UNIVERSITY OF CALIFORNIA

Los Angeles

Efficient, Affordable, and Scalable Deep Learning Systems

A dissertation submitted in partial satisfaction
of the requirements for the degree
Doctor of Philosophy in Computer Science

by

John Thorpe

2022

ABSTRACT OF THE DISSERTATION

Efficient, Affordable, and Scalable Deep Learning Systems

by

John Thorpe
Doctor of Philosophy in Computer Science
University of California, Los Angeles, 2022
Professor Harry Guoqing Xu, Chair

Deep Learning has become one of the most important tools in computer science in the last decade because of its ability to perform tasks that involve complex reasoning. Researchers have been pushing forward the state of the art in many ways, but we focus on two major trends that have shown promise. The first is creating models which can change based on the inputs to better learn patterns in the data. For example, with Graph Neural Networks, training the same model over two different graphs will result in models which have been specialized to the structure of the particular graph. This specialization results in the same model architectures having very different computation patterns depending on the input graph. The second is the rapidly increasing size of models, measured by the number of parameters. This increase leads to much more general and accurate machine learning agents, but requires a large amount of hardware to train efficiently. Because of these complexities, these trends result in state of the art models requiring massive amounts of computational and financial resources, limiting them to larger well-funded companies that have the funds to experiment.

Lowering the financial barrier to entry for experimentation with these models will allow many more small businesses and research labs to experiment with them. The current paradigm of machine learning focuses on throwing high-end computational power at the Deep Learning problem as the resource requirements grow. However, a key observation for addressing this problem lies in the increasingly heterogenous offering made by cloud providers. Cloud platforms offer a much more diverse set of resources than would be available to most onsite clusters allowing for a more fine-grained approach to the problem. By combining the heterogeneous offers of cloud providers with in-depth knowledge of the compute profile of deep learning models, we can achieve better *value* compared to most existing systems. Here, value is defined as the performance per dollar we can achieve.

My first system, Dorylus, a Graph Neural Network framework, provides up to **3.86x** more value than its GPU based variant running on large graphs by employing serverless threads to do asynchronous computation. My next work, Bamboo, focuses on reducing the cost of training massive pipeline-parallel models by making preemptible instances resilient and performant by intelligently introducing computation redundancy into the pipelines. Bamboo was able to provide almost **2x** the performance-per-dollar of training with full-priced non-preemptible instances, and provide **1.5x** the performance-per-dollar of existing systems designed to run on spot instances.

The dissertation of John Thorpe is approved.

Zhihao Jia

Yizhou Sun

Ravi Arun Netravali

Harry Guoqing Xu, Committee Chair

University of California, Los Angeles

2022

To my family

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

I would like to start by giving my sincerest thanks to my advisor, Harry Xu, who has helped me through the PhD program. Especially for being a particularly understanding and patient advisor. I have been working with for almost a decade since I started doing undergraduate research at UCI. I then joined him as a PhD student at UCI and moved with him to UCLA. I really appreciate the support and understanding towards the beginning of the PhD when I was still finding my footing.

I would also like to thank my other committee members, Zhihao Jia, Yizhou Sun, and Ravi Netravali for their feedback on my thesis direction. Zhihao Jia and Ravi Netravali have also been collaborators on both of my papers and have provided valuable feedback which significantly improved the papers.

I also want to give a huge thanks to both Yifan Qiao and Pengzhan Zhao. Yifan was a co-first author on Dorylus who put in a massive amount of work to help build a Graph Neural Network system from scratch. Pengzhan was co-first author on Bamboo and contributed significantly to the overall success of the project. I am incredibly thankful for all of their hard work and I have learned so much from both of them.

I also wish to show appreciation to all my collaborators across both projects, who have all contributed many different perspectives that helped shape our understanding of the problem and ensured we were creating interesting and worthwhile research. Specifically, I would like to thank Shen Teng, Guanzhou Hu, Jonathan Eyolfson, Minjia Zhang, Zhihao Jia, Jinliang Wei, Keval Vora, Ravi Netravali, and Miryung Kim.

I am also very fortunate to have been part of such an amazing and friendly lab who helped me maintain some semblance of a social life during the PhD through many dinners and lunches. I was very lucky to be surrounded by all the great people in the UCLA Systems lab: Christian Navasca, Jonathan Eyolfson, Arthi Padmanabhan, Yifan Qiao, Pengzhan Zhao, Shi Liu, Chenxi Wang, Usama Hameed, Haoran Ma, and Jiyuan Wang. Thanks for all the dinners and for helping me break out of my narrow research area and get exposure to a range of different concepts that I would not otherwise know about (and which I find quite interesting).

I would like to specifically thank Jonathan Eyolfson and Neil Agarwal as well for being huge helps during the PhD. Jon was always willing to help me solve obscure LaTeX and

C++ issues, go over my slides and papers, and generally made me a much clearer presenter, a better coder, and a better writer. Neil was always willing to talk about the finer details of projects he was not involved with which helped me work through many distributed systems bugs that I had been stuck on. He also pushed me to be a better and more well-rounded researcher through our unofficial reading group, NARG (Not Another Reading Group), as well as that time we went through a whole reinforcement learning text book on our own just to learn more about it.

I would also like to thank Joseph Brown, the graduate student affairs officer, who has been very understanding and willing to work with me as I progressed through the program. From helping me get all my classes transferred from UCI at the start to just always being responsive and helpful whenever I had questions he made the process of the PhD much more enjoyable.

I would also like to thank my family who have constantly asked me when I would be done but have otherwise been very supportive and helpful during this process. Early on in the program when I had doubts about continuing, they were there to talk through my reasoning with me without judgment which helped me feel more focused and reassured. My sisters, Molly and Bridget, in particular have always been incredibly supportive and are the first people I talk to whenever I am making a big decision. Thanks to my extended family as well, the Thorpe's and the Curry's, for being incredibly supportive and excited about my journey through the PhD.

A huge thanks to my all of my friends who have been supportive along the way. My friend Rafael Quiroz has also been a huge help along the way, allowing me to rant in great detail about things which he may or may not have understood so that I could process them. I really appreciate that. Thanks to Brett Settle and Xander Thompson for our (somewhat) weekly meetings and our talks about tech and random things going on in our lives. And thanks to my friend Matt for having some very genuine talks with me about my time in the program and just generally being a good person.

Also, a very special thanks to my partner, Arthi, who I met in the PhD program and provided me constant support and understanding during some of the most difficult points in the program.

This work contains modified versions of published and co-authored work. Chapter 2 is a modified version of [134] and chapter 3 is a modified version of [135] (PI: Harry Guoqing Xu).

VITA

John Thorpe

EDUCATION

University of California, Los Angeles 2018 - 2022
– Ph.D. in Computer Science *(expected)*

University of California, Irvine
– Ph.D. in Computer Science 2017 - 2018
Advisor: Prof. Harry Xu *(moved to UCLA)*

University of California, Irvine
– B.S. in Computer Science 2012 - 2017
Magna Cum Laude
– B.A. in Economics 2012 - 2017
Magna Cum Laude

EXPERIENCE

Research Assistant September 2018 - June 2022
University of California, Los Angeles, CA

Research Intern Summer 2020
Microsoft Inc

Research Intern Summer 2018
Microsoft Inc

Research Assistant September 2017 - June 2018
University of California, Irvine, CA

PUBLICATIONS

John Thorpe, Pengzhan Zhao, Jonathan Eyolfson, Yifan Qiao, Zhihao Jia, Minjia Zhang, Ravi Netravali, and Guoqing Harry Xu.
Bamboo: Making Preemptible Instances Resilient for Affordable Training of Large DNNs
arXiv

John Thorpe, Yifan Qiao, Jonathan Eyolfson, Shen Teng, Guanzhou Hu, Zhihao Jia, Jinliang Wei, Keval Vora, Ravi Netravali, Miryung Kim, and Guoqing Harry Xu.

Dorylus: Affordable, Scalable, and Accurate GNN Training with Distributed CPU Servers and Serverless Threads
The 15th USENIX Symposium on Operating Systems Design and Implementation (OSDI)

Zhiqiang Zuo, **John Thorpe**, Yifei Wang, Qiuhong Pan, Shenming Lu, Kai Wang, Guoqing Harry Xu, Linzhang Wang, and Xuandong Li.
Grapple: A Graph System for Static Finite-State Property Checking of Large-Scale Systems Code
The Fourteenth EuroSys Conference 2019 (EuroSys)

Kai Wang, Zhiqiang Zuo, **John Thorpe**, Tien Quang Nguyen, and Guoqing Harry Xu.
RStream: Marrying Relational Algebra with Streaming for Efficient Graph Mining on a Single Machine
The 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)

CHAPTER 1

Introduction

Over the past decade, Deep Learning has quickly become one of the most important workloads in both industry and academia. It has shown unparalleled performance on tasks that require complex reasoning, such as object detection which involves drawing bounding boxes on objects or contextual question answering in which an agent has to answer questions about a passage it just ”read”. This ability to tackle tasks typically thought of as requiring human intelligence has lead to its application in many different areas. Object detection models such as YOLO are currently deployed to analyze footage from traffic cameras [3], transformer based NLP models such as BERT are currently used for improved language understanding in search [2], and certain networks called Graph Neural Networks have been applied for product recommendation systems using edge prediction on graphs [161].

Given their current success across a wide range of domains, there is a huge demand to increase their accuracy, generality, and efficiency. Companies and academic institutions are trying to build models that can handle even more complex tasks with better accuracy which has to lead models becoming larger and more complicated. Two of the trends that have emerged to achieve this goal are the incorporation of the structural dependencies of data into the NN computation, such as in Tree-LSTMs [132] or Graph Neural Networks (GNNs) [60, 65, 83, 85, 116], and the exponential increase in the number of parameters networks use.

