Dorylus: Affordable, Scalable, and Accurate GNN Training with Distributed CPU Servers and Serverless Threads

John Thorpe[†♣] Yifan Qiao[†♣] Jonathan Eyolfson[†] Shen Teng[†] Guanzhou Hu[†‡] Zhihao Jia[§]
Jinliang Wei[*] Keval Vora[♭] Ravi Netravali[♯] Miryung Kim[†] Guoqing Harry Xu[†]
UCLA[†] *University of Wisconsin*[‡] *CMU*[§] *Google Brain*[*] *Simon Fraser*[♭] *Princeton University*[♯]

Abstract

A graph neural network (GNN) enables deep learning on structured graph data. There are two major GNN training obstacles: 1) it relies on high-end servers with many GPUs which are expensive to purchase and maintain, and 2) limited memory on GPUs cannot scale to today's billion-edge graphs. This paper presents Dorylus: a distributed system for training GNNs. Uniquely, Dorylus can take advantage of *serverless computing* to increase scalability at a low cost.

The key insight guiding our design is *computation separation*. Computation separation makes it possible to construct a *deep, bounded-asynchronous* pipeline where graph and tensor parallel tasks can fully overlap, effectively hiding the network latency incurred by Lambdas. With the help of thousands of Lambda threads, Dorylus scales GNN training to billion-edge graphs. Currently, for large graphs, CPU servers offer the best performance per dollar over GPU servers. Just using Lambdas on top of Dorylus offers up to 2.75× more performance-per-dollar than CPU-only servers. Concretely, Dorylus is 1.22× faster and 4.83× cheaper than GPU servers for massive sparse graphs. Dorylus is up to 3.8× faster and 10.7× cheaper compared to existing sampling-based systems.

1 Introduction

Graph Neural Networks (GNN) [40, 55, 71, 50, 53, 35] are a family of NNs designed for deep learning on graph structured data [103, 92]. The most well-known model in this family is the graph convolutional network (GCN) [40], which uses the connectivity structure of the graph as the filter to perform neighborhood mixing. Other models include graph recursive network (GRN) [51, 67], graph attention network (GAT) [6, 78, 100], and graph transformer network (GTN) [96]. Due to the prevalence of graph datasets, GNNs have gained increasing popularity across diverse domains such as drug discovery [85], chemistry [19], program analysis [2, 5], and recommendation systems [91, 95]. In fact, GNN is one of the most popular topics in recent AI/ML conferences [32, 45].

♣ Contributed equally.

GPUs are the de facto platform to train a GNN due to their ability to provide highly-parallel computations. While GPUs offer great efficiency for training, they (and their host machines) are expensive to use. To train a (small) million-edge graph, recent works such as NeuGraph [55] and Roc [34] need at least four such machines. A public cloud offers flexible pricing options, but cloud GPU instances still incur a non-trivial cost — the lowest-configured p3 instance type on AWS has a price of $3.06/h; training realistic models requires dozens/hundreds of such machines to work 24/7. While cost is not a concern for big tech firms, it can place a heavy financial burden on small businesses and organizations.

In addition to being expensive, GPUs have limited memory, hindering *scalability*. For context, real-world graphs are routinely *billion-edge* scale [69] and continue to grow [95]. NeuGraph and Roc enable coordinated use of multiple GPUs to improve scalability (at higher costs), but they remain unable to handle the billion-edge graphs that are commonplace today. Two main approaches exist for reducing the costs and improving the scalability of GNN training, but they each introduce new drawbacks:

- CPUs face far looser memory restrictions than GPUs, and operate at significantly lower costs. However, CPUs are unable to provide the parallelism in computations that GPUs can, and thus deliver far inferior *efficiency* (or speed).
- Graph sampling techniques select certain vertices and sample their neighbors when gathering data [25, 95]. Sampling techniques improve scalability by considering less graph data, and it is a generic technique that can be used on either GPU or CPU platforms. However, our experiments (§7.5) and prior work [34] highlight two limitations with graph sampling: (1) sampling must be done repeatedly per epoch, incurring time overheads and (2) sampling typically reduces *accuracy* of the trained GNNs. Furthermore, although sampling-based training converges often in practice, there is no convergence guarantee for trivial sampling methods [9].

Affordable, Scalable, and Accurate GNN Training. This

paper devises a *low-cost* training framework for GNNs on *billion-edge graphs*. Our goal is to simultaneously deliver high efficiency (*e.g.*, close to GPUs) and high accuracy (*e.g.*, higher than sampling). Scaling to billion-edge graphs is crucial for applicability to real-world use cases. Ensuring low costs and practical performance improves the accessibility for small organizations and domain experts to make the most out of their rich graph data.

To achieve these goals, we turn to the *serverless computing* paradigm, which has gained increasing traction [20, 43, 37] in recent years through platforms such as AWS Lambda, Google Cloud Functions, or Azure Functions. Serverless computing provides large numbers of parallel "cloud function" threads, or *Lambdas*, at an extremely low price (*i.e.*, $0.20 for launching one million threads on AWS [3]). Furthermore, Lambda presents a pay-only-for-what-you-use model, which is much more appealing than dedicated servers for applications that need only massive parallelism.

Although it appears that serverless threads could be used to complement CPU servers without significantly increasing costs, they were built to execute light asynchronous tasks, presenting two challenges for NN training:

- *Limited compute resources* (*e.g.*, 2 weak vCPUs)
- *Restricted network resources* (*e.g.*, 200 Mbps between Lambda servers and standard EC2 servers [42])

A neural network makes heavy use of (linear algebra based) tensor kernels. A Lambda[1] thread is often too weak to execute a tensor kernel on large data; breaking the data to tiny minibatches mitigates the compute problem at the cost of higher data-transfer overheads. Consequently, using Lambdas naïvely for training an NN could result in significant slowdowns (*e.g.*, 21× slowdowns for training of multi-layer perceptron NNs [29], even compared to CPUs).

Dorylus. To overcome these weaknesses, we developed Dorylus[2], a distributed system that uses cheap CPU servers and serverless threads to achieve the aforementioned goals for GNN training. Dorylus leverages GNN's special computation model to overcome the two challenges associated with the use of Lambdas. Details are elaborated below:

The *first* challenge is *how to make computation fit into Lambda's weak compute profile?* We observed: *not all operations in GNN training need Lambda's parallelism.* GNN training comprises of two classes of tasks [55] – neighbor propagations (*e.g.*, `Gather` and `Scatter`) over the input graph and per-vertex/edge NN operations (such as `Apply`) over the tensor data (*e.g.*, features and parameters). Training a GNN over a large graph is dominated by *graph computation* (see §7.6), not *tensor computation* that exhibits strong SIMD behaviors and benefits the most from massive parallelism.

Based on this observation, we divide a training pipeline into a set of *fine-grained tasks* (Figure 3, §4) based on the type of data they process. Tasks that operate over the graph structure belong to a *graph-parallel path*, executed by CPU instances, while those that process tensor data are in a *tensor-parallel path*, executed by Lambdas. Since the graph structure is taken out of tensors (*i.e.*, it is no longer represented as a matrix), the amount of tensor data and computation can be significantly reduced, providing an opportunity for each tensor-parallel task to run a *lightweight linear algebra operation* on *a data chunk of a small size* — a granularity that a Lambda is capable of executing quickly.

Note that Lambdas are a perfect fit to GNNs' tensor computations. While one could also employ regular CPU instances for compute, using such instances would incur a much higher monetary cost to provide the same level of burst parallelism (*e.g.*, **2.2**× in our experiments) since users not only pay for the compute but also other unneeded resources (*e.g.*, storage).

The *second* challenge is *how to minimize the negative impact of Lambda's network latency?* Our experiments show that Lambdas can spend one-third of their time on communication. To not let communication bottleneck training, Dorylus employs a novel parallel computation model, referred to as *bounded pipeline asynchronous computation* (BPAC). BPAC makes full use of *pipelining* where different fine-grained tasks overlap with each other, *e.g.*, when graph-parallel tasks process graph data on CPUs, tensor-parallel tasks process tensor data, simultaneously, with Lambdas. Although pipelining has been used in prior work [34, 63], in the setting of GNN training, pipelining would be impossible without fine-grained tasks, which are, in turn, enabled by computation separation.

To further reduce the wait time between tasks, BPAC incorporates *asynchrony* into the pipeline so that a fast task does not have to wait until a slow task finishes even if data dependencies exist between them. Although asynchronous processing has been widely used in the past, Dorylus faces a unique technical difficulty that no other systems have dealt with: as Dorylus has two computation paths, where exactly should asynchrony be introduced?

Dorylus uses asynchrony in a novel way at two distinct locations where staleness can be tolerated: *parameter updates* (in the tensor-parallel path) and *data gathering from neighbor vertices* (in the graph-parallel path). To not let asynchrony slow down the convergence, Dorylus *bounds* the degree of asynchrony at each location using different approaches (§5): *weight stashing* [63] at parameter updates and *bounded staleness* at data gathering. We have formally proved the convergence of our asynchronous model in §5.

Results. We have implemented two popular GNNs – GCN and GAT – on Dorylus and trained them over four real-world graphs: `Friendster` (3.6B edges), `Reddit-full` (1.3B), `Amazon` (313.9M), and `Reddit-small` (114.8M). With the help of 32 graph servers and thousands of Lambda threads, Dorylus was able to train a GCN, *for the first time without sampling*, over billion-edge graphs such as `Friendster`.

[1] We use "Lambda" in this paper due to our AWS-based implementation while our idea is generally applicable to all types of serverless threads.

[2] Dorylus is a genus of army ants that form large marching columns.

To enable direct comparisons among different platforms, we built new GPU- and CPU-based training backends based on Dorylus' distributed architecture (with computation separation). Across our graphs, Dorylus's performance is **2.05**× and **1.83**× higher than that of GPU-only and CPU-only servers *under the same monetary budget*. Sampling is surprisingly slow — to reach the same accuracy target, it is **2.62**× slower than Dorylus due to its slow accuracy climbing. In terms of accuracy, Dorylus can train a model with an accuracy **1.05**× higher than sampling-based techniques.

Key Takeaway. Prior work has demonstrated that Lambdas can only achieve suboptimal performance for DNN training due to the limited compute resources on a Lambda and the extra overheads to transfer model parameters/gradients between Lambdas. Through computation separation, Dorylus makes it possible, *for the first time*, for Lambdas to provide a scalable, efficient, and low-cost distributed computing scheme for GNN training.

Dorylus is useful in two scenarios. First, for small organizations that have tight cost constraints, Dorylus provides an affordable solution by exploiting Lambdas at an extremely low price. Second, for those who need to train GNNs on very large graphs, Dorylus provides a scalable solution that supports fast and accurate GNN training on billion-edge graphs.

2 Background

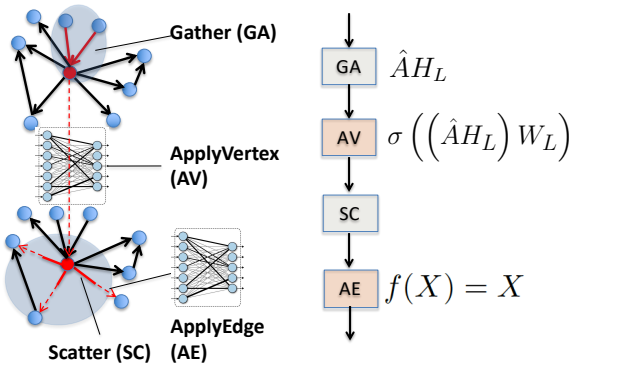
A GNN takes graph-structured data as input, where each vertex is associated with a feature vector, and outputs a feature vector for each individual vertex or the whole graph. The output feature vectors can then be used by various downstream tasks, such as, graph or vertex classification. By combining the feature vectors and the graph structure, GNNs are able to learn the patterns and relationships among the data, rather than relying solely on the features of a single data point.

GNN training combines graph propagation (*e.g.*, `Gather` and `Scatter`) and NN computations. Prior work [17, 89] discovered that GNN development can be made much easier with a programming model that provides a *graph-parallel* interface, which allows programmers to develop the NN with familiar graph operations. A typical example is the deep graph library (DGL) [17], which unifies a variety of GNN models with a common GAS-like interface.

