverages invariants of partitioning policies, and it outperforms DistDGL, a state-of-the-art distributed GNN system. We are in the process of adding GPU support to DeepGalois, and we also plan to add additional layer types such as GAT and techniques such as graph sampling.

Acknowledgements

This research was supported by the NSF grants 1406355, 1618425, 1705092, and 1725322, DARPA contracts FA8750-16-2-0004 and FA8650-15-C-7563, and XSEDE grant ACI-1548562 through allocation TG-CIE-170005.

References

Texas Advanced Computing Center (TACC) at the University of Texas at Austin. URL http://www.tacc.utexas.edu.

Efficient Distribution for Deep Learning

alibaba/euler.


Efficient Distribution for Deep Learning


A MORE EXPERIMENTAL SETUP DETAILS

This has been updated from the original version present on the GNNSys21 webpage: learning rates have been corrected and DGL details have been added.

The test accuracy used to determine the time-to-accuracy speedups in the abstract and intro was 93% for reddit and 60% for ogbn-products.

During partitioning for DeepGalois, we modified CuSP to balance training nodes among machines for reddit and ogbn-products, which had well-defined ranges for training nodes.

We disabled all minibatching in DistDGL in order to do a fair comparison with DeepGalois which does not have minibatching. DistDGL has neighborhood sampling (we could not disable this easily) and inductive training. DeepGalois does not do sampling and does transductive training. The Adam optimizer is used in all three systems. DistDGL and DeepGalois use beta1 0.9, beta2 0.999, and epsilon $1 \times 10^{-8}$ for Adam. DeepGalois uses 0.01 for learning rate while DistDGL uses 0.003.

We used/modified code as necessary to do full batch training with DGL. Weight decay is disabled for DGL’s AdamW, and AdamW’s parameters for AdamW are default other than learning rate. Learning rate of 0.01 is used for DGL except for reddit with GCN which uses a learning rate of 0.02. DGL’s GCN by default has a linear layer after GCN layers. Dropout is disabled.

The GitHub commit used for DistDGL runs is e22087aa36f7f6b331f3edbf1e5b14850c92e3 (March 25, 2021) with DGL 0.6. For DistDGL, we used the following hyper-parameters. Number of trainers was set to 1 (setting it higher seemed to slow down execution). Number of servers was set to 2 (we did not notice much difference varying this number). Minibatching was disabled as mentioned above by setting the batch size to a very high number. The profiler was disabled. The number of workers (equivalent to number of samplers) parameter was set to 0: this was to avoid a bug in which evaluation of the validation and testing set would not work. OMP_NUM_THREADS was set to 48 (the number of threads on the cluster it was run on). We used the balance_train and balance_edges arguments to the graph partitioner, and the number of partitions created was equal to the number of machines used.