1.1 Background and Motivation

Including data dependencies into the model itself increases the expressiveness of the models, however it changes a lot of the assumptions typically made about model training. Using most neural networks, the computation remains the same regardless of the input data which leads to homogeneous computation that can be partitioned more easily and distributed among workers as the scale of the data or model increases. In the case of networks such as Mixture-of-Experts models [47], only subsets of the network are run depending on the specific input sample, leading to different computation patterns throughout the course of training. In the case of networks that incorporate dependencies of the input data, such as Graph Neural Networks or Tree-LSTMs, the structure of the graph or of a sentence can change the distribution of the workload. For Graph Neural Networks specifically this becomes problematic as they can have graphs with billions of edges [158] that are unevenly distributed among the vertices, requiring more complicated techniques to partition and scale effectively.

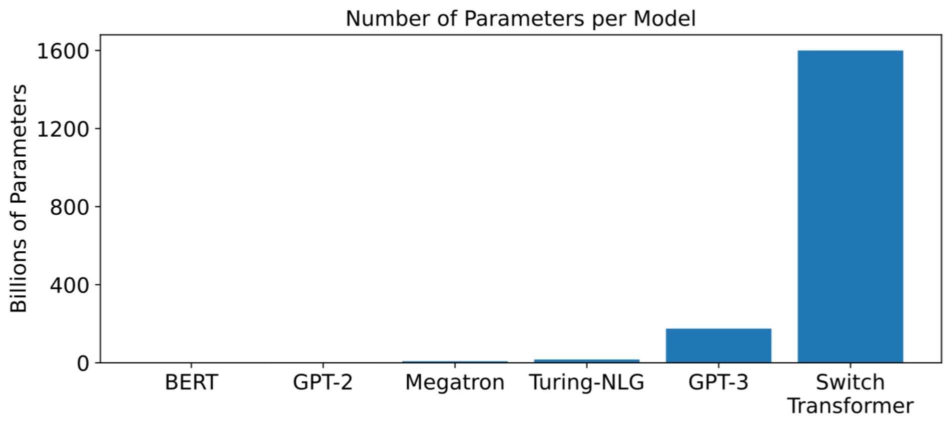


Figure 1.1: Exponentially increasing models sizes

The size of networks has also been increasing exponentially which presents a problem for the limited memory capacity of modern accelerators. AlexNet, the neural network that catalyzed the modern age of deep neural networks had 60 million parameters [70]. Modern models dwarf this with recent networks like GPT-3 having a version with 175 billion parameters [17], a work which was shortly surpassed by Google's Switch Transformer, with a massive *1.6 trillion parameters* [35]. GPT-3 demonstrates the effect of these larger models, demonstrating that they are able to generalize very well, performing near the fine-tuned state of the art models while only using zero, one, or few-shot learning. Likewise, Switch Transformer shows improved top-accuracy results for many NLP tasks, such as GLUE and SQuAD among others. The massive number of parameters leads to scalability problems however, as a typical modern GPU with 32GB of memory will be out of memory at 1.4B parameters [109].

The mainstream approach to addressing the growing computational and memory cost of models is to scale clusters horizontally and vertically, i.e. to give each worker in a cluster more resources and to simply add more workers to the cluster. This however adds a huge financial cost. Using V100 machines and using conservative estimates of the hourly cost to run these machines, the cost to train the GPT-3 model was predicted to be around $4.6 million [1]. This makes experimenting with and improving upon such state of the art models infeasible for many smaller companies and academic research groups. Bringing down this barrier to allow more researchers and groups to use state of the art models is beneficial in many ways. First, as experimenting with models becomes accessible to more people, they are more likely to discover new domains in which it could be applied and new applications that have not been previously though of. In addition, a well-known problem with these models is bias that can lead to unwanted and sometimes harmful results of a model [38, 126, 127]. If the financial barriers are decreased, a larger group of researchers would be able to focus on finding ways to introduce bias-reduction or model correction into pretraining, rather than having to rely on large companies to fix and audit the models themselves.

1.2 Key Approach

While there exists a body of work aiming to improve the scalability and resource utilization of GPU clusters [59, 85, 94, 109], the increasing financial burden of state of the art models motivates the need to focus on a metric which we call *value*. Here, value is defined as a system's *performance per dollar*. Value is an important metric to consider as users cannot always take the cheapest option if a job has to be done within a time constraint. Similarly, they cannot always choose the fastest option if it becomes prohibitively expensive.

The main observation that motivates this line of work is that the increasing heterogeneity of cloud resources makes it possible for us to make judicious selection of resources for tasks on which they offer the best value. Prior work has shown that the dollar normalized throughput of some workloads changes depending on the accelerator used. For example, running a transformer based model on a V100 GPU can bring a **3.3**× increase in throughput compared to the K80 GPU. However, despite the speed up, doing so results in **0.8**× dollar-normalized throughput [95]. Another example is the emergence of smart-networking hardware which is becoming more widely available. This has the potential to speed up network bound workloads [74, 115] such as federated learning if done carefully, especially in network constrained environments. In addition to choosing between the many accelerators available, clouds also offer variations of traditional resources which have different trade-offs. Serverless threads provide CPUs in a serverless abstraction to allow easy and efficient scaling of compute resources as well as fine grained billing [9]. However, these benefits come at the expense of weaker compute and slower network [68]. Clouds also all offer some instances on a volatile spot market, subjecting them to dynamic pricing based on demand. While these instances can often be used at a discount to their full-priced counterparts, they can be preepmted if a higher priority user needs the resources.

My work aims to build systems that combine in-depth knowledge of the characteristics of model training with careful selection of compute resources to mask any potential drawbacks while maintaining accurate and performant training. By understanding the compute profiles of different models, affects of asynchrony on convergence, and other factors we can make more informed decisions about resource use, rather than simply using the highest-end accelerator and incurring unnecessarily high cost. In this thesis I will present two systems developed along this line of reasoning.

1.3 Graph Neural Networks

The first system, Dorylus, is a Graph Neural Network framework for scaling to billion-edge graphs. As mentioned above, a GNN is a type of neural network in which the graph structure is considered during the training. This structural information is incorporated through an added aggregation step which aggregates data across edges in the graph. The de facto approach for training such networks is GPUs due to their high degree of parallelism. However, GPUs, are also costly and can cause scalability bottlenecks for training. While realistic graph workloads would ideally incorporate graphs with *billions of edges*, existing systems struggle to scale to graphs with even millions of edges while using GPUs. In addition they can be expensive. Recent works such as NeuGraph and Roc train only million-edge graphs using several high-end HPC machines.

To solve the problems of expense and poor scalability our first insight was to separate the graph and tensor processing components of a GNN system. Given that GNNs can be modeled using a computation model similar to that of the GAS model from graph processing, we note that we can leverage well studied graph processing techniques to increase scalability. Given our initial insight, we noticed that the tensor computation now existed as small, bursty computations interspersed with graph computations. The short, bursty nature of the tensor computation coupled with the fact that GNN models tend to be rather small lead us to the serverless paradigm. Given that serverless computation has very low prices and a scalable infrastructure, we saw the potential for a massive reduction in cost while maintaining good efficiency compared to GPUs.

Our results show that we are able to increase the value available to users, especially as graphs grow larger. Dorylus offers 2.75× more performance-per-dollar than CPUs only. In addition, it is 1.22× faster and 4.83× cheaper than GPUs on large, sparse graphs.

1.4 Very Large Models

While we found serverless to be a good fit for out initial target of GNNs due to the smaller, shallower models and interspersed tensor computation, this does not hold for all models, especially for very large models. Models that use billions of parameters are becoming more common, and are epitomized by the transfer-based models such as GPT-3. These models consist of repeated blocks of homogeneous and dense tensor computation which requires accelerators for acceptable performance. In order to enable the use of accelerators for good performance while reducing costs, we found the right cloud resource to be spot instances. Spot instances allow a user to use cloud servers at significantly reduced prices that perform equivalently to their full-priced counterparts, except that they can be preempted when the cloud needs to free up capacity for higher priority users (people paying full price).

There are existing works which attempt to allow users to use spot instances by performing checkpointing, a common approach in environments where failures can occur. However, we found that failures can occur too often in a spot environment to be efficiently handled by checkpointing. In addition, attempts that try to allow training to continue in the case of failures by simply dropping some training samples have a similar problem. When failure rates are low they lead to an acceptable trade-off, but when failures are frequent and bulky, as happens with spot instances, they can lead to unacceptable decreases in model accuracy.

Our approach, called Bamboo, is to closely investigate one of the most common parallelization strategies for training large models, pipeline-parallelism. We intelligently introduce computation redundancy into the pipeline to allow workers to quickly recover from any failures without long pauses. We also utilize aspects of pipeline-parallelism, such as small stalls while waiting for dependencies from other workers, to minimze the overhead of redundancy.

Overall, with Bamboo we were able to provide nearly 2$\times$ as much value as the full-priced non-preemptible baseline and 1.5$\times$ as much value as checkpointing techniques that were designed to use spot instances.

CHAPTER 2

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

2.1 Introduction

Graph Neural Networks (GNN) [60, 65, 79, 83, 85, 116] are a family of NNs designed for deep learning on graph structured data [153, 169]. The most well-known model in this family is the graph convolutional network (GCN) [65], which uses the connectivity structure of the graph as the filter to perform neighborhood mixing. Other models include graph recursive network (GRN) [80, 104], graph attention network (GAT) [16, 137, 164], and graph transformer network (GTN) [159]. Due to the prevalence of graph datasets, GNNs have gained increasing popularity across diverse domains such as drug discovery [144], chemistry [33], program analysis [8, 14], and recommendation systems [151, 158]. In fact, GNN is one of the most popular topics in recent AI/ML conferences [55, 71].

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 [85] and Roc [59] 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 [112] and continue to grow [158]. 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 [43, 158]. 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 (§2.7.5) and prior work [59] 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 [20].

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 [36, 62, 68] 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 [9]). 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 [67])

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 naively for training an NN could result in significant slowdowns (e.g., 21× slowdowns for training of multi-layer perceptron NNs [51], 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 [85] – neighbor propagations (e.g., `Gather` and `Scatter`) over

[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.

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 §2.7.6), 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 2.3, §2.4) 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 [59, 94], 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 (§2.5): *weight stashing* [94] at parameter updates and *bounded staleness* at data gathering. We have formally proved the convergence of our asynchronous model in §2.5.