Forward Pass. To illustrate, consider graph convolutional network (GCN) as an example. GCN is the simplest and yet most popular model in the GNN family, with the following forward propagation rule for the L-th layer [40]:

$$(R1) \qquad H_{L+1} = \sigma(\hat{A}H_L W_L)$$

A is the adjacency matrix of the input graph, and $\tilde{A} = A + I_N$ is the adjacency matrix with self-loops constructed by adding A with I_N, the identity matrix. $\tilde{D}$ is a diagonal matrix such that $\tilde{D}_{ii} = \Sigma_j \tilde{A}_{ij}$. With $\tilde{D}$, we can construct a *normalized adjacency matrix*, represented by $\hat{A} = \tilde{D}^{-\frac{1}{2}}\tilde{A}\tilde{D}^{-\frac{1}{2}}$. W_L is a layer-specific trainable weight matrix. $\sigma(.)$ denotes a non-linear activation function, such as



(a) Computation model (b) Annotated dataflow graph

Figure 1: A graphical illustration of GCN's computation model and dataflow graph in each forward layer. In (a), edges in red represent those along which information is being propagated; solid edges represent standard `Gather`/`Scatter` operations while dashed edges represent NN operations. (b) shows a mapping between the SAGA-NN programming model and the rule R1.

$ReLU$. H_L is the *activations matrix* of the L-th layer; $H_0 = X$ is the input feature matrix for all vertices.

Mapping R1 to the vertex-centric computation model is familiar to the systems community [55] — each forward layer has four components: `Gather` (GA), `ApplyVertex` (AV), `Scatter` (SC), and `ApplyEdge` (AE), as shown in Figure 1(a). One can think of layer L's activations matrix H_L as a group of *activations vectors*, each associated with a vertex (as analogous to *vertex value* in the graph system's terminology). The goal of each forward layer is to compute a new activations vector for each vertex based on the vertex's previous activations vector (which, initially, is its feature vector) and the information received from its in-neighbors. Different from traditional graph processing, the computation of the new activations matrix H_{L+1} is based on computationally intensive NN operations rather than a numerical function.

Figure 1(b) illustrates how these vertex-centric graph operations correspond to various components in R1. First, GA retrieves a vector from each in-edge of a vertex and aggregates these vectors into a new vector v. In essence, applying GA on all vertices can be implemented as a matrix multiplication $\hat{A}H_L$, where $\hat{A}$ is the normalized adjacency matrix and H_L is the input activations matrix. Second, $(\hat{A}H_L)$ is fed to AV, which performs neural network operations to obtain a new activations matrix H_{L+1}. For GCN, AV multiplies $(\hat{A}H_L)$ with a trainable weight matrix W_L and applies a non-linear activation function σ. Third, the output of AV goes to SC, which propagates the new activations vector of each vertex along all out-edges of the vertex. Finally, the new activations vector of each vertex goes into an edge-level NN architecture to compute an activations vector for each edge. For GCN, the edge-level NN is not needed, and hence, AE is an identity

function. We leave AE in the figure for generality as it is needed by other GNN models.

The output of AE is fed to GA in the next layer. Repeating this process k times (*i.e.*, k layers) allows the vertex to consider features of vertices k hops away. Other GNNs such as GGNNs and GATs have similar computation models, but each varies the method used for aggregation and the NN.

Backward Pass. A GNN's backward pass computes the gradients for all trainable weights in the vertex- and edge-level NN architectures (*i.e.*, AV and AE). The backward pass is performed following the chain rule of back propagation. For example, the following rule specifies how to compute the gradients in the first layer for a 2-layer GCN:

$$(R2) \quad \nabla_{W_0}\mathcal{L} = \left(\hat{A}X\right)^T \left[\sigma'\left(in_1\right) \odot \hat{A}^T(Z-Y)W_1^T\right]$$

Here Z is the output of the GCN, Y is the label matrix (*i.e.*, ground truth), X is the input features matrix, W_i is the weight matrix for layer i, and $in_1 = \hat{A}XW_0$. $\hat{A}^T$ and W_i^T are the transpose of $\hat{A}$ and W_i, respectively.

A training *epoch* consists of a forward and a backward pass, followed by *weights update*, which uses the gradients computed in the backward pass to update the trainable weights in the vertex- and edge-level NN architectures in a GNN. The training process runs epochs repeatedly until reaching acceptable accuracy.

3 Design Overview

This section provides an overview of the Dorylus architecture. The next three sections discuss technical details including how to split training into fine-grained tasks and connect them in a deep pipeline (§4), and how Dorylus bounds the degree of asynchrony (§5), manages and autotunes Lambdas (§6).

Figure 2 depicts Dorylus's architecture, which is comprised of three major components: EC2 graph servers, Lambda threads for tensor computation, and EC2 parameter servers. An input graph is first partitioned using an edge-cut algorithm [104] that takes care of load balancing across partitions. Each partition is hosted by a graph server (GS).

GSes communicate with each other to execute graph computations by sending/receiving data along cross-partition edges. GSes also communicate with Lambda threads to execute tensor computations. Graph computation is done in a conventional way, breaking a vertex program into vertex-parallel (*e.g.*, `Gather`) and edge-parallel stages (*e.g.*, `Scatter`).

Each vertex carries a vector of float values and each edge carries a value of a user-defined type specific to the model. For example, for a GCN, edges do not carry values and `ApplyEdge` is an identity function; for a GGNN, each edge has an integer-represented type, with different weights for different edge types. After partitioning, each GS hosts a graph partition where vertex data are represented as a two-dimension array and edge data are represented as a single array. Edges are stored in the *compressed sparse rows* (CSR) format; inverse edges are also maintained for the backpropagation.

Each GS maintains a *ghost buffer*, storing data that are scattered in from remote servers. Communication between GSes is needed only during `Scatter` in both (1) forward pass where activation values are propagated along cross-partition edges and (2) backward pass where gradients are propagated along the same edges in the reverse direction.

Tensor operations such as AV and AE, performed by Lambdas, interleave with graph operations. Once a graph operation finishes, it passes data to a Lambda thread, which employs a high-performance linear algebra kernel for tensor computation. Both the forward and backward passes use Lambdas, which communicate frequently with parameter servers (PS) — the forward-pass Lambdas retrieve weights from PSes to compute layer outputs, while the backward-pass Lambdas compute updated weights.

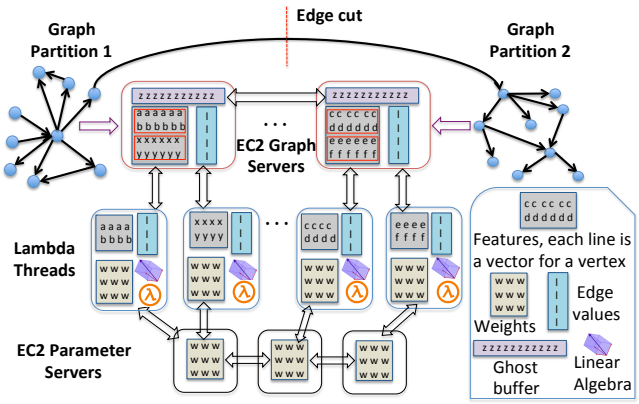


Figure 2: Dorylus's architecture.

4 Tasks and Pipelining

Fine-grained Tasks. As discussed in §1, the first challenge in using Lambdas for training is to decompose the process into a set of fine-grained tasks that can (1) overlap with each other and (2) be processed by Lambdas' weak compute. In Dorylus, task decomposition is done based on both *data type* and *computation type*. In general, computations that involve the adjacency matrix of the input graph (*i.e.*, any computation that multiplies any form of the adjacency matrix A with other matrices) are formulated as graph operations performed on GSes, while computations that involve only tensor data can benefit the most from massive parallelism and hence run in Lambdas. Next, we discuss specific tasks over each training epoch, which consists of a *forward pass* that computes the output using current weights, followed by a *backward pass* that uses a loss function to compute weight updates.

A forward pass can be naturally divided into four tasks, as shown in Figure 1(a). `Gather` (GA) and `Scatter` (SC) perform computation over the graph structure; they are thus graph-parallel tasks for execution on GSes. `ApplyVertex`

(AV) and `ApplyEdge` (AE) multiply matrices involving only features and weights and apply activation functions such as $ReLU$. Hence, they are executed by Lambdas.

For AV, Lambda threads retrieve vertex data (H_L in §2) from GSes and weight data (W_L) from PSes, compute their product, apply $ReLU$, and send the result back to GSes as the input for `Scatter`. When AV returns, SC sends data, along cross-partition edges, to the machines that host their destination vertices.

AE immediately follows SC. To execute AE on an edge, each Lambda thread retrieves (1) vertex data from the source and destination vertices of the edge (*i.e.*, activations vectors), and (2) edge data (such as edge weights) from GSes. It computes a per-edge update by performing model-specific tensor operations. These updates are streamed back to GSes and become the inputs of the next layer's GA task.

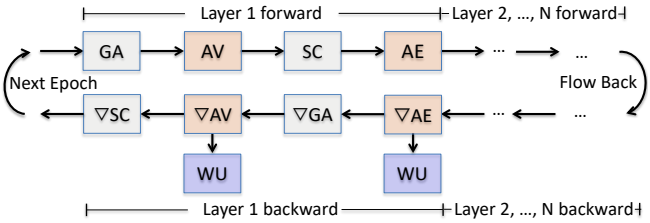


Figure 3: Dorylus's forward and backward dataflow with nine tasks: `Gather` (GA) and `Scatter` (SC) and their corresponding backward tasks ∇GA and ∇SC; `ApplyVertex` (AV), `ApplyEdge` (AE), and their backward tasks ∇AV and ∇AE; the weight update task `WeightUpdate` (WU).

A backward pass involves GSes, Lambdas, and PSes that *coordinate* to run a graph-augmented SGD algorithm, as specified by R2 in §2. For each task in the forward pass, there is a corresponding backward task that either propagates information in the *reverse direction of edges* on the graph or computes the gradients of its trainable weights with respect to a given loss function. Additionally, a backward pass includes `WeightUpdate` (WU), which aggregates the gradients across PSes. Figure 3 shows their dataflow. ∇GA and ∇SC are the same as GA and SC except that they propagate information in the reverse direction. ∇AE and ∇AV are the backward tasks for AE and AV, respectively. AE and AV apply weights to compute the output of the edge and vertex NN. Conversely, ∇AE and ∇AV compute weight updates for the NNs, which are the inputs to WU.

∇AE and ∇AV perform tensor-only computation and are executed by Lambdas. Similar to the forward pass, GA and SC in the backward pass are executed on GSes. WU performs weights updates and is conducted by PSes.

Pipelining. In the beginning, vertex and weight data take their initial values (*i.e.*, H_0 and W_0), which will change as the training progresses. GSes kick off training by running parallel graph tasks. To establish a full pipeline, Dorylus divides vertices in each partition into intervals (*i.e.*, minibatches). For each interval, the amount of tensor computation (done by a Lambda) depends on both the numbers of vertices (*i.e.*, AV) and edges (*i.e.*, AE) in the interval, while the amount of graph computation (on a GS) depends primarily on the number of edges (*i.e.*, GA, and SC). To balance work across intervals, our division uses a simple algorithm to ensure that different intervals have the same numbers of vertices and vertices in each interval have similar numbers of inter-interval edges. These edges incur cross-minibatch dependencies that our asynchronous pipeline needs to handle (see §5).

Each interval is processed by a task. When the pipeline is saturated, different tasks will be executed on distinct *intervals* of vertices. Each GS maintains a *task queue* and enqueues a task once it is ready to execute (*i.e.*, its input is available). To fully utilize CPU resources, the GS uses a thread pool where the number of threads equals the number of vCPUs. When the pool has an available thread, the thread retrieves a task from the task queue and executes it. The output of a GS task is fed to a Lambda for tensor computation.

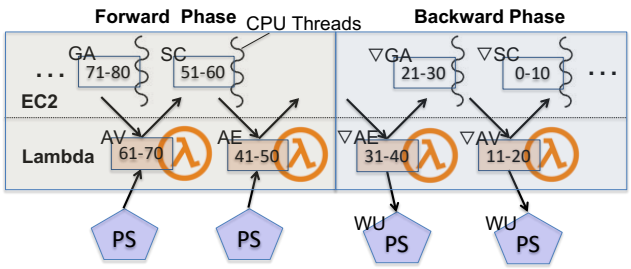


Figure 4: A Dorylus pipeline for an epoch: the number range (*e.g.*, 71-80) in each box represents a particular vertex interval (*i.e.*, minibatch); different intervals are at different locations of the pipeline and processed by different processing units: GS, Lambda, or PS.

Figure 4 shows a typical training pipeline under Dorylus. Initially, Dorylus enqueues a set of GA tasks, each processing a vertex interval. Since the number of threads on each GS is often much smaller than the number of tasks, some tasks finish earlier than others and their results are pushed immediately to Lambda threads for AV. Once they are done, their outputs are sent back to the GS for `Scatter`. During a backward phase, both ∇AE and ∇AV compute gradients and send them to PSes for weight updates.

Through effective pipelining, Dorylus overlaps the graph-parallel and tensor-parallel computations so as to hide Lambdas' communication latency. Note that although pipelining is not a new idea, enabling pipelining in GNN training requires fine-grained tasks and the insight of computation separation, which are our unique contributions.

5 Bounded Asynchrony

To unleash the full power of pipelining, Dorylus performs a unique form of *bounded asynchronous* training so that work-

ers do not need to wait for updates to proceed in most cases. This is paramount for the pipeline's performance especially because Lambdas run in an extremely dynamic environment and stragglers almost always exist. On the other hand, a great deal of evidence [13, 63, 99] shows that asynchrony slows down convergence — fast-progressing minibatches may use out-of-date weights, prolonging the training time.

Bounded staleness [15, 65] is an effective technique for mitigating the convergence problem by employing lightweight synchronization. However, Dorylus faces a unique challenge that does not exist in any existing system, that is, there are *two synchronization points* in a Dorylus pipeline: (1) weight synchronization at each WU task and (2) synchronization of (vertex) activations data from neighbors at each GA.

5.1 Bounded Asynchrony at Weight Updates

To bound the degree of asynchrony for weight updates, we use *weight stashing* proposed in PipeDream [63]. A major reason for slow convergence is that, under full asynchrony, different vertex intervals are at their own training pace; some intervals may use a particular version v_0 of weights during a forward pass to compute gradients while applying these gradients on another version v_1 of weights on their way back in the backward pass. In this case, the weights on which gradients are computed are not those on which they are applied, leading to *statistical inefficiency*. Weight stashing is a simple technique that allows any interval to use the latest version of weights available in a forward pass and stashes (*i.e.*, caches) this version for use during the corresponding backward pass.

Although weight stashing is not new, applying it in Dorylus poses unique challenges in the PS design. Weight stashing occurs at PSes, which host weight matrices and perform updates. To balance loads and bandwidth usage, Dorylus supports multiple PSes and always directs a Lambda to a PS that has the lightest load. Since Lambdas can be in different stages of an epoch (*e.g.*, the forward and backward passes, and different layers), Dorylus lets each PS host a replication of weight matrices of *all layers*, making load balancing much easier to do since any Lambda can use any PS in any stage. Note that this design is different from that of traditional PSes [49], each of which hosts parameters for a layer. Since a GNN often has very few layers, replicating all weights would not take much memory and is thus feasible to do at each PS. Clearly, this approach does *not* work for regular DNNs with many layers.

However, weight stashing significantly complicates this design. A vertex interval can be processed by different Lambdas when it flows to different tasks — *e.g.*, its AV and AE are executed by different Labmdas, which can retrieve weights from different PSes. Hence, if we allow any Lambda to use any PS, each PS has to maintain not only the latest weight matrices but also their stashed versions for *all* intervals in the graph; this is practically impossible due to its prohibitively high memory requirement.

To overcome this challenge, we do *not* replicate all weight stashes across PSes. Instead, each PS still contains a replication of all the latest weights but weight stashes only for a *subset* of vertex intervals. For each interval in a given epoch, the interval's weight stashes are only maintained on the first PS it interacts with in the epoch. In particular, once a Lambda is launched for an AV task, which is the first task that uses weights in the pipeline, its launching GS picks the PS with the lightest load and notifies the Lambda of its address. Furthermore, the GS remembers this choice for the interval — when this interval flows to subsequent tensor tasks (*i.e.*, AE, ∇AV, ∇AE, and WU), the GS assigns the same PS to their executing Lambdas because the stashed version for this interval only exists on that particular PS in this epoch.

PSes periodically broadcast their latest weight matrices.

5.2 Bounded Asynchrony at `Gather`

Asynchronous `Gather` allows vertex intervals to progress independently using stale vertex values (*i.e.*, activations vectors) from their neighbors without waiting for their updates. Although asynchrony has been used in a number of graph systems [82, 15], these systems perform iterative processing with the assumption that with more iterations they will eventually reach convergence. Different from these systems, the number of layers in a GNN is determined statically and an n-layer GNN aims to propagate the impact of a vertex's n-hop neighborhood to the vertex. Since the number of layers cannot change during training, an important question that needs be answered is: can asynchrony change the semantics of the GNN? This boils down to two sub-questions: (1) Can vertices *eventually* receive the effect of their n-hop neighborhood? (2) Is it possible for any vertex to receive the effect of its m-hop neighbor where $m > n$ after many epochs? We provide informal correctness/convergence arguments in this subsection and turn to a formal approach in §5.3.

The answer to the first question is *yes*. This is because the GNN computation is driven by the accuracy of the computed weights, which is, in turn, based on the effects of n-hop neighborhoods. To illustrate, consider a simple 2-layer GNN and a vertex v that moves faster in the pipeline than all its neighbors. Assume that by the time v enters the GA of the second layer, none of its neighbors have finished their first-layer SC yet. In this case, the GA task of v uses stale values from its neighbors (*i.e.*, the same as what were used in the previous epoch). This would clearly generate large errors at the end of the epoch. However, in subsequent epochs, the slow-moving neighbors update their values, which are gradually propagated to v. Hence, the vertex eventually receives the effects of its n-hop neighborhood (collectively) across different epochs depending on its neighbors' progress. After each vertex observes the required values from the n-hop neighborhood, the target accuracy is reached.

The answer to the second question is *no*. This is because the number of layers determines the *farthest distance* the impact of a vertex can travel despite that training may execute

many epochs. When a vertex interval finishes an epoch, it comes back to the initial state where their values are *reset* to their initial feature vectors (*i.e.*, the accumulative effect is cleared). Hence, even though a vertex v progresses asynchronously relative to its neighbors, the neighbors' activation vectors are scattered out in the previous SC and carry the effects of their at most $\{n-1\}$-hop neighbors (after which the next GA will cycle back to effects of 1-hop neighbors), which, for vertex v, belong strictly in its n-hop neighborhood. This means, it is impossible for any vertex to receive the impact of any other vertex that is more than n-hops away.

We use *bounded staleness* at `Gather` — a fast-moving vertex interval is allowed to be at most S epochs away from the slowest-moving interval. This means vertices in a given epoch are allowed to use stale vectors from their neighbors only if these vectors are within S epochs away from the current epoch. Bounded staleness allows fast-moving intervals to make quick progress when recent updates are available (for efficiency), but makes them wait when updates are too stale (to avoid launching Lambdas for useless computation).

5.3 Convergence Guarantee

Asynchronous weight updates with bounded staleness has been well studied, and its convergence has been proved by [30]. The convergence of asynchronous `Gather` with bounded staleness S is guaranteed by the following theorem:

Theorem 1. *Suppose that (1) the activation $\sigma(\cdot)$ is ρ-Lipschitz, (2) the gradient of the cost function $\nabla_z f(y,z)$ is ρ-Lipschitz and bounded, (3) the gradients for weight updates $\|g_{AS}(W)\|_\infty$, $\|g(W)\|_\infty$, and $\|\nabla\mathcal{L}(W)\|_\infty$ are all bounded by some constant $G > 0$ for all $\hat{A}$, X, and W, (4) the loss $\mathcal{L}(W)$ is ρ-smooth*[3]*. Then given the local minimizer W^*, there exists a constant $K > 0$, s.t., $\forall N > L \times S$ where L is the number of layers of the GNN model and S is the staleness bound; if we train a GNN with asynchronous `Gather` under a bounded staleness for $R \leq N$ iterations where R is chosen uniformly from $[1, N]$, we will have*

$$\mathbb{E}_R \left\|\nabla\mathcal{L}\left(W_R\right)\right\|_F^2 \leq 2\frac{\mathcal{L}\left(W_1\right) - \mathcal{L}\left(W^*\right) + K + \rho K}{\sqrt{N}},$$

for the updates $W_{i+1} = W_i - \gamma g_{AS}(W_i)$ and the step size $\gamma = min\left\{\frac{1}{\rho}, \frac{1}{\sqrt{N}}\right\}$.

In particular, we have $\lim_{N\to\infty} \mathbb{E}_R \left\|\nabla\mathcal{L}\left(W_R\right)\right\|^2 = 0$, indicating that asynchronous GNN training will eventually converge to a local minimum. The full-blown proof can be found in Dorylus' arXiv version [77]. It mostly follows the convergence proof of the *variance reduction (VR)* algorithm in [9]. However, our proof differs from [9] in two major aspects: (1) Dorylus performs whole-graph training and updates weights only once per layer per epoch, while VR samples the graph and trains on mini-batches and thus it updates weights multiple times per layer per epoch; (2) Dorylus's asynchronous GNN training can use neighbor activations that are up to S-epoch stale, while VR can take only 1-epoch-stale neighbor activations. Since S is always *bounded* in Dorylus, the convergence is guaranteed regardless of the value of S.

Note that compared to a sampling-based approach, our asynchronous computation is guaranteed to converge. On the contrary, although sampling-based training converges often in practice, there is no guarantee for trivial sampling methods [9], not to mention that sampling incurs a per-epoch overhead and reduces accuracy.

6 Lambda Management

Each GS runs a Lambda controller, which launches Lambdas, batches data to be sent to each Lambda, monitors each Lambda's health, and routes its result back to the GS.

Lambda threads are launched by the controller for a task t at the time t's previous task starts executing. For example, Dorylus launches n Lambda threads, preparing them for AV when n GA tasks start to run. Each Lambda runs with OpenBLAS library [93] that is optimized to use AVX instructions for efficient linear algebra operations. Lambdas communicate with GSes and PSes using ZeroMQ [97].

All of our Lambdas are deployed inside the virtual private cloud (VPC) rather than public networks to maximize Lambdas' bandwidth when communicating with EC2 instances. When a Lambda is launched, it is given the addresses of its launching GS and a PS. Once initialized, the Lambda initiates communication with the GS and the PS, pulling vertex, edge and weight data from these servers. Since Lambda threads are used throughout the training process, these Lambdas quickly become "warm" (*i.e.*, the AWS reuses a container that already has our code deployed instead of cold-starting a new container) and efficient. Our controller also times each Lambda execution and relaunches it after timeout.

Lambda Optimizations. One significant challenge to overcome is Lambdas' limited network bandwidth [29, 42]. Although AWS has considerably improved Lambdas' network performance [4], the per-Lambda bandwidth goes down as the number of Lambdas increases. For example, for each GS, when the number of Lambdas it launches reaches 100, the per-Lambda bandwidth drops to ∼200Mbps, which is more than 3× lower than the peak bandwidth we have observed (∼800Mbps). We suspect that this is because many Lambdas created by the same user get scheduled on the same machine and share a network link.

Dorylus provides three optimizations for Lambdas:

The first optimization is task fusion. Since AV of the last layer in a forward pass is connected directly to ∇AV of the last layer in the next backward pass (see Figure 4), we merge them into a single Lambda-based task, reducing invocations

[3] $\mathcal{L}$ is *ρ-Lipschitz smooth* if $\forall W_1, W_2, |\mathcal{L}(W_2) - \mathcal{L}(W_1) - \langle\nabla\mathcal{L}(W_1), W_2 - W_1\rangle| \leq \frac{\rho}{2}\|W_2 - W_1\|_F^2$, where $\langle A, B\rangle = tr(A^T B)$ is the inner product of matrix A and matrix B, and $\|A\|_F$ is the Frobenius norm of matrix A.

of thousands of Lambdas for each epoch and saving a round-trip communication between Lambdas and GSes.

The second optimization is tensor rematerialization [33, 41]. Existing frameworks cache intermediate results during the forward pass as these results can be reused in the backward pass. For GNN training, for instance, $\hat{A}HW$ is such a computation whose result needs to be cached. Here tensor computation is performed by Lambdas while caching has to be done on GSes. Since a Lambda's bandwidth is limited and network communication is a bottleneck, it is more profitable to rematerialize these intermediate tensors by launching more Lambdas rather than retrieving them from GSes.

The third optimization is Lambda-internal streaming. In particular, if a Lambda is created to process a data chunk, we let the Lambda retrieve the first half of the data, with which it proceeds to computation while simultaneously retrieving the second half. This optimization overlaps computation with communication from within each Lambda, leading to reduced Lambda response time.

Autotuning Numbers of Lambdas. Due to inherent dynamism in Lambda executions, it is not feasible to statically determine the number of Lambdas to be used. On the performance side, the effectiveness of Lambdas depends on whether the pipeline can be saturated. In particular, since certain graph tasks (such as SC) rely on results from tensor tasks (such as AV), too few Lambdas would not generate enough task instances for the graph computation G to saturate CPU cores. On the cost side, too many Lambdas *overstaturate* the pipeline — they can generate too many CPU tasks for the GS to handle. The optimal number of Lambdas is also related to the pace of the graph computation, which, in turn, depends on the graph structure (*e.g.*, density) and partitioning that are hard to predict before execution.