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`.

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.

2.2 Background

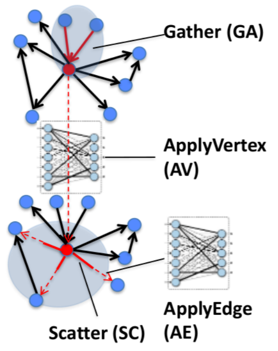
A GNN takes graph-structured data as input, where each vertex is associated with a feature vector, and outputs a feature vector for each individual vertex or the whole graph. The output feature vectors can then be used by various downstream tasks, such as, graph or vertex classification. By combining 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 [30, 148] 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) [30], which unifies a variety of GNN models with a common GAS-like interface.

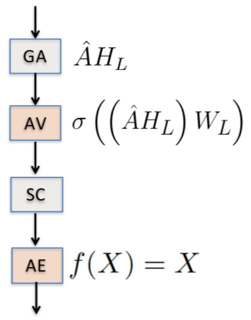
Forward Pass. To illustrate, consider graph convolutional network (GCN) as an example. GCN is the simplest and yet most popular model in the GNN family, with the following forward propagation rule for the L-th layer [65]:

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

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



(a) Computation model



(b) Annotated dataflow graph

Figure 2.1: 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.

Mapping R1 to the vertex-centric computation model is familiar to the systems community [85] — each forward layer has four components: `Gather` (GA), `ApplyVertex` (AV), `Scatter` (SC), and `ApplyEdge` (AE), as shown in Figure 2.1(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 2.1(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:

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

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.

2.3 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 (§2.4), and how Dorylus bounds the degree of asynchrony (§2.5), manages and autotunes Lambdas (§2.6).

Figure 2.2 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 [170] 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.

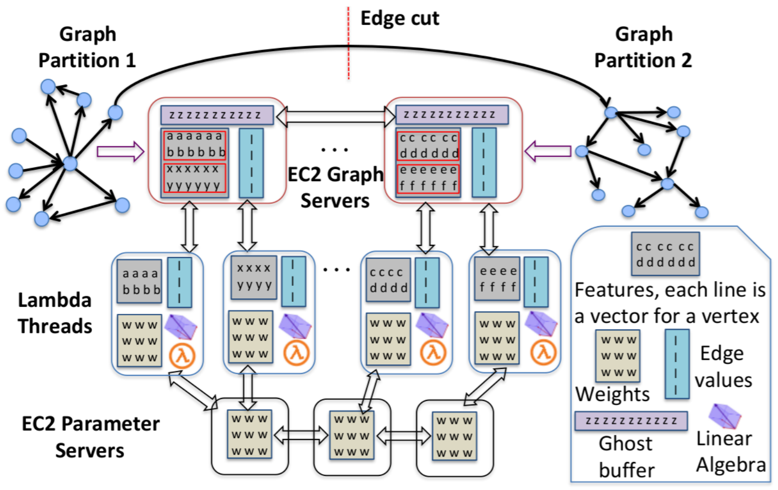


Figure 2.2: Dorylus's architecture.

2.4 Tasks and Pipelining

Fine-grained Tasks. As discussed in §2.1, 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 2.1(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 §2.2) 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.

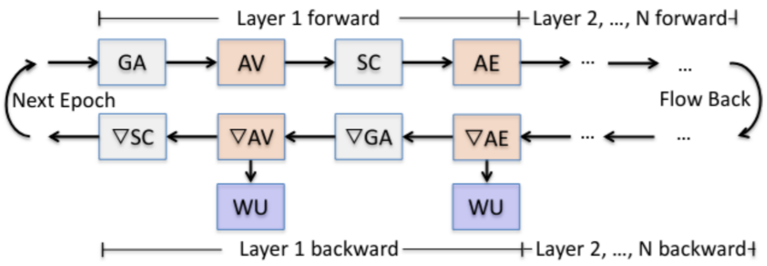


Figure 2.3: Dorylus's forward and backward dataflow with nine tasks: `Gather` (GA) and `Scatter` (SC) and their corresponding backward tasks ▽GA and ▽SC; `ApplyVertex` (AV), `ApplyEdge` (AE), and their backward tasks ▽AV and ▽AE; the weight update task `WeightUpdate` (WU).

A backward pass involves GSes, Lambdas, and PSes that *coordinate* to run a graph-augmented SGD algorithm, as specified by R2 in §2.2. 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 2.3 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 §2.5).

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.

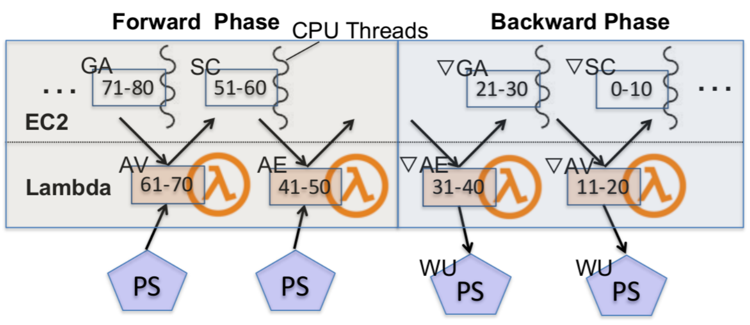


Figure 2.4: A Dorylus pipeline for an epoch: the number range (e.g., 71-80) in each box represents a particular vertex interval (i.e., minibatch); different intervals are at different locations of the pipeline and processed by different processing units: GS, Lambda, or PS.

Figure 2.4 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 $\triangledown$AE and $\triangledown$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.

2.5 Bounded Asynchrony

To unleash the full power of pipelining, Dorylus performs a unique form of *bounded asynchronous* training so that workers 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 [25, 94, 163] shows that asynchrony slows down convergence — fast-progressing minibatches may use out-of-date weights, prolonging the training time.

Bounded staleness [27, 99] 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.

2.5.1 Bounded Asynchrony at Weight Updates

To bound the degree of asynchrony for weight updates, we use *weight stashing* proposed in PipeDream [94]. 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 [78], 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.

2.5.2 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 [27, 141], 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 §2.5.3.

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).

2.5.3 Convergence Guarantee

Asynchronous weight updates with bounded staleness has been well studied, and its convergence has been proved by [52]. 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

[3] $\mathcal{L}$ is *ρ-Lipschitz smooth* if $\forall W_1, W_2, |\mathcal{L}(W_2) - \mathcal{L}(W_1) - \langle\nabla\mathcal{L}(W_1), W_2 - W_1\rangle| \le \frac{\rho}{2}\|W_2 - W_1\|_F^2$, 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.

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 $\lim_{N\rightarrow\infty} \mathbb{E}_R \left\|\nabla \mathcal{L}\left(W_R\right)\right\|^2 = 0$, indicating that asynchronous GNN training will eventually converge to a local minimum. The full-blown proof can be found in appendix §A. It mostly follows the convergence proof of the *variance reduction (VR)* algorithm in [20]. However, our proof differs from [20] 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 [20], not to mention that sampling incurs a per-epoch overhead and reduces accuracy.

2.6 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 [167] that is optimized to use AVX instructions for efficient linear algebra operations. Lambdas communicate with GSes and PSes using ZeroMQ [160].

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 [51, 67]. Although AWS has considerably improved Lambdas' network performance [12], 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 $\triangledown$AV of the last layer in the next backward pass (see Figure 2.4), we merge them into a single Lambda-based task, reducing invocations of thousands of Lambdas for each epoch and saving a round-trip communication between Lambdas and GSes.

The second optimization is tensor rematerialization [57, 66]. 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.

2.7 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 [100]. Our CUDA code includes deep learning operations such as dense layer and activation layer with cuDNN [24]. Dorylus supports common stochastic optimizations including *Xavier initialization* [39], *He initialization* [48], a *vanilla SGD optimizer* [63], and an *Adam optimizer* [64], which help training converge smoothly.

Graph	**Size ($\|V\|$, $\|E\|$)**	**# features**	**# labels**	**Avg. degree**
Reddit-small [43]	(232.9K, 114.8M)	602	41	492.9
Reddit-large [43]	(1.1M, 1.3B)	301	50	645.4
Amazon [50, 91]	(9.2M, 313.9M)	300	25	35.1
Friendster [77]	(65.6M, 3.6B)	32	50	27.5

Table 2.1: We use 4 graphs, 2 with billions of edges.

2.7.1 Experiment Setup

We experimented with four graphs, as shown in Table 2.1. `Reddit-small` and `Reddit-large` are both generated from the Reddit dataset [111]. `Amazon` is the *largest graph* in RoC's [59] 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 [59, 85]). 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) [65] and graph attention network (GAT) [159] 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 [59, 85].

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, performance, 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 2.2: 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.

2.7.2 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× as long as computation on c5. Therefore, as shown in Table 2.2 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× 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.

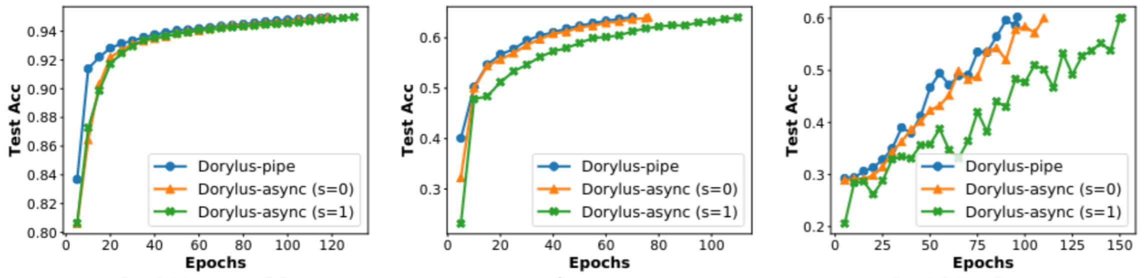
Each Lambda is a container with 0.11 vCPUs and 192 MB memory. Lambdas have a static cost of \$0.20 per 1 M requests, and a compute cost of \$0.01125/h (billed per 100 ms). This billing granularity enables serverless threads to handle short bursts of massive parallelism much better than CPU instances.