To solve the problem, we develop an autotuner that starts the pipeline by using `min`(#`intervals`, 100) as the number of Lambdas where `intervals` represents the number of vertex intervals on each GS. Our autotuner auto-adjusts this number by periodically checking the size of the CPU's task queue — if the size of the queue constantly grows, this indicates that CPU cores have too many tasks to process, and hence we scale down the number of Lambdas; if the queue quickly shrinks, we scale up the number of Lambdas. The goal here is to stabilize the size of the queue so that the number of Lambdas matches the pace of graph tasks.

7 Evaluation

We wrote a total of 11629 SLOC in C++ and CUDA. There are 10877 of the lines of C++ code: 5393 for graph servers, 2840 for Lambda management (and communication), 1353 for parameter servers, and 1291 for common libraries and utilities. There are 752 lines of CUDA code for GPU kernels including common graph operations like GCN and mean-aggregators with cuSPARSE [66]. Our CUDA code includes deep learning operations such as dense layer and activation layer with cuDNN [12]. Dorylus supports common stochastic optimizations including *Xavier initialization* [22], *He initialization* [27], a *vanilla SGD optimizer* [38], and an *Adam optimizer* [39], which help training converge smoothly.

Graph	Size ($\|V\|$, $\|E\|$)	# features	# labels	Avg. degree
Reddit-small [25]	(232.9K, 114.8M)	602	41	492.9
Reddit-large [25]	(1.1M, 1.3B)	301	50	645.4
Amazon [60, 28]	(9.2M, 313.9M)	300	25	35.1
Friendster [48]	(65.6M, 3.6B)	32	50	27.5

Table 1: We use 4 graphs, 2 with billions of edges.

7.1 Experiment Setup

We experimented with four graphs, as shown in Table 1. `Reddit-small` and `Reddit-large` are both generated from the Reddit dataset [68]. `Amazon` is the *largest graph* in RoC's [34] evaluation. We added a larger 1.8 billion (undirected) edge `Friendster` social network graph to our experiments. For GNN training, we turned undirected edges into two directed edges, effectively doubling the number of edges (which is consistent with how edge numbers are reported in prior GNN work [34, 55]). The first three graphs come with features and labels while `Friendster` does not. For scalability evaluation we generated random features and labels for `Friendster`.

We implemented two GNN models on top of Dorylus: graph convolutional network (GCN) [40] and graph attention network (GAT) [96] with 279 and 324 lines of code. GCN is a popular network that has AV but not AE, while GAT is a recently-developed recurrent network with both AV and AE. Their development is straightforward and other GNN models can be easily implemented on Dorylus as well. Each model has 2 layers, consistent with those used in prior work [34, 55].

Value is the major benefit Dorylus brings to training GNNs. We define value as a system's *performance per dollar*, computed as $V = 1/(T \times C)$ where T is the training time and C is the monetary cost. For example: if system A trains a network twice as fast as system B, and yet costs the same to train, we say A has twice the value of B. If one has a time constraint, the most inexpensive option to train a GNN is to pick the system/configuration that meets the time requirement with the best value. In particular, value is important for training since users cannot take the cheapest option if it takes too long to train; neither can they take the fastest option if it is extremely expensive in practice. Throughout the evaluation, we use both value and performance (runtime) as our metrics.

We evaluated several aspects of Dorylus. First, we compared several different instance types to determine the configurations that give us the optimal value for each backend. Second, we compared several synchronous and asynchronous variants of Dorylus. In later subsections, we use our best variant (which is asynchronous with a staleness value of 0) in comparisons with other existing systems. Third, we compared the effects of Lambdas using Dorylus against more traditional CPU- and GPU-only implementations in terms of value, per-

formance, and scalability. Next, we evaluate Dorylus against existing systems. Finally, we break down our performance and costs to illustrate our system's benefits.

Backend	Graph	Instance Type	Relative Value
CPU	Reddit-large	r5.2xlarge (4)	1
		c5n.2xlarge (12)	4.46
	Amazon	r5.xlarge (4)	1
		c5n.2xlarge (8)	2.72
GPU	Amazon	p2.xlarge (8)	1
		p3.2xlarge (8)	4.93

Table 2: Comparison of the values provided by different instance types. r5 and p2 instances provided significantly lower values than the (c5 and p3) instances we chose.

7.2 Instance Selection

To choose the instance types for our evaluation, we ran a set of experiments to determine the types that gave us the best value for each backend. We compared across memory optimized (r5) and compute optimized (c5) instances, as well as the p2 and p3 GPU instances, which have K80 and V100 GPUs, respectively. As r5 offers high memory, we were able to fit the graph in a smaller number of instances, lowering costs in some cases. However, due to the smaller amount of computational resources available, training on the r5 instances typically took nearly $3\times$ as long as computation on c5. Therefore, as shown in Table 2 the average increases in value c5 instances provided relative to r5 instances are 4.46 and 2.72, respectively, for `Reddit-large` and `Amazon`. We therefore selected c5 as our choice for any CPU based computation.

Similarly, for GPU instances, training on `Amazon` with 8 K80s took 1578 seconds and had a total cost of $3.16. Using 8 V100s took 385 seconds and cost $2.62—it improves both costs and performance, resulting in a value increase of $4.93\times$ compared to training on K80 GPUs. As value is the main metric which we use to evaluate our system, we choose the instance type which gives the best value to each different backend to ensure a fair comparison.

Given these results, we selected the following instances to run our evaluation: (1) c5, compute-optimized instances, and (2) c5n, compute and network optimized instances. c5n instances have more memory and faster networking, but their CPUs have slightly lower frequency than those in c5. The base c5 instance has 2 vCPU, 4 GB RAM, and 10 Gbps per-instance network bandwidth costing $0.085/h[4]. The base c5n instance has 2 vCPU, 5.25 GB RAM (33% more), and 25 Gbps per-instance network bandwidth, costing $0.108/h. We used the base p3 instance, p3.2xlarge, with Telsa V100 GPUs. Each p3 base instance has 1 GPU (with 16 GB memory), 8 vCPUs, and 61 GB memory, costing $3.06/h.

Each Lambda is a container with 0.11 vCPUs and 192 MB memory. Lambdas have a static cost of $0.20 per 1 M requests, and a compute cost of $0.01125/h (billed per 100 ms). This billing granularity enables serverless threads to handle short bursts of massive parallelism much better than CPU instances.

[4]These prices are from the Northern Virginia region.

Model	Graph	CPU cluster	GPU cluster
GCN	Reddit-small	c5.2xlarge (2)	p3.2xlarge (2)
	Reddit-large	c5n.2xlarge (12)	p3.2xlarge (12)
	Amazon	c5n.2xlarge (8)	p3.2xlarge (8)
	Friendster	c5n.4xlarge (32)	p3.2xlarge (32)
GAT	Reddit-small	c5.2xlarge (10)	p3.2xlarge (10)
	Amazon	c5n.2xlarge (12)	p3.2xlarge (12)

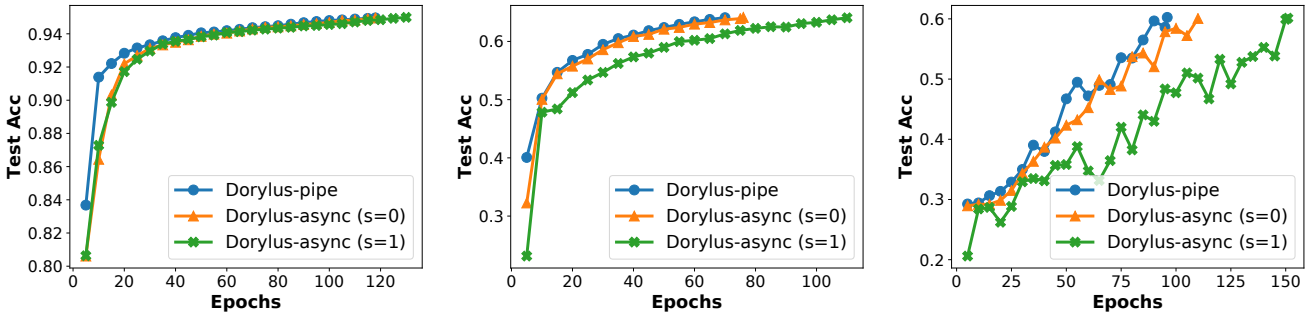
Table 3: We used (mostly) c5n instances for CPU clusters, and equivalent numbers of p3 instances for GPU clusters.

Table 3 shows our CPU and GPU clusters for each pair of model and graph we evaluated. For each graph, we picked the number of servers such that they have just enough memory to hold the graph data and their tensors. For example, `Amazon` needs 8 c5n.2xlarge servers (with 16 GB memory) provide enough memory. For `Friendster` we need 32 c5n.4xlarge instances (with a total of 1344 GB memory). Our goal is to train a model with the minimum amount of resources. Of course, using more servers will lead to better performance and higher costs (discussed in §7.4). For all experiments (except `Reddit-small`), c5n instances offered the best value.

TPU has become an important type of computation accelerator for machine learning. This paper focuses on AWS and its serverless platform, and hence we did not implement Dorylus on TPUs. Although we did not compare directly with TPUs, we note several important features of GNNs that make the limitations of TPUs comparable to GPUs. First, GNNs are unlike conventional DNNs in that they require large amounts of data movement for neighborhood aggregation. As a result, GNN performance is mainly bottlenecked by memory constraints and the resulting communication overheads (*e.g.*, between GPUs or TPUs), *not* computation efficiency [34]. Second, GNN training involves computation on large sparse tensors that incur irregular data accesses, resulting in sub-optimal performance on TPUs which are optimized for dense matrix operations over regularly structured data.

7.3 Asynchrony

We compare three versions of Dorylus: a synchronous version with full intra-layer pipelining (pipe), and two asynchronous versions using $s = 0$ and $s = 1$ as the staleness values over all four graphs. Pipe synchronizes at each `Gather` — a vertex cannot go into the next layer until all its neighbors have their latest values scattered. As a result, all vertex intervals have to be in the same layer in the same epoch. However, inside each layer, pipelining is enabled, and hence different tasks are fully overlapped. Async enables both pipelining and asynchrony (*i.e.*, stashing weights and using stale values at GA). When the staleness value is s = 0, Dorylus allows a vertex to use a stale value from a neighbor as long as the neighbor is in the same epoch (*e.g.*, can be in a previous layer). In other words, Async (s=0) enables fully pipelining across different layers



(a) Reddit-small
R[s=0]:1.00, R[s=1]:1.07, Accuracy:94.96%

(b) Amazon
R[s=0]:1.09, R[s=1]:1.57, Accuracy:64.08%

(c) Reddit-large
R[s=0]:1.14, R[s=1]:1.58, Accuracy:60.07%

Figure 5: Asynchronous progress for GCN: All three versions of Dorylus achieve the final accuracy *i.e.*, **94.96%**, **64.08%**, **60.07%** for the three graphs). Friendster is not included because it does not come with meaningful features and labels.

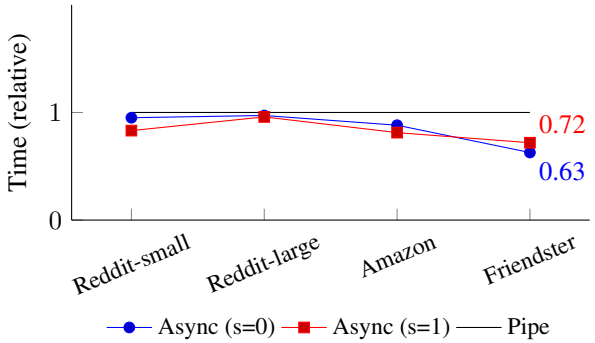


Figure 6: Per-epoch GCN time for async (s=0) and async (s=1) normalized to that of pipe.

in the same epoch, but pipelining tasks in different epochs are not allowed and synchronization is needed every epoch. Similarly, async (s=1) enables a deeper pipeline across two consecutive epochs.

Training Progress. Due to the use of asynchrony, it may take the asynchronous version of Dorylus more epochs to reach the same accuracy as pipe. To enable a fair comparison, we first ran Dorylus-pipe until convergence (*i.e.*, the difference of the model accuracy between consecutive epochs is within 0.001, unless otherwise stated) and then used this accuracy as the target accuracy to run async when collecting training time. However, this approach does not work for Friendster, because it uses randomly generated features/labels and hence accuracy is not a meaningful target. To solve this problem, we computed an average ratio, across the other three graphs, between the numbers of epochs needed for async and pipe, and used this ratio to estimate the training time for Friendster. For example, this ratio is 1.08 for s=0 and 1.41 for s=1. As such, we let async (s=0) run N$\times$1.08 epochs and async (s=1) run N$\times$1.41 epochs when measuring performance for Friendster where N is the number of epochs pipe runs.

Figure 5 reports the GCN training progress for each variant, that is, how many epochs it took for a version to reach the target accuracy. Annotated with each figure are two ratios: R[s=0] and R[s=1], representing the ratio between the number of epochs needed by async (s=0/1) and that by Dorylus-pipe to reach the same target accuracy. On average, async (s=0/1) increases the number of epochs by 8%/41%.

Figure 6 compares the per-epoch running time for each version of Dorylus, normalized to that of pipe. As expected, async has lower per-epoch time; in fact, async (s=0) achieves almost the same reduction ($\sim$15%) in per-epoch time as s=1. This indicates that choosing a large staleness value has little usefulness — it cannot further reduce per-epoch time and yet the number of epochs grows significantly.

To conclude, asynchrony can provide overall performance benefits in general but too large a staleness value leads to slow convergence and poor performance, although the per-epoch time reduces. This explains why async (s=0) outperforms async (s=1) by a large margin. Overall, async (s=0) is **1.234**$\times$ faster than pipe and **1.233**$\times$ than async (s=1). It also provides **1.288**$\times$ and **1.494**$\times$ higher value than pipe and async (s=1) respectively. Thus we choose it as the default Lambda variant in our following experiments unless otherwise specified. From this point on, Dorylus refers to this particular version.

7.4 Effects of Lambdas

We developed two traditional variants of Dorylus to isolate the effects of serverless computing using Lambdas, one using CPU-only servers for computations, and the other using GPU-only servers (both without Lambdas). These variants perform all tensor and graph computations directly on the graph server. They both use Dorylus' (tensor and graph) computation separation for scalability. Note that without computation separation, no existing GPU-based training system has been shown to scale to a billion-edge graph.

Since Lambdas have weak compute that we cannot find in regular EC2 instances, it is not possible for us to translate Lambda resources directly into equivalent EC2 resources, keeping the total amount of compute constant when selecting the number of servers for each variant. To address this con-

cern, we compared the value of different systems in addition to their absolute times and costs.

Model	Graph	Mode	Time (s)	Cost ($)
GCN	Reddit-small	Dorylus	860.6	0.20
		CPU only	1005.4	0.19
		GPU only	162.9	0.28
	Reddit-large	Dorylus (pipe)	1020.1	1.69
		CPU only	1290.5	1.85
		GPU only	324.9	3.31
	Amazon	Dorylus	512.7	0.79
		CPU only	710.2	0.68
		GPU only	385.3	2.62
	Friendster	Dorylus	1133.3	13.8
		CPU only	1990.8	15.3
		GPU only	1490.4	40.5
GAT	Reddit-small	Dorylus	496.3	1.15
		CPU only	1270.4	1.20
		GPU only	130.9	1.11
	Amazon	Dorylus	853.4	2.67
		CPU only	2092.7	3.01
		GPU only	1039.2	10.60

Table 4: We ran Dorylus in 3 different modes: "Dorylus", our best Lambda variant using async(s=0) (except in one case), the "CPU only" variant, and the "GPU only" variant. For each mode we used multiple combinations of models and graphs. For each run we report the total end-to-end running time and the total cost.

We ran GCN and GAT on our graphs (Table 4). We only ran the GAT model on one small and large graph because it was simply too monetarily expensive (even for our system!). GAT has an intensive AE computation, which adds cost. Note that this is *not* a limitation of our system—our system can scale GAT to graphs larger than `Amazon` if cost is not a concern.

Performance and cost by themselves do not properly illustrate the value of Dorylus. For example, training GAT on `Amazon` with Dorylus is both more efficient and cheaper than the CPU- and GPU-only variants. Hence, we report the value results as well. Recall that to compute the value, we take the reciprocal of the total runtime (*i.e.*, the performance or rate of completion) and divide it by the cost. In this case Dorylus with Lambdas provides a 2.75× higher value than CPU-only (*i.e.*, $1/(853.4 \times 2.67)$ compared to $1/(2092.7 \times 3.01)$). Figure 7 shows the value results for all our runs, *normalized to GPU-only servers*.

Dorylus adds value for large, sparse graphs (*i.e.*, `Amazon` and `Friendster`) for both GCN and GAT, compared to CPU- and GPU-only variants. Sparsity of each graph can be seen from the average vertex degree reported in Table 1. As shown, `Amazon` and `Friendster` are much more sparse than `Reddit-small` and `Reddit-large`. For these graphs, the GPU-only variant has the lowest value, even compared to the CPU-only variant. In most cases, the CPU-only variant provides twice as much value (*i.e.*, performance per dollar)

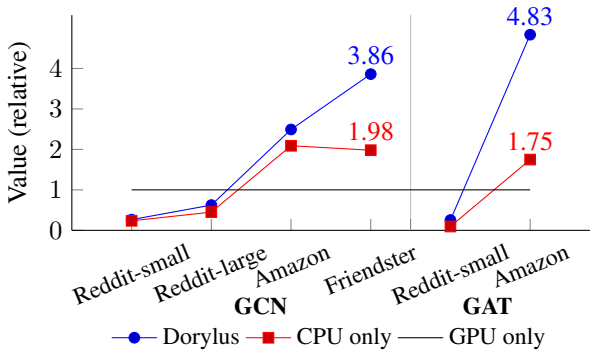


Figure 7: Dorylus, with Lambdas, provides up to 2.75× performance-per-dollar than using the CPU-only variant.

than the GPU-only variant. Dorylus adds another leap in value over the CPU-only variant.

However, for small dense graphs (*i.e.*, `Reddit-small` and `Reddit-large`), both Dorylus and the CPU-only variant have a value lower that that of the GPU-only variant (*i.e.*, below 1 in Figure 7). Dorylus always provides more value than the CPU-only variant. These results suggest that GPUs may be better suited to process small, dense graphs rather than large, sparse graphs.

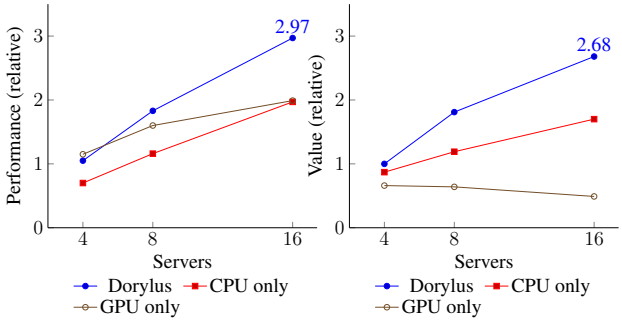


Figure 8: Normalized GCN training performance and value over `Amazon` with varying numbers of graph servers.

Scaling Out. Dorylus can gain even more value by scaling out to more servers, due to the burst parallelism provided by Lambdas and deep pipelining. To understand the impact of the number of servers on performance/costs, we varied the number of GSes when training a GCN over `Amazon`. In particular, we ran Dorylus and the CPU-only variant with 4, 8, and 16 c5n.4xlarge servers, and the GPU-only variant with the same numbers of p3.xlarge servers. Figure 8 reports their performance and values, normalized to those of Dorylus under 4 servers.

In general, Dorylus scales well in terms of both performance and value. Dorylus gains a 2.82× speedup with only 5% more cost when the number of servers increases from 4 to 16, leading to a 2.68× gain in its value. As shown in Figure 8(b), Dorylus's value curve is always above that of the

CPU-only variant. Furthermore, *Dorylus can roughly provide the same value as the CPU-only variant with only half of the number of servers*. For example, Dorylus with 4 servers provides a comparable value to the CPU-only variant with 8 servers; Dorylus with 8 servers provides more value to the CPU-only variant with 16 servers. These results suggest that as more servers are added, the value provided by Dorylus increases, at a rate much higher than the value increase of the CPU-only variant. As such, Dorylus is always a better choice than the CPU-only variant under the same monetary budget.

Other Observations. In addition to the results discussed above, we make three other observations on performance.

Our first observation is that the more sparse the graph, the more useful Dorylus is. For `Amazon` and `Friendster`, Dorylus even outperforms the GPU-only version for two reasons:

First, for all the three variants, the fraction of time on `Scatter` is significantly larger when training over `Friendster` and `Amazon` than `Reddit-small` and `Reddit-large`. This is, at first sight, counter-intuitive because one would naturally expect less efforts on inter-partition communications for sparse graphs than dense graphs. A thorough inspection discovered that the `Scatter` time actually depends on a *combination* of the number of ghost vertices and inter-partition edges. For the two Reddit graphs, they have many inter-partition edges, but very few ghost vertices, because (1) their $|V|$ is small and (2) many inter-partition edges come from/go to the same ghost vertices due to their high vertex degrees.

Second, `Scatter` takes much longer time in GPU clusters. Moving ghost data between GPU memories on different nodes is much slower than data transferring between CPU memories. As a result, the poor performance of the GPU-only variant is due to a combinatorial effect of these two factors: Dorylus scatters significantly more data for `Friendster` and `Amazon`, which amplifies the negative impact of poor scatter performance in a GPU cluster. Note that p3 also offers multi-GPU servers, which may potentially reduce scatter time. We have also experimented with these servers, but we still observed long scatter time due to extensive communication between between servers and GPUs. Reducing such communication costs requires fundamentally different techniques such as those proposed by NeuGraph [55]. We leave the incorporation of such techniques to future work.

Our second observation is that Lambda threads are more effective in boosting performance for GAT than GCN. This is because GAT includes an additional AE task, which performs intensive per-edge tensor computation and thus benefits significantly from a high degree of parallelism.

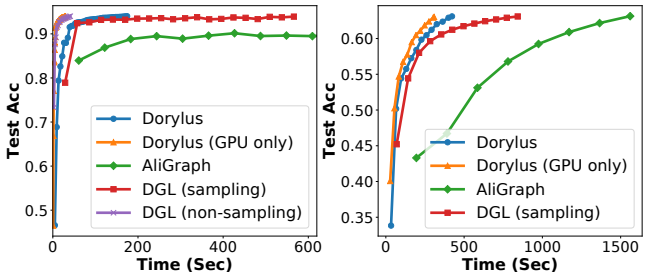
Our third observation is that Dorylus achieves comparable performance with the CPU-only variant that uses twice as many servers. For example, the training time of Dorylus under 4 servers is only 1.1× longer than that of the CPU only variant with 8 servers. Similarly, Dorylus under 8 servers is only 1.05× slower than the CPU only variant with 16 servers. These results demonstrate our efficient use of Lambdas.

7.5 Comparisons with Existing Systems

Our goal was to compare Dorylus with all existing GNN tools. However, NeuGraph [55] and AGL [98] are not publicly available; neither did their authors respond to our requests. Roc [34] is available but we could not run it in our environment due to various CUDA errors; we were not able to resolve these errors after multiple email exchanges with the authors. Roc was not built for scalability because each server needs to load the entire graph into its memory during processing. This is not possible when processing billion-edge graphs. This subsection focuses on the comparison of Dorylus, DGL [17], which is a popular GNN library with support for sampling, as well as AliGraph [94], which is also a sampling-based system that trains GNNs only with CPU servers. All experiments use the cluster configuration specified above for each graph unless otherwise stated.

DGL represents an input graph as a (sparse) matrix; both graph and tensor computations are executed by PyTorch or MXNet as matrix multiplications. We experimented with two versions of DGL, one with sampling and one without. DGL-non-sampling does full-graph training on a single machine. DGL-sampling partitions the graph and distributes partitions to different machines. Each machine performs sampling on its partition and trains a GNN on sampled subgraphs.

AliGraph runs in a distributed setting with a server that stores the graph information. A set of clients query the server to obtain graph samples and use them as minibatches for training. Similar to DGL, AliGraph uses a traditional ML framework as a backend and performs all of its computation as tensor operations.



(a)`Reddit-small` (b) `Amazon`

Figure 9: Accuracy comparisons between Dorylus, Dorylus (GPU only), AliGraph, DGL (sampling), and DGL (non-sampling). DGL (non-sampling) uses a single V100 GPU and could not scale to `Amazon`. Each dot indicates five epochs for Dorylus and DGL (non-sampling), and one epoch for DGL (sampling) and AliGraph.