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

Table 2.3 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

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



(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 2.5: 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.

(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 §2.7.4). 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 [59]. 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.

2.7.3 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

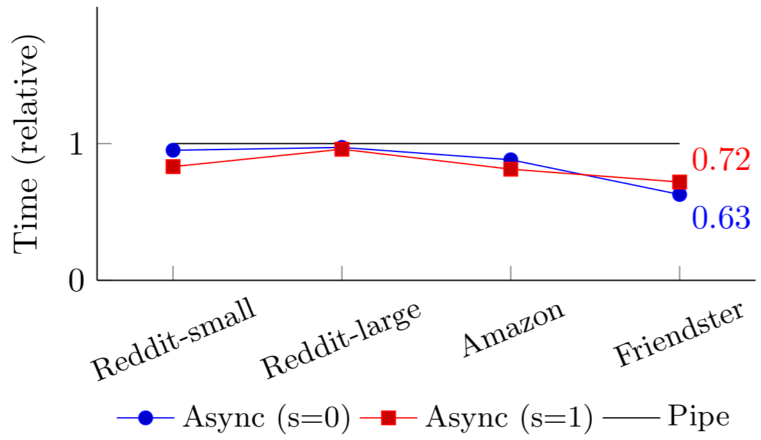


Figure 2.6: Per-epoch GCN time for async (s=0) and async (s=1) normalized to that of pipe.

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 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×1.08 epochs and async (s=1) run N×1.41 epochs when measuring performance for `Friendster` where N is the number of epochs pipe runs.

Figure 2.5 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 2.6 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 (∼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**× faster than pipe and **1.233**× than async (s=1). It also provides **1.288**× and **1.494**× 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.

2.7.4 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 concern, 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 2.4: 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 2.4). 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 × 2.67) compared to 1/(2092.7 × 3.01)). Figure 2.7 shows the value results for all our runs, *normalized to GPU-only servers.*

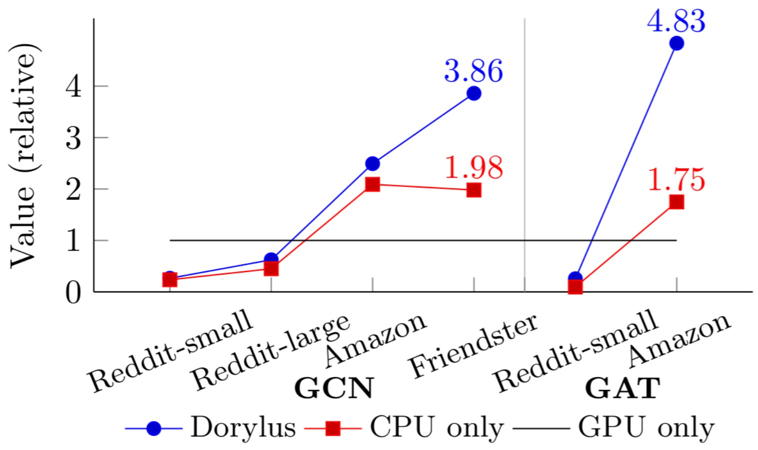


Figure 2.7: Dorylus, with Lambdas, provides up to 2.75× performance-per-dollar than using the CPU-only variant.

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 2.1. 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) 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 2.7). 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.

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

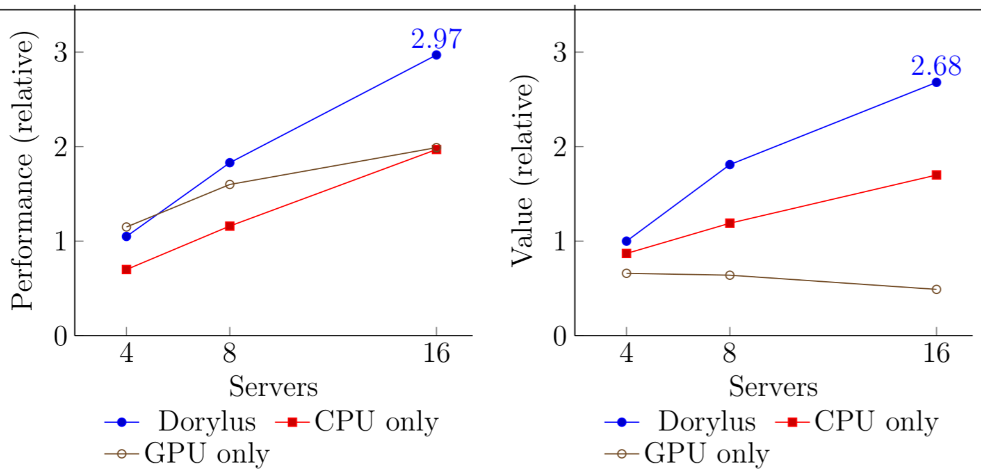


Figure 2.8: Normalized GCN training performance and value over `Amazon` with varying numbers of graph servers.

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 2.8 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 2.8(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 [85]. 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.

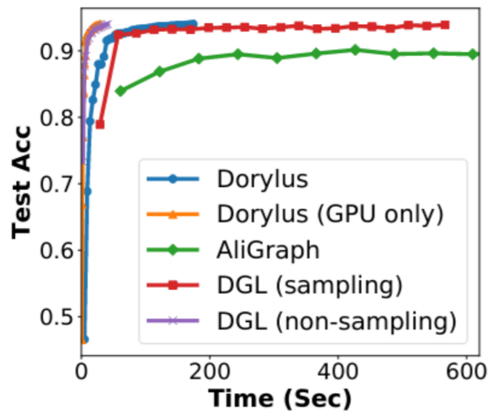
2.7.5 Comparisons with Existing Systems

Our goal was to compare Dorylus with all existing GNN tools. However, NeuGraph [85] and AGL [161] are not publicly available; neither did their authors respond to our requests. Roc [59] 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 [30], which is a popular GNN library with support for sampling, as well as AliGraph [157], 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.

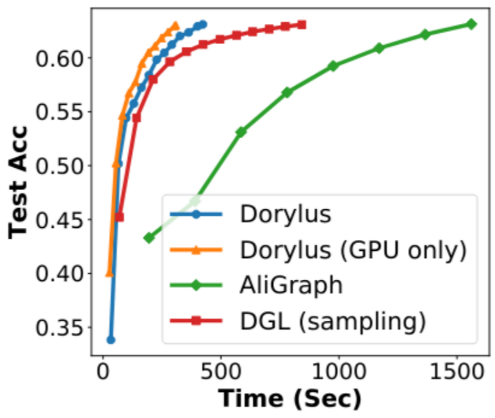
DGL represents an input graph as a (sparse) matrix; both graph and tensor computations are executed by PyTorch or MXNet as matrix multiplications. We experimented with two versions of DGL, one with sampling and one without. DGL-non-sampling does full-graph training on a single machine. DGL-sampling partitions 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.

Accuracy Comparison with Sampling. Figure 2.9 reports the accuracy-time curve for five configurations: Dorylus, Dorylus (GPU-only), DGL (sampling), DGL (non-sampling), 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.



(a) `Reddit-small`



(b) `Amazon`

Figure 2.9: 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.

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 2.5: 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.

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 2.9(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.

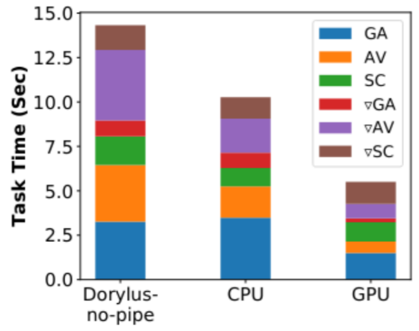
For the `Amazon` graph, DGL cannot scale without sampling. As shown in Figure 2.9(b), to reach the same target accuracy, Dorylus is 1.99× faster than DGL (sampling), and 1.37× slower than Dorylus (GPU only). AliGraph is able to reach the target accuracy for `Amazon`. However, Dorylus is significantly 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 2.5 are smaller than those reported in Table 2.4. This is due to the lower target accuracy we set for these experiments.

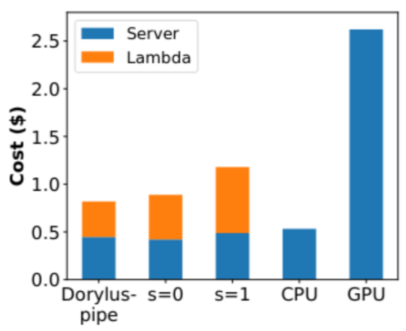
Value Comparison. To demonstrate the promise of Dorylus, we compared these systems using the value metric. As expected, given the small size of the `Reddit-small` graph, the GPU-based systems perform quite well. In fact, in this case the normalized value of DGL (non-sampling) is 1.48, providing a higher 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.

2.7.6 Breakdown of Performance and Costs

Figure 2.10 shows a breakdown in task time (a) and costs (b) for training a GCN over the `Amazon` graph. In Figure 2.10(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 naively to train a DNN. Without



(a) Task time breakdown



(b) Cost breakdown

Figure 2.10: Time and cost breakdown for the `Amazon` graph.

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 2.10(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.

2.8 Summary

Recall the central premise of this thesis: to reduce the cost of training while maintaining performance and accuracy by utilizing in-depth knowledge of ML workloads to choose the best cheap resources for the job while minimizing their drawbacks. With Dorylus, we took advantage of several characteristics of Graph Neural Networks, specifcally that they tend to be shallow and small and that they consist of interleaved graph and tensor workloads, to make effective use of serverless. We also closely analyzed the effects of asynchrony to develop our Bounded Pipeline Asynchrnous Computation while ensuring that it would not significantly degrade the accuracy of the model. We found that our combination of CPU servers with serverless threads was able to offer 2.75× more performance-per-dollar than CPU only servers, and 4.83× more than GPU only servers. Dorylus is also 3.8× faster and 10.7× cheaper than the existing sampling-based approaches. Based on the trends we observed, Dorylus can scale to even larger graphs than we evaluated, offering even higher values.