Accuracy Comparison with Sampling. Figure 9 reports the accuracy-time curve for five configurations: Dorylus, Dorylus (GPU-only), DGL (sampling), DGL (non-sampling),

Graph	System	Time (s)	Cost ($)
Reddit-small	Dorylus	165.77	0.045
	Dorylus (GPU only)	28.06	0.052
	DGL (sampling)	566.33	0.480
	DGL (non-sampling)	33.64	0.028
	AliGraph	–	–
Amazon	Dorylus	415.23	0.654
	Dorylus (GPU only)	308.27	2.096
	DGL (sampling)	842.49	5.728
	DGL (non-sampling)	–	–
	AliGraph	1560.66	1.498

Table 5: Evaluation of end-to-end performance and costs of Dorylus and other GNN training systems. Each time reported is the time to reach the target accuracy.

and AliGraph, over `Reddit-small` and `Amazon`. When run enough epochs to fully converge, Dorylus can reach an accuracy of **95.44%** and **67.01%**, respectively, for the two graphs. DGL (non-sampling) can run only on the `Reddit-small` graph, reaching 94.01% as the highest accuracy. DGL (sampling) is able to scale to both graphs, and its accuracy reaches 93.90% and 65.78%, respectively, for `Reddit-small` and `Amazon`. AliGraph is able to scale to both `Reddit-small` and `Amazon`. On `Reddit-small` it reaches a maximum accuracy of 91.12% and 65.23% on `Amazon`.

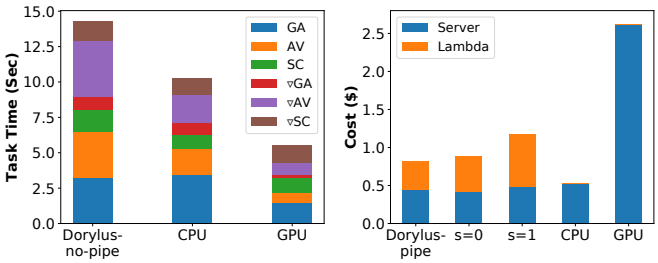
Performance. To enable meaningful performance comparisons and make training finish in a reasonable amount of time, we set 93.90% and 63.00% as our target accuracy for the two graphs. As shown in Figure 9(a), Dorylus (GPU only) has the best performance, followed by DGL (non-sampling). Since `Reddit-small` is a small graph that fits into the memory of a single (V100) GPU, DGL (non-sampling) performs much better than DGL (sampling), which incurs *per-epoch* sampling overheads. To reach the same accuracy (93.90%), Dorylus is 3.25× faster than DGL (sampling), but 5.9× slower than Dorylus (GPU only). AliGraph is unable to reach our target accuracy after many epochs.

For the `Amazon` graph, DGL cannot scale without sampling. As shown in Figure 9(b), to reach the same target accuracy, Dorylus is 1.99× faster than DGL (sampling), and 1.37× slower than Dorylus (GPU only). AliGraph is able to reach the target accuracy for `Amazon`. However, Dorylus is significantly faster. As these results show, graph sampling improves scalability at the cost of increased overheads and reduced accuracy.

The times reported for Dorylus and its GPU-only variant in Table 5 are smaller than those reported in Table 4. This is due to the lower target accuracy we set for these experiments.

Value Comparison. To demonstrate the promise of Dorylus, we compared these systems using the value metric. As expected, given the small size of the `Reddit-small` graph, the GPU-based systems perform quite well. In fact, in this case the normalized value of DGL (non-sampling) is 1.48, providing a higher value than Dorylus (GPU only). However, as mentioned earlier, DGL cannot scale without sampling; hence, this benefit is limited only to small graphs. As we process `Amazon`, the value of Dorylus quickly improves as is consistent with our findings earlier (on large, sparse graphs). With this dataset, Dorylus provides a higher performance-per-dollar rate than *all* the other systems—17.7× the value of DGL (sampling) and 8.6× the value of AliGraph.

7.6 Breakdown of Performance and Costs



(a) Task time breakdown (b) Cost breakdown

Figure 10: Time and cost breakdown for the `Amazon` graph.

Figure 10 shows a breakdown in task time (a) and costs (b) for training a GCN over the `Amazon` graph. In Figure 10(a), to understand the time each task spends, we disabled pipelining and asynchrony in Dorylus, producing a version referred to as no-pipe, in which different tasks never overlap. This makes it possible for us to collect each task's running time. Note that no-pipe represents a version that uses Lambdas naïvely to train a DNN. Without pipelining and overlapping Lambdas with CPU-based tasks, we saw a **1.9**× degradation, making no-pipe lose to both CPU and GPU in training time.

As shown, the tasks GA, AV, and ∇AV take the majority of the time. Another observation is that to execute the tensor computation AV, GPU is the most efficient backend and Lambda is the least efficient one. This is expected — Lambdas have less powerful compute (much less than CPUs in the c5 family) and high communication overheads. Nevertheless, these results also demonstrate that *when CPUs on graph servers are fully saturated with the graph computation, large gains can be obtained by running tensor computation in Lambdas that fully overlap with CPU tasks!*

To compute the cost breakdown in Figure 10(b), we simply calculated the total amounts of time for Lambdas and GSes for each of the five Dorylus variants and used these times to compute the costs of Lambdas and servers. Due to Dorylus' effective use of Lambdas, we were able to run a large number of Lambdas for the forward and backward pass. As such, the cost of Lambdas is about the same as the cost of CPU servers.

8 Related Work

Dorylus is the first system that successfully uses tiny Lambda threads to train a GNN by exploiting various graph-related optimizations. There are three categories of techniques in par-

allelization (§8.1), GNN training (§8.2), and graph systems (§8.3).

8.1 Parallel Computation for Model Training

How to exploit parallelism in model training is a topic that has been extensively studied. There are two major dimensions in how to effectively parallelize the training work: (1) what to partition and (2) how to synchronize between workers.

What to Partition. The most straightforward parallelism model is *data parallelism* [8, 14, 16, 24, 72, 73, 76, 99], where *inputs* are partitioned and processed by individual workers. Each worker learns parameters (weights) from its own portion of inputs and periodically shares its parameters with other workers to obtain a global view. Both share-memory systems [8, 73, 24] and distributed systems [49, 99, 13] have been developed for data-parallel training. Another parallelization strategy is to partition the work, often referred to as *model parallelism* [61] where the operators in a model are partitioned and each worker evaluates and updates only a subset of parameters *w.r.t.* its model partition for all inputs.

A recent line of work develops techniques for *hybrid parallelism* [63, 31, 36, 44]. PipeDream [63] adds pipelining into model parallelism to fully utilize compute without introducing significant stalls. Although Dorylus also uses pipelining, tasks on a Dorylus pipeline are much finer-grained. For example, instead of splitting a model into layers, we construct graph and tensor tasks in such a way that graph tasks can be parallelized on graph servers, while each tensor task is small enough to fit into a Lambda's resource profile. Dorylus uses pipelining to overlap graph and tensor computations specifically to mitigate Lambdas' network latency. FlexFlow [36] automatically splits an iteration along four dimensions.

How Workers Synchronize. When workers work on different portions of inputs (*i.e.*, data parallelism), they need to share their learned parameters with other workers. Parameter updating requires synchronization between workers. For share-memory systems, they often rely on primitives such as `all_reduce` [8] that broadcasts each worker's parameters to all other workers. Distributed systems including Dorylus use parameter servers [49, 99, 13], which periodically communicate with workers for updating parameters. The most commonly-used approach for synchronization is the bulk synchronous parallel (BSP) model, which poses a barrier at the end of each epoch. All workers need to wait for gradients from other workers at the barrier. Wait-free backpropagation [99] is an optimization of the BSP model.

Since synchronous training often introduces computation stalls, *asynchronous* training [8, 16] has been proposed to reduce such stalls — each worker proceeds with the next input minibatch before receiving the gradients from the previous epoch. An asynchronous approach reduces time needed for each epoch at the cost of increased epochs to reach particular target accuracy. This is because allowing workers to use parameters learned in epoch m to perform forward computations in epoch n ($n \neq m$) leads to *statistical inefficiency*. This problem can be mitigated with a hybrid approach such as *bounded staleness* [63, 15, 82, 65].

8.2 GNN Training and Graph Systems

As the GNN family keeps growing [91, 96, 18, 47, 95, 103, 51, 35, 94, 98], developing efficient and scalable GNN training systems becomes popular. GraphSage [25] uses graph sampling, NeuGraph [55] extends GNN training to multiple GPUs, and RoC [34] uses dynamic graph partitioning to achieve efficiency. Other systems that can scale to large graphs are all based on sampling [94, 98].

Programming frameworks such as DGL [17] have been proposed to create a graph-parallel interface (*i.e.*, GAS) for developers to easily mix graph operations with NNs. However, such frameworks still represent the graph as a matrix and push it to an underlying training framework such as TensorFlow for training. We solve this fundamental scalability problem with a ground-up system redesign that separates the graph computation from the tensor computation.

8.3 Graph-Parallel Systems

There exists a body of work on scalable and efficient graph systems of many kinds: single-machine share-memory systems [75, 64, 21, 59, 58], disk-based out-of-core systems [46, 70, 105, 87, 52, 102, 86, 26, 84, 56, 79, 1, 88], and distributed systems [57, 54, 23, 10, 69, 11, 104, 101, 74, 81, 62, 90, 7, 80, 83]. These systems were built on top of a graph-parallel computation model, whether it is vertex-centric or edge-centric. Inspired by these systems, Dorylus formulates operations involving the graph structure as graph-parallel computation and runs it on CPU servers for scalability.

9 Conclusion

Dorylus is a distributed GNN training system that scales to large billion-edge graphs with low-cost cloud resources. We found that CPU servers, in general, offer more performance per dollar than GPU servers for large sparse graphs. Adding Lambdas added $2.75\times$ more performance-per-dollar than CPU only servers, and $4.83\times$ more than GPU only servers. Compared to existing sampling-based systems Dorylus is up to $3.8\times$ faster and $10.7\times$ cheaper. Based on the trends we observed Dorylus can scale to even larger graphs than we evaluated, offering even higher values.

Acknowledgments

We thank the anonymous reviewers for their comments. We are grateful to our shepherd Amar Phanishayee for his feedback. This work is supported by NSF grants CCF-1629126, CNS-1703598, CCF-1723773, CNS-1763172, CCF-1764077, CNS-1907352, CNS-1901510, CNS-1943621, CHS-1956322, CNS-2007737, CNS-2006437, CNS-2106838, ONR grants N00014-16-1-2913 and N00014-18-1-2037, as well as a Sloan Fellowship.

A Artifact Appendix

A.1 Artifact Summary

Dorylus is a distributed GNN training system that scales to large billion-edge graphs using cheap cloud resources—specifically CPU servers and serverless threads. It launches a set of graph servers which are used for processing graph data and doing operations such as gather and scatter. In addition, parameter servers hold the weights for the model. It can be configured to run with multiple different backends, such as a pure CPU backend and a GPU backend. By separating the graph and tensor components of a graph neural network Dorylus is able to effectively utilize serverless threads by providing a deep asynchronous-parallel pipeline in which tensor and graph operations are overlapped. By doing this Dorylus significantly improves the performance-per-dollar of serverless training over both the CPU and GPU backends.

A.2 Artifact Check-list

- **Hardware: AWS cloud account**
- **Public link:** `https://github.com/uclasystem/dorylus`
- **Code licenses: The GNU General Public License (GPL)**

A.3 Description

A.3.1 Dorylus's Codebase

Dorylus contains the following three components:

- The Graph Server which performs graph operations and manages Lambda threads (which can also use CPU and GPU backends)
- The Weight Server which holds the model parameters and sends them to the workers
- The Lambda functions which can be uploaded to AWS to be used during training

A.3.2 Deploying Dorylus

To build Dorylus, the first step is to make sure you have the following dependencies installed on your local machine:

- `awscli`
- `python3-venv`

Make sure to run `aws configure` and set up your credentials to allow you to have access to AWS services. Once these are installed, we need to download the code and setup the environment:

```
git clone
git@github.com:uclasystem/dorylus.git

cd dorylus/
git checkout v1.0 # artifact tag

python3 -m venv venv
source venv/bin/activate
pip install -U pip
pip install -r requirements.txt
```

Set Up the Cluster. We now discuss how to setup the cluster with all different roles. To do this, we use the `ec2man` python module. To start, setup the profile in the following way:

```
$ vim ec2man/profile

default # Profile from ~/.aws/credentials
ubuntu # Cluster username
${HOME}/.ssh/id_rsa # Path to SSH key
us-east-2 # AWS region
```

As mentioned previously, we work with two types of workers which we call 'contexts', specifically graph and weight servers. To add machines to these two contexts, we use one of the following commands:

```
python -m ec2man allocate --ami [AMI]
  --type [ec2 type] --cnt [#servers]
  --sg [security group]
  --ctx [weight|graph]

python -m ec2man add [graph|weight]
  [list of ec2 ids]
```

Run the first command with an AMI ID that presents a fresh install of Ubuntu, ideally with about 36 GB of storage. Alternatively if you have created instances already, say 4 graph servers you can add them to the module using the `add` command with a list of their IDs. Finally, run the command `python -m ec2man setup` to get the data about the instances so they can be managed by the module. To make sure everything is setup correctly, try SSHing into graph server 0 using `python -m ec2man graph 0 ssh`.

Building Dorylus. The next step is to make sure all dependencies are installed to build Dorylus on the cluster machines. To do this, run the following commands:

```
local$ ./gnnman/send-source [--force]
  # '--force' removes existing code

local$ ./gnnman/install-dep
```

This will sync the source code with the nodes on the cluster. Then, it will install all dependencies required to build Dorylus.

If this fails for some reason you may need to ssh into each node, move into the `dorylus/gnnman/helpers` directory, and run:

```
remote$ ./graphserver.install
remote$ ./weightserver.install
```

Parameter Files. There are a number of parameter files relating to things such as the ports used during training. Most of these will be fine as they are and should only be changed if there is a conflict.

Compiling the Code. To build and synchronize the code on all nodes in the cluster run:

```
local$ ./gnnman/setup-cluster
local$ ./gnnman/build-system
  [graph|weight] [cpu|gpu]
```

The first command sets up each node of the cluster to be aware of each other. This is important as we only build the code on node 0 and distribute it to other nodes. The second command runs CMake to build the actual system. Not specifying a context builds for all contexts. Not specifying either `cpu` or `gpu` as the backend builds the serverless version.

Setting up Lambda Functions. To install the Lambda functions, you can SSH into one of the weight or graph servers. Once there, run the following commands:

```
# Install the Lambda dependencies
remote$ ./funcs/manage-funcs.install

# Build and upload the function to the cloud
remote$ cd src/funcs
remote$ ./<function-name>/upload-func
```

A.3.3 Preparing the Data

There are 4 main inputs to Dorylus:

- The graph structure
- Graph partition info
- Input features
- Training labels

Graph Input. To prepare an input graph for Dorylus, the format should be a binary edge list with vertices numbered from 0 to $|V|$ with no breaks using 4 byte values. The file should be named `graph.bsnap`.

Partition Info. Dorylus uses edge-cut partitioning. While we do limit partitioning to edge-cuts, we allow flexibility in how the edge cut is implemented by partitioning at runtime. Provide a text file that lists partition assignments line by line, where each line number corresponds to the vertex ID and the number is the partition to which it is assigned. The file should be called `graph.bsnap.parts`.

Input Features. The input features take the form of a tensor of size $|V| \times d$ where d is the number of input features. The file should be binary and take the format of:

```
[numFeats][v0_feats][v1_feats][v2_feats]...
```

The file should be called `features.bsnap`.

Training Labels. The labels file should be binary and take the form:

```
[numLabels][label0][label1]...
```

This file should be called `labels.bsnap`.

Preparing the NFS Server. On an NFS server setup the dataset in the following format under a directory called `/mnt/filepool/`. If the dataset we are preparing is called `amazon`, the directory structure would look like this:

```
amazon
|-- features.bsnap
|-- graph.bsnap
|-- labels.bsnap
|-- parts_<#partitions>/
  |-- graph.bsnap.edges
  |-- graph.bsnap.parts
```

where `graph.bsnap.edges` is a symlink to `../graph.bsnap`. Use the `add` command from above to add the NFS server to a special context called `nfs` so that `ec2man` knows where to look for it. Finally, run

```
local$ ./gnnman/mount-nfs-server
```

A.3.4 Running Dorylus.

Once the cluster has been setup, the code compiled, the Lambda functions installed, and the datasets prepared, we can run Dorylus. To run it use the following command from the `dorylus/` directory on your local machine:

```
# <dataset>: the dataset you prepared
# --l: the #lambdas/server
# --p: enable asynchronous pipelining
# --s: degree of staleness
# [cpu|gpu]: backend to use (blank means lambda)

./run/run-dorylus <dataset>
  [--l=#lambdas] [--lr=learning_rate]
  [--p] [--s=staleness] [cpu|gpu]
```

You will see the output of the Graph Servers, but can see the output of both the Graph and Weight Servers in `graphserver-out.txt` and `weightserver-out.txt`. More details of Dorylus's installation and deployment can be found in Dorylus's code repository.

References

[1] Z. Ai, M. Zhang, Y. Wu, X. Qian, K. Chen, and W. Zheng. Squeezing out all the value of loaded data: An out-of-core graph processing system with reduced disk i/o. In *USENIX ATC*, pages 125–137, 2017.

[2] M. Allamanis, M. Brockschmidt, and M. Khademi. Learning to represent programs with graphs. In *ICLR*, 2018.

[3] Amazon. AWS Lambda Pricing. https://aws.amazon.com/lambda/pricing/, 2020.

[4] A. AWS. Announcing improved vpc networking for aws lambda functions. https://aws.amazon.com/blogs/compute/announcing-improved-vpc-networking-for-aws-lambda-functions/, 2019.

[5] M. Balog, A. L. Gaunt, M. Brockschmidt, S. Nowozin, and D. Tarlow. Deepcoder: Learning to write programs. In *ICLR*, 2017.

[6] X. Bresson and T. Laurent. Residual gated graph convnets. *CoRR*, abs/1711.07553, 2017.

[7] Y. Bu, V. Borkar, J. Jia, M. J. Carey, and T. Condie. Pregelix: Big(ger) graph analytics on a dataflow engine. *Proc. VLDB Endow.*, 8(2):161–172, Oct. 2014.

[8] J. Chen, R. Monga, S. Bengio, and R. Jozefowicz. Revisiting distributed synchronous sgd. In *ICLR Workshop Track*, 2016.

[9] J. Chen, J. Zhu, and L. Song. Stochastic training of graph convolutional networks with variance reduction. In J. Dy and A. Krause, editors, *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 942–950, Stockholmsmässan, Stockholm Sweden, 10–15 Jul 2018. PMLR.

[10] R. Chen, X. Ding, P. Wang, H. Chen, B. Zang, and H. Guan. Computation and communication efficient graph processing with distributed immutable view. In *HPDC*, pages 215–226, 2014.

[11] R. Chen, J. Shi, Y. Chen, and H. Chen. PowerLyra: Differentiated graph computation and partitioning on skewed graphs. In *EuroSys*, pages 1:1–1:15, 2015.

[12] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer. cuDNN: Efficient primitives for deep learning, 2014.

[13] T. Chilimbi, Y. Suzue, J. Apacible, and K. Kalyanaraman. Project adam: Building an efficient and scalable deep learning training system. In *OSDI*, pages 571–582, 2014.

[14] C. Coleman, D. Kang, D. Narayanan, L. Nardi, T. Zhao, J. Zhang, P. Bailis, K. Olukotun, C. Ré, and M. Zaharia. Analysis of dawnbench, a time-to-accuracy machine learning performance benchmark. *SIGOPS Oper. Syst. Rev.*, 53(1):14–25, 2019.

[15] H. Cui, J. Cipar, Q. Ho, J. K. Kim, S. Lee, A. Kumar, J. Wei, W. Dai, G. R. Ganger, P. B. Gibbons, G. A. Gibson, and E. P. Xing. Exploiting bounded staleness to speed up big data analytics. In *USENIX ATC*, pages 37–48, June 2014.

[16] H. Cui, H. Zhang, G. R. Ganger, P. B. Gibbons, and E. P. Xing. GeePS: Scalable deep learning on distributed GPUs with a gpu-specialized parameter server. In *EuroSys*, 2016.

[17] DeepGraphLibrary. Why DGL? https://www.dgl.ai/pages/about.html, 2018.

[18] M. Defferrard, X. Bresson, and P. Vandergheynst. Convolutional neural networks on graphs with fast localized spectral filtering. In *NIPS*, pages 3844—3852, Red Hook, NY, USA, 2016.

[19] D. Duvenaud, D. Maclaurin, J. Aguilera-Iparraguirre, R. Gómez-Bombarelli, T. Hirzel, A. Aspuru-Guzik, and R. P. Adams. Convolutional networks on graphs for learning molecular fingerprints. In *NIPS*, pages 2224–2232, 2015.

[20] S. Fouladi, R. S. Wahby, B. Shacklett, K. V. Balasubramaniam, W. Zeng, R. Bhalerao, A. Sivaraman, G. Porter, and K. Winstein. Encoding, fast and slow: Low-latency video processing using thousands of tiny threads. In *NSDI*, pages 363–376, 2017.

[21] S. Fushimi, M. Kitsuregawa, and H. Tanaka. An overview of the system software of A parallel relational database machine GRACE. In *VLDB*, pages 209–219, 1986.

[22] X. Glorot and Y. Bengio. Understanding the difficulty of training deep feedforward neural networks. *AISTATS*, pages 249–256, 2010.

[23] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *OSDI*, pages 17–30, 2012.

[24] P. Goyal, P. Dollár, R. B. Girshick, P. Noordhuis, L. Wesolowski, A. Kyrola, A. Tulloch, Y. Jia, and K. He. Accurate, large minibatch SGD: training ImageNet in 1 hour. *CoRR*, abs/1706.02677, 2017.

[25] W. L. Hamilton, Z. Ying, and J. Leskovec. Inductive representation learning on large graphs. In *NIPS*, pages 1024–1034, 2017.

[26] W.-S. Han, S. Lee, K. Park, J.-H. Lee, M.-S. Kim, J. Kim, and H. Yu. TurboGraph: A fast parallel graph engine handling billion-scale graphs in a single PC. In *KDD*, pages 77–85, 2013.

[27] K. He, X. Zhang, S. Ren, and J. Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *ICCV*, pages 1026–1034, 2015.

[28] R. He and J. McAuley. Ups and downs: Modeling the visual evolution of fashion trends with one-class collaborative filtering. In *WWW*, pages 507–517, 2016.

[29] J. M. Hellerstein, J. M. Faleiro, J. Gonzalez, J. Schleier-Smith, V. Sreekanti, A. Tumanov, and C. Wu. Serverless computing: One step forward, two steps back. In *CIDR*, 2019.

[30] Q. Ho, J. Cipar, H. Cui, J. K. Kim, S. Lee, P. B. Gibbons, G. A. Gibson, G. R. Ganger, and E. P. Xing. More effective distributed ml via a stale synchronous parallel parameter server. In *NIPS*, pages 1223–1231, Red Hook, NY, USA, 2013.

[31] Y. Huang, Y. Cheng, D. Chen, H. Lee, J. Ngiam, Q. V. Le, and Z. Chen. Gpipe: Efficient training of giant neural networks using pipeline parallelism. *CoRR*, abs/1811.06965, 2018.

[32] C. Huyen. Key trends from NeurIPS 2019. https://huyenchip.com/2019/12/18/key-trends-neurips-2019.html, 2019.

[33] P. Jain, A. Jain, A. Nrusimha, A. Gholami, P. Abbeel, J. Gonzalez, K. Keutzer, and I. Stoica. Checkmate: Breaking the memory wall with optimal tensor rematerialization. In *MLSys*, pages 497–511, 2020.

[34] Z. Jia, S. Lin, M. Gao, M. Zaharia, and A. Aiken. Improving the accuracy, scalability, and performance of graph neural networks with Roc. In *MLSys*, 2020.

[35] Z. Jia, S. Lin, R. Ying, J. You, J. Leskovec, and A. Aiken. Redundancy-free computation for graph neural networks. In *KDD*, pages 997–1005, 2020.

[36] Z. Jia, M. Zaharia, and A. Aiken. Beyond data and model parallelism for deep neural networks. In *MLSys*, 2019.

[37] E. Jonas, Q. Pu, S. Venkataraman, I. Stoica, and B. Recht. Occupy the cloud: Distributed computing for the 99%. In *SoCC*, pages 445–451, 2017.

[38] J. Kiefer and J. Wolfowitz. Stochastic estimation of the maximum of a regression function. *Annals of Mathematical Statistics*, 23:462–466, 1952.

[39] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization, 2014.

[40] T. N. Kipf and M. Welling. Semi-supervised classification with graph convolutional networks. In *ICLR*, 2017.