Dorylus's are well-suited to the properties of Graph Neural Networks and have increased the value to the user significantly for training. However, there are still other ML trends which need to be addressed and we need to carefully consider which resource would be an appropriate fit depending on the workload. Next, we look into the exponentially increasing size of traditional Neural Networks and try to understand how to continue this line of reasoning with new types of networks.

CHAPTER 3

Bamboo: Making Preemptible Instances Resilient for Affordable Training of Large DNNs

3.1 Introduction

DNNs are becoming progressively larger to deliver improved predictive performance across a variety of tasks, including computer vision and natural language processing. For instance, recent language models such as BERT [136] and GPT [108] already have a massive number of parameters, and their newer variants continue to grow at a rapid pace. For example, BERT-large has 340 million parameters, GPT-2 has 1.5 billion, and GPT-3 increases to 175 billion; the next generation of models embed upwards of trillions of parameters [35].

Of course, model growth also entails larger training costs. For instance, GPT-3 consumes several thousand petaflop/s-days, costing over \$12 million to train on a public cloud (needing hundreds of GPU servers) [17]. Unfortunately, such costs are prohibitive for small organizations. Even for large tech firms, training today's models incurs an exceedingly high monetary cost that eventually gets billed to the training department. While pretrained models may be reused and fine-tuned for different applications, training new models is often required to keep pace with changing or emerging workloads and datasets.

Although there exists a body of work on improving the training of large models [19, 24, 26, 28, 41, 54, 57, 61, 69, 94, 95, 117, 118, 133, 163], existing techniques focus primarily on *scalability* and *efficiency*, with monetary costs often being neglected. However, when *affordability* and *accessibility* are considered, resource usage becomes a key concern and none of these techniques were targeted at improving *cost-efficiency* (e.g., performance-per-dollar) for training.

Preemptible Instances. This paper explores the possibility of using preemptible instances—a popular class of cheap cloud resources—to reduce the cost of training large models. There are several kinds of preemptible instances. For example, major public clouds provide *spot instances* with a price much cheaper than *on-demand instances*—e.g., the hourly rate of a GPU-based spot instance is only ∼30% of that for its on-demand counterpart on Amazon EC2 [11]. As another example, large datacenters often maintain certain amounts of compute resources that can be allocated for any non-urgent tasks but will be preempted as urgent tasks arise [15, 97]. Similarly, recent ML systems [13, 58, 154] allow training jobs to use inference-dedicated machines to fully utilize GPU resources but preempts those machines when high-priority inference jobs arrive. The presentation of this paper focuses on spot instances, but we note that our techniques are generally applicable to any type of preemptible resources.

Despite their substantial cost benefits, preemptible instances pose major challenges in reliability and efficiency due the frequent and unpredictable nature of their preemptions. When and how many instances get preempted depends primarily on the number of priority jobs/users in a cluster. In a public spot market, preemption can also result from the market price exceeding the user's bid price. While price-based preemption can be avoided via a high bid price (e.g., the on-demand price), capacity-based preemption is unavoidable. Preemption patterns vary drastically across clouds and even across families/zones on the same cloud (§3.3).

Given the unpredictable nature of spot instances, users can often only run short, stateless jobs and simply restart these jobs if they get preempted. Model training, on the contrary, is stateful and time-consuming. Discarding the state (e.g., learned weights) upon each instance preemption not only wastes computation but also prevents training from making progress. Checkpointing-based techniques can reduce wasted computation to a degree, but still spend a significant fraction of the training time (e.g., **77%** when training GPT-2 with 64 EC2 spot instances, see §3.3) on restarting and redoing prior work in the presence of frequent preemptions [45, 46]—a largely different scenario compared to conventional clusters where failures are rare.

Bamboo. This paper presents Bamboo, a distributed system that provides resilience and efficiency for DNN training over preemptible instances. Bamboo supports both pipeline parallelism and (pure) data parallelism with the same approach. Since pipeline parallelism is a more complex and general approach (for training large models), our discussion focuses on pipeline parallelism; we briefly discuss our support for pure data parallelism in Section §3.8. Bamboo currently does *not* support model parallelism.

Redundant Computation. Key to the success of Bamboo is a set of novel techniques centered around *redundant computation* (RC), inspired by how disk redundancies such as RAID [103] provide resilience in the presence of disk failures. A training system that uses pipeline parallelism runs a set of data-parallel pipelines, each training on a partition of the dataset. Each node[1] in a data-parallel pipeline performs (forward and backward) computations over a shard of NN layers with a *microbatch* of data items [54]. Bamboo lets each node in each data-parallel pipeline carry its own shard of layers as well as its successor's shard. Each node performs *normal* computation over its own layers and *redundant* computation over its successor's layers. The reason why we use a neighbor node (as opposed to a random node) to run RC is to exploit data locality in pipeline parallelism (see §3.5). Upon a node preemption, its predecessor has all the information (e.g., layers, activations) needed for the training to progress; continuing training requires running a failover schedule on the predecessor node without wasting prior computations.

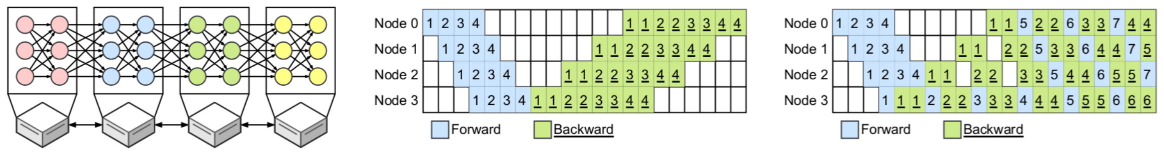
At first glance, running RC on every node appears infeasible due to concerns with both time and memory. Bamboo overcomes these challenges by taking into account pipeline characteristics to carefully reduce/hide these overheads.

[1]In this paper, "instance" and "node" both refer to a spot instance.

First, to minimize the time overhead from RC, Bamboo leverages a key insight that *bubbles* [54, 109] inherently exist in systems using synchronous pipeline parallelism (§3.2). Bubbles are idle times on each node due to the gaps between the forward and backward processing of microbatches (Figure 3.1). Bamboo schedules the forward redundant computation (FRC) on each node asynchronously into the bubble. For the part of FRC that cannot fit into the bubble, Bamboo overlaps it with the normal computation. As a result, FRC incurs a tolerable overhead (i.e., no extra communication is needed due to locality, and it can overlap with normal computation), and hence Bamboo performs it *eagerly* in each epoch. If a node is preempted during a forward pass, the pipeline continues after a node rerouting step whose overhead is negligible.

Unfortunately, for backward redundant computation (BRC), such a bubble does not exist. Eager BRC would require much extra work and data-dense communication on the critical path, which could delay training significantly (§3.5). As such, Bamboo runs BRC *lazily* only when a preemption actually occurs. If a node is preempted in a backward pass, continuing the pipeline requires a pause for the node's predecessor to perform BRC to restore the lost state. However, since FRC is performed eagerly, when BRC runs, much of what it needs is already in memory, keeping pauses short.

Second, performing RC increases each node's GPU memory usage. Note that the major source of the memory overhead is storing intermediate results (activations and optimizer state) from FRC, *not* the redundant layers, which take only little extra memory. To mitigate the memory issue, we leverage Bamboo's unique way of performing RC described above. Note that the purpose of saving intermediate results of a forward pass is that these results are used by its backward computation. However, in Bamboo, BRC is performed lazily upon preemptions and the intermediate results of FRC are thus not needed in normal backward passes. Hence, Bamboo swaps out the intermediate results of each node's FRC into the node's CPU memory, leading to substantial reduction in GPU memory usage. These results are swapped back into GPU memory for BRC only upon preemptions.



(a) Pipeline parallelism (b) GPipe scheduling (c) PipeDream scheduling

Figure 3.1: Illustration of pipeline parallelism on a 4-node cluster: (a) the model is divided into 4 shards, each with 2 layers; (b) and (c) show the scheduling of two recent systems GPipe [54] and PipeDream [94].

Bamboo continues normal training with the help of RC in the presence of non-consecutive preemptions, i.e., preempted instances are not neighbors in the same data-parallel pipeline. Once consecutive instances are preempted, RC can no longer provide resilience. More redundancies could be added to provide stronger resilience, but this would incur (compute and communication) overheads that are too significant to hide. Instead, based on our empirical observation that most concurrent preemptions come from the same allocation group (e.g., a zone), Bamboo takes care to ensure that consecutive nodes in each pipeline come from different zones, minimizing the chance of consecutive preemptions at a small (<5%) overhead (see §3.9.2).

Reconfiguration. In cases where consecutive preemptions do happen, we must reconfigure the pipelines (§3.7). Further, even if preemptions are non-consecutive, continuing training with RC after many preemptions is a "spare-tire" approach, which is vulnerable to future preemptions. To solve these problems, Bamboo provides a Kubernetes-based framework that monitors preemptions and reconfigures the pipelines by dynamically adding instances and adjusting pipeline configurations (e.g., the number of pipelines). Bamboo checkpoints the model state periodically. If no allocations can be made (i.e., a rare situation where the cluster is exhausted) and the remaining nodes are too few to sustain the training, Bamboo suspends the training until enough new instances can be obtained and the training can restart from the checkpoint.

Results. We built Bamboo atop DeepSpeed [109] and evaluated it by training 6 representative DNN models using EC2 spot clusters comprised of p3 instances. Compared to a baseline using on-demand instances, Bamboo delivers a **3.6**× cost reduction. Bamboo also outperforms a checkpointing approach by **3.7**×. We developed a simulation framework that takes preemption traces from real spot clusters and training parameters to simulate how training progresses with larger numbers of nodes. A deep-dive with BERT across a wide range of preemption probabilities shows that the *value* (i.e., performance-per-dollar) Bamboo provides stays constant and is much higher (**2.48**×) than that of on-demand instances. Bamboo will be open-sourced.