[41] M. Kirisame, S. Lyubomirsky, A. Haan, J. Brennan, M. He, J. Roesch, T. Chen, and Z. Tatlock. Dynamic tensor rematerialization. In *ICLR*, 2021.

[42] A. Klimovic, Y. Wang, C. Kozyrakis, P. Stuedi, J. Pfefferle, and A. Trivedi. Understanding ephemeral storage for serverless analytics. In *USENIX ATC*, pages 789–794, 2018.

[43] A. Klimovic, Y. Wang, P. Stuedi, A. Trivedi, J. Pfefferle, and C. Kozyrakis. Pocket: Elastic ephemeral storage for serverless analytics. In *OSDI*, pages 427–444, 2018.

[44] A. Krizhevsky. One weird trick for parallelizing convolutional neural networks. *CoRR*, abs/1404.5997, 2014.

[45] M. Kustosz and B. Osinski. Trends and fads in machine learning – topics on the rise and in decline in ICLR submissions. https://deepsense.ai/key-findings-from-the-international-conference-on-learning-representations-iclr/, 2020.

[46] A. Kyrola, G. Blelloch, and C. Guestrin. GraphChi: Large-Scale Graph Computation on Just a PC. In *OSDI*, pages 31–46, 2012.

[47] J. B. Lee, R. A. Rossi, X. Kong, S. Kim, E. Koh, and A. Rao. Graph convolutional networks with motif-based attention. In *CIKM*, pages 499–508, 2019.

[48] J. Leskovec. Stanford network analysis project. https://snap.stanford.edu/, 2020.

[49] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B.-Y. Su. Scaling distributed machine learning with the parameter server. In *OSDI*, pages 583–598, 2014.

[50] Y. Li, D. Tarlow, M. Brockschmidt, and R. Zemel. Gated graph sequence neural networks. In *ICLR*, 2016.

[51] Y. Li, D. Tarlow, M. Brockschmidt, and R. S. Zemel. Gated graph sequence neural networks. In Y. Bengio and Y. LeCun, editors, *ICLR*, 2016.

[52] Z. Lin, M. Kahng, K. M. Sabrin, D. H. P. Chau, H. Lee, , and U. Kang. MMap: Fast billion-scale graph computation on a pc via memory mapping. In *BigData*, pages 159–164, 2014.

[53] Q. Liu, M. Nickel, and D. Kiela. Hyperbolic graph neural networks. In *NIPS*, pages 8230–8241. Curran Associates, Inc., 2019.

[54] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. GraphLab: A new framework for parallel machine learning. In *UAI*, pages 340–349, 2010.

[55] L. Ma, Z. Yang, Y. Miao, J. Xue, M. Wu, L. Zhou, and Y. Dai. NeuGraph: Parallel deep neural network computation on large graphs. In *USENIX ATC*, pages 443–457, 2019.

[56] S. Maass, C. Min, S. Kashyap, W. Kang, M. Kumar, and T. Kim. Mosaic: Processing a trillion-edge graph on a single machine. In *EuroSys*, pages 527–543, 2017.

[57] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *SIGMOD*, pages 135–146, 2010.

[58] M. Mariappan, J. Che, and K. Vora. DZiG: Sparsity-Aware Incremental Processing of Streaming Graphs. In *EuroSys*, page 83–98, 2021.

[59] M. Mariappan and K. Vora. GraphBolt: Dependency-Driven Synchronous Processing of Streaming Graphs. In *EuroSys*, page 25:1–25:16, 2019.

[60] J. McAuley, C. Targett, Q. Shi, and A. van den Hengel. Image-based recommendations on styles and substitutes. In *SIGIR*, pages 43–52, 2015.

[61] A. Mirhoseini, H. Pham, Q. V. Le, B. Steiner, R. Larsen, Y. Zhou, N. Kumar, M. Norouzi, S. Bengio, and J. Dean. Device placement optimization with reinforcement learning. In *ICML*, pages 2430—-2439, 2017.

[62] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. Naiad: A timely dataflow system. In *SOSP*, pages 439–455, 2013.

[63] D. Narayanan, A. Harlap, A. Phanishayee, V. Seshadri, N. R. Devanur, G. R. Ganger, P. B. Gibbons, and M. Zaharia. PipeDream: Generalized pipeline parallelism for DNN training. In *SOSP*, page 1–15, 2019.

[64] D. Nguyen, A. Lenharth, and K. Pingali. A lightweight infrastructure for graph analytics. In *SOSP*, pages 456–471, 2013.

[65] F. Niu, B. Recht, C. Re, and S. J. Wright. HOGWILD! a lock-free approach to parallelizing stochastic gradient descent. In *NIPS*, pages 693—-701, 2011.

[66] NVIDIA. The cuSPARSE CUDA toolkit. https://docs.nvidia.com/cuda/cusparse/index.html, 2020.

[67] N. Peng, H. Poon, C. Quirk, K. Toutanova, and W. Yih. Cross-sentence n-ary relation extraction with graph LSTMs. *TACL*, 5:101–115, 2017.

[68] Reddit. The reddit datasets. https://www.reddit.com/r/datasets/, 2020.

[69] A. Roy, L. Bindschaedler, J. Malicevic, and W. Zwaenepoel. Chaos: Scale-out graph processing from secondary storage. In *SOSP*, pages 410–424, 2015.

[70] A. Roy, I. Mihailovic, and W. Zwaenepoel. X-Stream: Edge-centric graph processing using streaming partitions. In *SOSP*, pages 472–488, 2013.

[71] F. Scarselli and et al. The graph neural network model. *IEEE Trans. Neur. Netw.*, 20(1):61–80, Jan. 2009.

[72] F. Seide, H. Fu, J. Droppo, G. Li, and D. Yu. 1-bit stochastic gradient descent and application to data-parallel distributed training of speech dnns. In *Interspeech 2014*, September 2014.

[73] F. Seide, H. Fu, J. Droppo, G. Li, and D. Yu. On parallelizability of stochastic gradient descent for speech dnns. In *ICASSP*, pages 235–239, 2014.

[74] J. Shi, Y. Yao, R. Chen, H. Chen, and F. Li. Fast and concurrent RDF queries with rdma-based distributed graph exploration. In *USENIX ATC*, pages 317–332, 2016.

[75] J. Shun and G. E. Blelloch. Ligra: A lightweight graph processing framework for shared memory. In *PPoPP*, pages 135–146, 2013.

[76] R. Thakur, R. Rabenseifner, and W. Gropp. Optimization of collective communication operations in mpich. *Int. J. High Perform. Comput. Appl.*, 19(1):49—-66, 2005.

[77] J. Thorpe, Y. Qiao, J. Eyolfson, S. Teng, G. Hu, Z. Jia, J. Wei, K. Vora, R. Netravali, M. Kim, and G. H. Xu. Dorylus: affordable, scalable, and accurate GNN training with distributed CPU servers and serverless threads. https://arxiv.org/abs/2105.11118, 2021.

[78] P. Velickovic, G. Cucurull, A. Casanova, A. Romero, P. Liò, and Y. Bengio. Graph attention networks. In *ICLR*, 2018.

[79] K. Vora. LUMOS: Dependency-driven disk-based graph processing. In *USENIX ATC*, pages 429–442, 2019.

[80] K. Vora, R. Gupta, and G. Xu. Synergistic analysis of evolving graphs. *ACM Trans. Archit. Code Optim.*, 13(4):32:1–32:27, 2016.

[81] K. Vora, R. Gupta, and G. Xu. KickStarter: Fast and accurate computations on streaming graphs via trimmed approximations. In *ASPLOS*, pages 237–251, 2017.

[82] K. Vora, S. C. Koduru, and R. Gupta. ASPIRE: Exploiting asynchronous parallelism in iterative algorithms using a relaxed consistency based dsm. In *OOPSLA*, pages 861–878, 2014.

[83] K. Vora, C. Tian, R. Gupta, and Z. Hu. CoRAL: Confined Recovery in Distributed Asynchronous Graph Processing. In *ASPLOS*, page 223–236, 2017.

[84] K. Vora, G. Xu, and R. Gupta. Load the edges you need: A generic I/O optimization for disk-based graph processing. In *USENIX ATC*, pages 507–522, 2016.

[85] A. D. Vose, J. Balma, D. Farnsworth, K. Anderson, and Y. K. Peterson. PharML.Bind: Pharmacologic machine learning for protein-ligand interactions, 2019.

[86] K. Wang, A. Hussain, Z. Zuo, G. Xu, and A. A. Sani. Graspan: A single-machine disk-based graph system for interprocedural static analyses of large-scale systems code. In *ASPLOS*, pages 389–404, 2017.

[87] K. Wang, G. Xu, Z. Su, and Y. D. Liu. GraphQ: Graph query processing with abstraction refinement—programmable and budget-aware analytical queries over very large graphs on a single PC. In *USENIX ATC*, pages 387–401, 2015.

[88] K. Wang, Z. Zuo, J. Thorpe, T. Q. Nguyen, and G. H. Xu. Rstream: Marrying relational algebra with streaming for efficient graph mining on a single machine. In *OSDI*, pages 763–782, 2018.

[89] M. Wang, L. Yu, D. Zheng, Q. Gan, Y. Gai, Z. Ye, M. Li, J. Zhou, Q. Huang, C. Ma, Z. Huang, Q. Guo, H. Zhang, H. Lin, J. Zhao, J. Li, A. J. Smola, and Z. Zhang. Deep graph library: Towards efficient and scalable deep learning on graphs. *CoRR*, abs/1909.01315, 2019.

[90] M. Wu, F. Yang, J. Xue, W. Xiao, Y. Miao, L. Wei, H. Lin, Y. Dai, and L. Zhou. GraM: Scaling graph computation to the trillions. In *SoCC*, pages 408–421, 2015.

[91] S. Wu, Y. Tang, Y. Zhu, L. Wang, X. Xie, and T. Tan. Session-based recommendation with graph neural networks. *AAAI*, 33:346–353, Jul 2019.

[92] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, and P. S. Yu. A comprehensive survey on graph neural networks. *CoRR*, abs/1901.00596, 2019.

[93] Z. Xianyi and M. Kroeker. OpenBLAS. https://www.openblas.net, 2019.

[94] H. Yang. Aligraph: A comprehensive graph neural network platform. In *KDD*, pages 3165–3166, 2019.

[95] R. Ying, R. He, K. Chen, P. Eksombatchai, W. L. Hamilton, and J. Leskovec. Graph convolutional neural networks for web-scale recommender systems. In *KDD*, pages 974–983, 2018.

[96] S. Yun, M. Jeong, R. Kim, J. Kang, and H. J. Kim. Graph transformer networks. In *Annual Conference on Neural Information Processing Systems 2019*, pages 11960–11970, 2019.

[97] ZeroMQ. ZeroMQ networking library for C++. https://zeromq.org/, 2020.

[98] D. Zhang, X. Huang, Z. Liu, J. Zhou, Z. Hu, X. Song, Z. Ge, L. Wang, Z. Zhang, and Y. Qi. AGL: A scalable system for industrial-purpose graph machine learning. *Proc. VLDB Endow.*, 13(12):3125–3137, 2020.

[99] H. Zhang, Z. Zheng, S. Xu, W. Dai, Q. Ho, X. Liang, Z. Hu, J. Wei, P. Xie, and E. P. Xing. Poseidon: An efficient communication architecture for distributed deep learning on GPU clusters. In *USENIX ATC*, pages 181—-193, 2017.

[100] J. Zhang, X. Shi, J. Xie, H. Ma, I. King, and D. Yeung. GaAN: Gated attention networks for learning on large and spatiotemporal graphs. In A. Globerson and R. Silva, editors, *UAI*, pages 339–349, 2018.

[101] M. Zhang, Y. Wu, K. Chen, X. Qian, X. Li, and W. Zheng. Exploring the hidden dimension in graph processing. In *OSDI*, pages 285–300, 2016.

[102] D. Zheng, D. Mhembere, R. Burns, J. Vogelstein, C. E. Priebe, and A. S. Szalay. FlashGraph: processing billion-node graphs on an array of commodity ssds. In *FAST*, pages 45–58, 2015.

[103] J. Zhou, G. Cui, Z. Zhang, C. Yang, Z. Liu, and M. Sun. Graph neural networks: A review of methods and applications. *CoRR*, abs/1812.08434, 2018.

[104] X. Zhu, W. Chen, W. Zheng, and X. Ma. Gemini: A computation-centric distributed graph processing system. In *OSDI*, pages 301–316, 2016.

[105] X. Zhu, W. Han, and W. Chen. GridGraph: Large scale graph processing on a single machine using 2-level hierarchical partitioning. In *USENIX ATC*, pages 375–386, 2015.