3.2 Background

This section discusses necessary background for parallelism strategies. *Data parallelism* keeps a replica of an entire DNN on each device, which processes a subset of training samples and iteratively synchronizes model parameters with other devices. Data parallelism is often combined with pipeline and/or model parallelism to train large models that do not fit on a single device. *Model parallelism* [29] partitions model operators across training devices. However, efficient model parallelism algorithms are extremely hard to design, requiring difficult choices among scaling capacity, flexibility, and training efficiency. As such, model-parallel algorithms are often architecture- and task-specific.

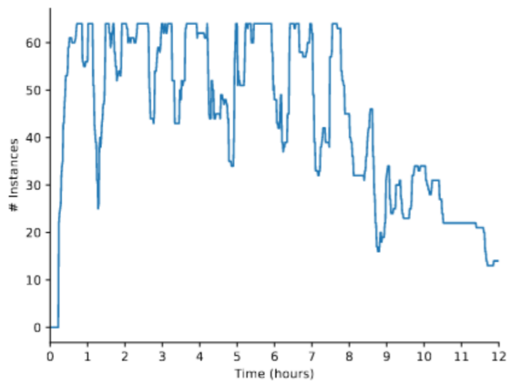
Pipeline parallelism [54, 94, 156] has gained much traction recently due to its flexibility and applicability to a variety of neural networks. Pipeline parallelism divides a model at the granularity of layers and assigns a shard of layers to each device. Figure 3.1(a) shows an example where the model is partitioned into four shards and each worker hosts one shard (with two layers). Each worker defines a computation stage and the number of stages is referred to as the *pipeline depth* (e.g., 4 in the example). One worker only communicates with nodes holding its previous stage or next stage. Each input batch is further divided into *microbatches*. In each iteration, each microbatch goes through all stages in a forward pass and then returns in an opposite direction in a backward pass. There are often multiple microbatches residing in the pipeline and different nodes can process different microbatches in parallel to improve utilization.

A key challenge in efficient pipeline parallelism is how to schedule microbatches. GPipe [54] schedules forward passes of all microbatches before any backward pass, as shown in Figure 3.1(b) where each node processes four microbatches. This approach leaves a ”bubble” (i.e., white cells) in the middle of the pipeline, leading to inefficient use of compute devices. PipeDream [94] proposes the one-forward-one-backward (1F1B) schedule to interleave the backward and forward passes, as shown in Figure 3.1(c). 1F1B can reduce the bubble size and the peak memory usage.

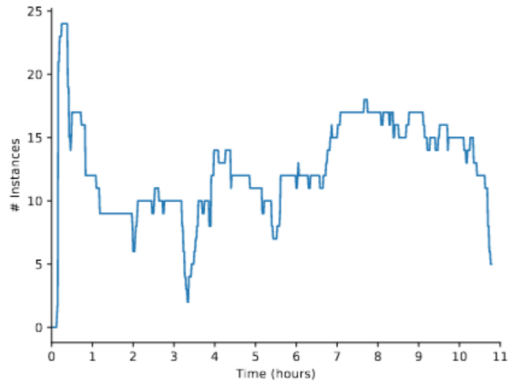
However, even with carefully-designed schedules, the pipeline bubble is still hard to eliminate. A fundamental reason is that it is extremely difficult to find the optimal layer partitioning to have each stage processed at the same rate. There exists a body of algorithms proposed recently to optimize layer partitioning and most of them are model- and hardware-specific [34, 94]. These algorithms are often time-consuming for large models, unsuitable for preemptible instances where the number of nodes keeps changing [10].

PipeDream [94] proposes asynchronous pipelining to eliminate the bubble—a node is allowed to work with stale weights to reduce the wait time. However, asynchronous microbatching introduces uncertainty in model convergence. In general, the effectiveness of synchronous v.s. asynchronous training is still open to debate. Furthermore, asynchronous training introduces inconsistencies in model state, which can create a more significant convergence issue when training occurs on preemptible instances, due to the need of frequent reconfigurations. For example, under synchronous microbatching, a reconfiguration can be performed at the end of each optimizer step (i.e., parameter update), and hence the reconfigured pipelines can start with the up-to-date parameters. This is impossible to do under asynchronous microbatching.

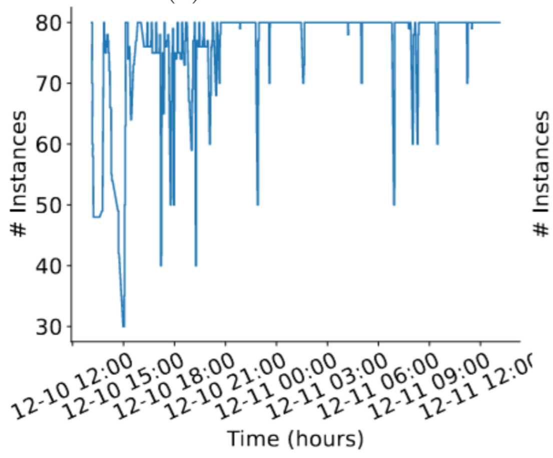
As a result, we built Bamboo atop synchronous microbatching where model state is always consistent. Instead of attempting to reduce the bubble, we explore an orthogonal direction—how to leverage the bubble to run RC efficiently.



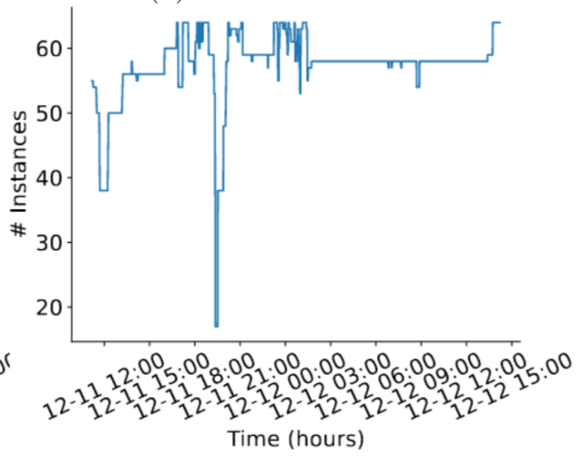
(a) P3 @ EC2



(b) G4dn @ EC2



(c) n1-standard-8 @ GCP



(d) a2-highgpu-1g @ GCP

Figure 3.2: Preemptions traces for a target cluster of size 64 instances on EC2 and 80 instances on GCP. Each graph shows a full-day trace for a GPU family in a cloud.

3.3 Motivation

This section motivates Bamboo from two aspects: (1) high preemption rates and unpredictability of spot instances, and (2) high performance overheads of strawman approaches.

Preemptions of Spot Instances. We first studied failure models with on major public clouds. Figure 3.2 shows a set of real preemption traces collected from running spot instances in two public clouds: Amazon EC2 and Google Cloud Platform (GCP). For EC2, we used two GPU families: P3 (NVIDIA V100 GPUs with 32GB of memory) and G4dn (NVIDIA T4 GPUs with 16GB of memory). For GCP, we used `n1-standard-8` (NVIDIA V100 GPUs with 16GB GRAM) and `a2-highgpu-1g` (NVIDIA A100 GPUs with 40GB GRAM). For each

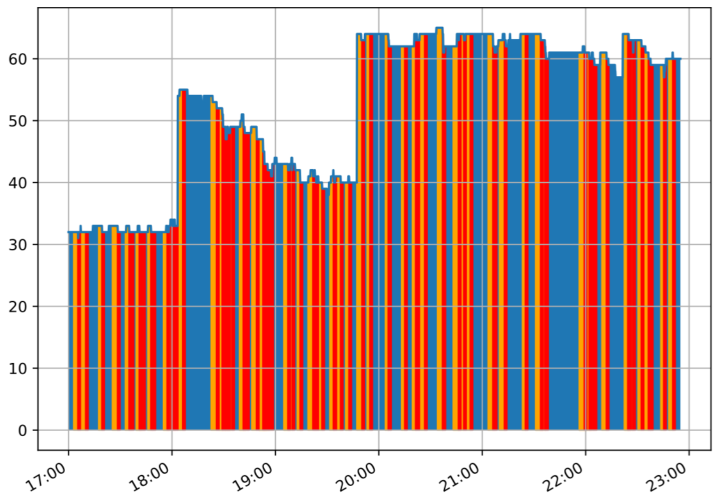


Figure 3.3: Training GPT-2 using checkpointing/restart with an autoscaling group of 64 P3 spot instances. Each color represents time spent in a distinct state,including Blue: training actively made progress; Orange: the cluster made progress that was then wasted; and Red: cluster restarting.

family, we collected traces for a 24-hour window. In each experiment, we used an autoscaling group to maintain a cluster of 64 with an exception of `us-east1-c` in GCP, whose cluster size is 80. The autoscaling group, provided by each cloud, automatically allocates new instances upon preemptions to maintain the size (though without any guarantee).

From both families, node preemptions and additions are frequent and bulky (i.e., many nodes get preempted at each time). This can make a checkpointing-based approach restart many times in a short window of time, leading to large inefficiencies (discussed shortly). Furthermore, both preemptions and allocations are unpredictable. While the autoscaling group attempts to allocate new nodes to maintain the user-specified size, allocations are committed incrementally; new allocations are mixed with preemptions of existing instances, making the spot cluster an extremely dynamic environment.

To understand the nature of the nodes that are preempted at the same time, we carefully analyzed two 24-hour preemption traces collected respectively from EC2 and GCP. For the EC2 trace, preemptions occur at 127 distinct timestamps, each of which see many preempted nodes. Of these 127 timestamps, only 7 see preemptions from multiple zones; at each of the remaining 120 timestamps, all nodes preempted come from the same zone. A similar observation was made on the GCP trace (12 out of 328 timestamps see cross-zone preemptions). These results confirmed the observations made by existing works [45, 46]: preemptions tend to be independent based on each individual spot market and each availability zone has a different and independent spot market—this is because each availability zone maintains capacity separately and therefore capacity preemptions in one zone are not associated with capacity preemptions in another.

These observations motivate our design—even with 1-node redundancies, Bamboo can recover from a majority of preemptions if consecutive nodes are not preempted at the same time; we maximize this possibility with a best-effort approach that makes consecutive nodes in each pipeline come from different zones. Although this may increase communication costs, it does not lead to visible performance impacts for Bamboo because Bamboo only sends (small amounts of) activations data between nodes.

Strawman #1: Checkpointing. We next show why a technique based on checkpointing and restarting does not work. We developed a new checkpointing system on top of DeepSpeed [109], providing checkpointing and restarting functionalities similar to TorchElastic [106]. We modified DeepSpeed to checkpoint *continuously* and *asynchronously*. In particular, each worker moves a copy of any relevant model state to CPU memory whenever the state is generated; the CPU then asynchronously writes it to remote storage so that training and checkpointing can fully overlap. During restarting, our system automatically adapts the prior checkpoints to the new pipeline configurations.

To understand how well this technique performs, we used it to train GPT-2 over 64 p3.2xlarge GPU spot instances on EC2. We profiled the training process and collected the checkpointing times, reconfiguration overheads, and total execution time. Figure 3.3 reports these results. The blue sections represent the times the system spent making actual progress for training. The red sections represent the times on reconfiguring (i.e., restarting) while the orange sections show the times for wasted work—the computation that was done but not saved in checkpoints; the system ended up redoing these computations after restarting. This is

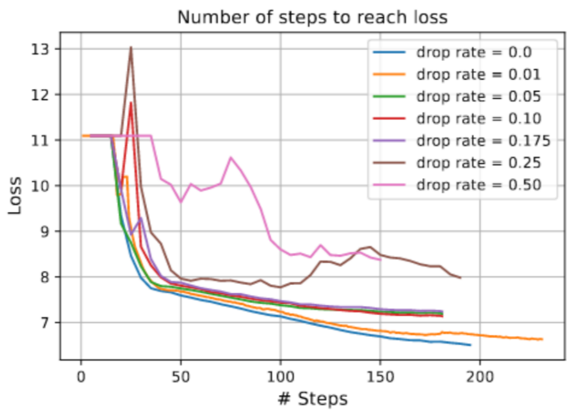


Figure 3.4: Effects of sample dropping under different rates.

because preemptions often occur during checkpointing, and hence, the system must roll back to a previous checkpoint. Frequent rollbacks slows down the training significantly. As shown, although checkpointing itself can be done efficiently, the restarting overheads (i.e., for adapting existing checkpoints to new pipeline configurations) and the wasted computations take **77%** of the training time.

Strawman #2: Sample Dropping. An alternative approach that has shown promise is to take advantage of the statistical robustness of DNN training and allow some samples to be dropped so that training can continue without significant loss of accuracy [81, 149]. These techniques are also known as *elastic batching* because dropping samples is equivalent to changing the effective batch size at a training iteration (with the learning rate dynamically adjusted).

In the case of pipeline parallelism, we implemented sample dropping by suspending a pipeline upon losing an instance while letting other data-parallel pipelines continue to run. The system performs optimizer steps with the gradients of whichever data-parallel pipelines are able to complete that training step. Learning rate was adapted linearly with respect to the effective batch size to make sure that the only effect on the accuracy is the lost samples, but *not* a mismatch between hyperparameters and training configurations. In doing so, the training can continue for sometime without a reconfiguration (which is needed upon allocations).

We conducted a set of experiments to simulate the effect of sample dropping on model accuracy with a range of drop rates. Note that we could not obtain these results with the actual spot instances because we could not control the preemption rate. We ran a pre-training benchmark with GPT-2 using 16 on-demand instances from the same EC2 family, which form four data-parallel pipelines, each with four stages. To consider a range of different failure models, we used different rates of preemption to generate preemption events. Upon a preemption event, we randomly selected a pipeline and zero out the pipeline's gradients in that iteration. We measured the model's evaluation accuracy every 5 training steps. These results are shown in 3.4 where each curve represents the function of the number of steps needed to reach a given loss for a particular drop rate.

Similarly to checkpointing, sample dropping works well for low preemption rates, but when frequent preemptions occur, many samples can be lost quickly and its impact on model accuracy quickly grows to be too significant to overlook. While this experiment was not an exact recreation of a sample dropping scenario, these results represent an *under-approximation* of the effect of the actual sample dropping (which can lose more accuracy than reported by Figure 3.4). This is because the actual sample dropping rate should be higher than the instance preemption rate—a preempted instance would likely be down for some time and consecutive samples would be dropped in a real setting. Note that training samples are shuffled before loading; hence, the effects of randomly dropping consecutive samples (i.e., the actual scenario) and dropping random samples sporadically (i.e., our experiment) should be similar.

Strawman #3: Live Migration in Grace Periods. Another potential approach is to use the grace period before each preemption to migrate data from the preempted node to a live node. However, this approach suffers from significant drawbacks. First, grace periods vary from cloud to cloud. While AWS spot instances have 2 minutes before preemption, GCP and Azure provide only 30 seconds, with GCP not even guaranteeing such a warning. For large models, this can be too short of a warning and may not leave sufficient time to save model updates into a checkpoint.

A more important issue is that this approach depends on always maintaining enough idle nodes as migration targets. There is no way to guarantee that, unfortunately, with the current spot market. Even if we could over-provision and reserve a certain number of standby nodes, these nodes can also be preempted and it is impossible to ensure when a set of nodes in the pipeline are preempted, there are enough standby nodes for data migration. In fact, during our experiments, for each preemption event, the number of new allocations we could obtain was always less than the number of preempted nodes (as shown in Figure 3.2).

3.4 Overview

Goal and Non-Goal. Our goal is *not* to automatically determine the cheapest way to train a given model (e.g., which parallelism model can lead to the largest cost savings). Instead, Bamboo aims to enable efficient and preemption-safe training over cheap spot instances.

User Interface. To use Bamboo, a user specifies two system parameters D and P, as they normal would to use other pipeline-parallel systems, where D is the number of data-parallel pipelines and P is the pipeline depth. Due to the need of storing redundant layers, Bamboo requires a larger pipeline depth P than a normal pipeline-parallel system such as PipeDream [94]. We observed, empirically, that to avoid swapping data between CPU and GPU memory on the critical path, Bamboo's pipeline should be $\sim$1.5× (see §3.6.4) longer than an on-demand pipeline due to the extra memory needed to (1) hold the redundant layers and (2) accommodate potential pipeline adjustments. Given that spot instances are much cheaper (e.g., 3-4× on EC2) than on-demand instances, training with 1.5× more nodes still leads to significantly reduced costs. While we recommend 1.5× more nodes, the number of active instances in a cluster is often much smaller due to preemptions and incremental allocations.

$P \times D$ will be the size of the spot cluster Bamboo attempts to maintain throughout training. Preemptions can cause Bamboo to reduce the pipeline depth and/or the number of pipelines; in such cases, Bamboo would request more instances to bring the size of the cluster back to $P \times D$. However, Bamboo would never try to scale the training beyond $P \times D$. In other

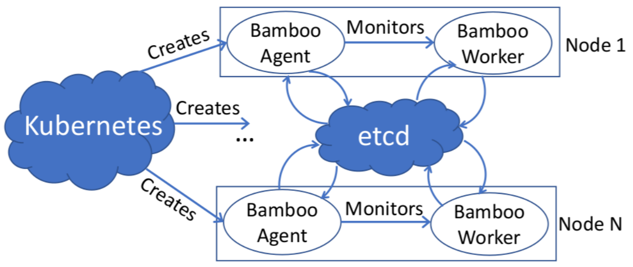


Figure 3.5: Bamboo runs one agent process per node (i.e., spot instance). An agent monitors worker processes (each running a training script) that use our modified DeepSpeed. All workers and agents coordinate through `etcd` [5].

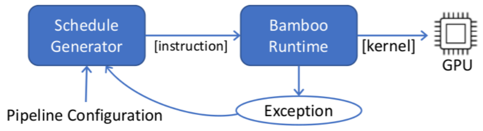


Figure 3.6: Bamboo worker.

words, P and D are the *upper bound* of the pipeline depth and number of pipelines. It is important to note that the goal of the autoscaling framework we build for Bamboo is to adjust the pipelines *passively* in response to node preemptions and additions that we cannot control, rather than *proactively* finding an optimal cluster configuration to achieve better performance. This distinguishes Bamboo from existing works on autoscaling distributed training [10, 56, 101], whose goal was to find better configurations.

System Overview. Figure 3.5 shows an overview of our system. We built Bamboo on TorchElastic [106] and DeepSpeed [109]. In particular, we built the Bamboo agent, which runs on each node to kill/add a data-parallel pipeline, on top of TorchElastic. The agent monitors a Bamboo worker process on the same node, which is a DeepSpeed application enhanced with our support for redundant computation. Bamboo workers run D data-parallel pipelines that use an `all-reduce` phase to synchronize weights at the end of each iteration. Our spot instances are managed by Kubernetes [4], which is configured to automatically scale by launching a Bamboo agent on each new allocation. Our agents communicate and store cluster state on `etcd` [5], a distributed key-value store.

Each Bamboo worker uses a runtime to interpret the schedule, which produces a sequence of instructions, as shown in Figure 3.6. The schedule is generated statically based on the stage ID of the current worker and pipeline configurations, including the depth of pipeline and total number of microbatches. The instructions consist of a computation component (i.e., forward, backward, and apply gradient), and a communication component (i.e., send/receive activation, send/receive gradient, and all-reduce). The Bamboo runtime interprets these instructions by launching their corresponding kernels on GPU. Communication instructions can fail due to preemptions. Upon a failure, the runtime throws an exception and falls back to use a failover schedule.

3.5 Redundant Computation

For ease of presentation, our discussion focuses on one node running one stage in the pipeline. Support for multi-GPU nodes will be discussed shortly.

Preemption of a node is detected by its neighboring nodes in the same pipeline during the execution of communication instructions. If a node on one side of the communication is preempted, the node on another side will catch an IO exception due to broken socket and update cluster state on `etcd`. Bamboo detects preemptions based on socket timeout. Although we could let a node to be preempted actively notify its neighbors in the grace period before the preemption, the length of this period varies across different clouds and hence Bamboo does not use it currently.

Since the victim node communicates with two nodes in the pipeline, both of its neighbors can catch the exception. The observed exception will be shared between these two nodes through `etcd`. This two-side detection is necessary for Bamboo to understand which node fails and generate the failover schedule. In addition to the two neighbors, nodes in other pipelines involved in the `all-reduce` operation also need to be informed. To safely perform `all-reduce`, each node participating in `all-reduce` reads the up-to-date cluster state on `etcd` and, if another pipeline has a failure, waits until the failure is handled.

3.5.1 Redundant Layers and Computation

To quickly recover from preemptions, Bamboo replicates the model partition on each worker node in each data-parallel pipeline. Instead of saving these replicas to a centralized remote storage (like checkpointing), Bamboo takes a *decentralized* approach by letting each node replicate its own model partition (i.e., layer shard) on its predecessor node in the same pipeline. The first node has its layer replica stored on the last node in the pipeline. Conceptually, the last node is considered the "predecessor" of the first node. For simplicity of presentation, we use *forward stage IDs* to identify nodes, that is, a node that runs the forward stage $n+1$ is always considered as a successor of a node running the forward stage n (although in the backward pass, $n+1$ is a stage before n).

Our key idea is to let each node run normal (forward and backward) computation over its own layers and redundant (forward and backward) computation over the replica layers for its successor node. Let $\mathrm{FRC}_n^m/\mathrm{BRC}_n^m$ denote the forward/backward redundant computation that is performed on node m for node n, respectively. In Bamboo, $n = (m+1)\ mod\ P$ where P is the pipeline depth. Let $\mathrm{FNC}_n/\mathrm{BNC}_n$ denote the forward/backward normal computation on node n. In Bamboo's pipeline, $\mathrm{FRC}_{n+1}^n/\mathrm{BRC}_{n+1}^n$ is exactly the same computation as $\mathrm{FNC}_{n+1}/\mathrm{BNC}_{n+1}$, working with the same model parameters and optimizer states. To enable the last node to perform RC for the first node, we let it fetch input samples directly.

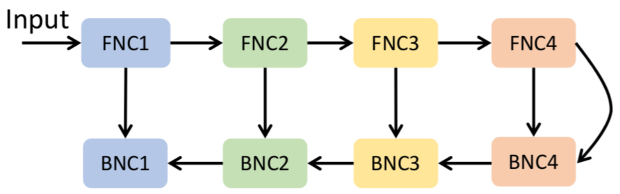


Figure 3.7: Dependencies between normal pipeline stages.

Why Neighboring Nodes? Due to our focus on pipeline parallelism, Bamboo performs RC on predecessor nodes to exploit *locality* for increased efficiency. To see this, we first need to understand the dependencies between different (backward and forward) pipeline stages that a microbatch goes through, as illustrated in Figure 3.7. For each forward stage FNC_n, it depends only on the output of its previous stage FNC_{n-1}. However, for each backward stage

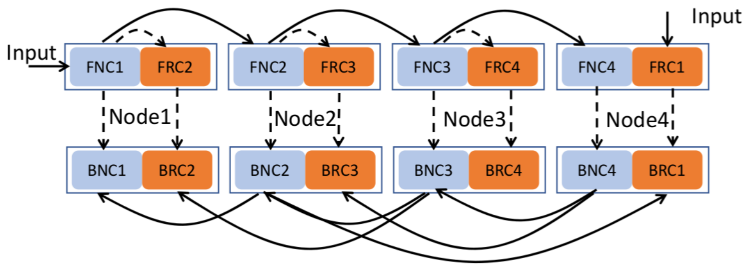


Figure 3.8: Dependencies between RC-enabled pipeline stages: solid/dashed arrows represent inter/intra-node dependencies; for simplicity, $\text{FRC}_n/\text{BRC}_n$ in the figure represents $\text{FRC}_n^{n-1}/\text{BRC}_n^{n-1}$.

BNC_n, it has two dependencies: one on the output of stage BNC_{n+1} and a second on its corresponding forward stage FNC_n. The first is a *hard* dependency without which BNC_n cannot be done, while the second is a *soft* dependency primarily for efficiency—intermediate results produced by FNC_n can be reused to accelerate BNC_n. Without such cached results, BNC_n has to recompute many tensors (i.e., tensor rematerialization [23]), leading to inefficiencies.

Figure 3.8 shows dependencies on an RC-enable pipeline where each node performs both normal and redundant (backward and forward) computation. Here solid/dashed arrows represent inter/intra-node dependencies. By running FRC for node $n+1$ on node n, *locality benefit* can be clearly seen because FRC only creates intra-node dependencies, which do not incur any extra communication overhead. However, in a backward pass, such a locality benefit does not exist for BRC_{n+1}^n, which requires the output of BNC_{n+2} and incurs much extra communication. This motivates our eager-FRC-lazy-BRC design which does not perform BRC until a preemption occurs and hence eliminates the extra communication cost in normal executions.

Note that we could also perform FRC lazily, but this would significantly increase the pause time for recovery. This is because (1) recovering from preemptions at both forward and backward pass now require a pause; and (2) lazy FRC would not produce intermediate results that can be used to speed up BRC and hence BRC's pause would be much longer. Since FRC can be scheduled in the pipeline bubble and overlap with FNC, performing it eagerly is a better choice.

The careful reader may think of an alternative approach that places node n's layer replica on node $n+1$ as opposed to node $n-1$ (i.e., its successor rather than its predecessor). This approach is symmetric to our design in that it turns inter-node dependencies for BRC into intra-node dependencies, but intra-node dependencies for FRC into inter-node dependencies. As a result, it eliminates the extra backward communication at the cost of increased forward communication. However, unlike Bamboo's design that can use lazy BRC to eliminate the extra backward communication, it is not as easy to eliminate the extra forward communication with lazy FRC—if FRC is not done eagerly in each iteration, BRC (regardless of whether it is eager or lazy) must perform tensor re-materialization, which incurs a long delay.

Level of Redundancy. As with any redundancy-based systems, the more redundancies, the higher level of resilience. For example, since Bamboo performs redundant computations only for one node, it cannot provide resilience when preemptions occur on consecutive nodes in a pipeline, in which case a reconfiguration is needed (see §3.7). However, enabling RC for multiple nodes can significantly increase the FRC time, making it much longer than what the bubbles can accommodate. Furthermore, the locality benefit (i.e., FRC only incurs intra-node dependency) does not hold anymore, because FRC now depends on the outputs of multiple nodes. This can slow down the training substantially.

Takeaway. Storing each node's replica layers on its predecessor and running eager-FRC-lazy-BRC achieves low-overhead RC for pipeline parallelism. While this design does not support consecutive preemptions, Bamboo takes care to make consecutive nodes come from different zones. As discussed in §3.3, if multiple preemptions occur at the same time, the preempted nodes are highly likely to be from the same zone. As a result, our node assignment reduces the chance of consecutive preemptions, making RC effective for most preemptions. Although cross-zone data transfer can incur an overhead, this overhead is negligible (e.g., <3%), as reported in Section §3.9.2, because in pipeline-parallel training, each node only passes a small amount of activation data to its neighbors.

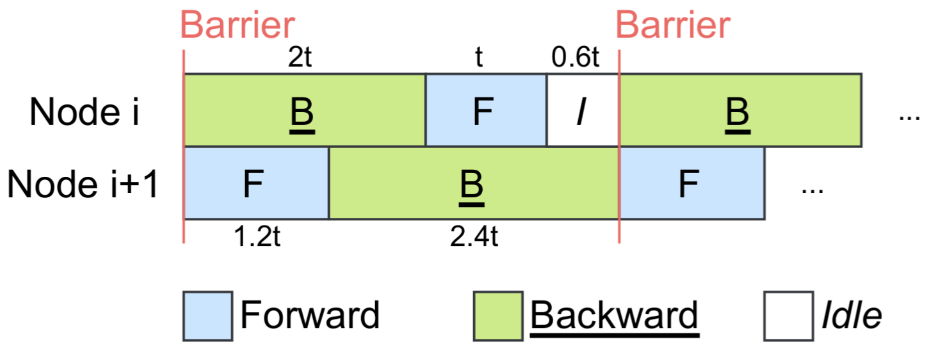


Figure 3.9: A closer examination of the pipeline bubble. Here we assume the forward pass on node i and $i+1$ takes time t and $1.2t$, respectively. Hence, a bubble of $0.6t$ exists before each communication barrier.

We refer to the preempted node as the *victim node*, and the node saving the replica of the victim as its *shadow node*.

3.5.2 Schedule Redundant Computation

It is straightforward to see that RC incurs an overhead in both time and memory. We propose to (1) schedule FRC into the pipeline bubble to reduce forward computation overhead, (2) perform BRC lazily to reduce backward computation/communication overhead, and (3) offload unnecessary tensors to CPU memory to reduce memory overhead.

Eager FRC. As discussed in §3.2, the pipeline bubble can come from either imperfect scheduling or unbalanced pipeline partitioning. To illustrate, consider Figure 3.9 with PipeDream's 1F1B schedule. Suppose there are two consecutive nodes in the pipeline where both the forward and the backward computation of node $i+1$ run 1.2× slower than those of node i. The communication between these two nodes serves as a barrier. Since node i runs faster, it always reaches the barrier earlier and waits there until node $i+1$ arrives. This wait period is where we should schedule FRC.

Bamboo builds on the 1F1B schedule (Figure 3.1(a)) due to its additional efficiency compared to GPipe's schedule (Figure 3.1(b)). However, even for 1F1B, bubbles widely exist in a pipeline—as a microbatch passes different pipeline stages, the later a stage, the longer the (backward and forward) computation takes. This is because for the 1F1B schedule, the number of active microbatches in a later stage is always smaller than that in an earlier stage. In Figure 3.1 (c), for example, node 1 has 3 active microbatches while node 2 only has 2. Consequently, later stages often consume less memory. To balance memory usage, the layer partition on a later node is often larger that that on an earlier node in the pipeline, and hence a later stage runs slower.

Based on this observation, we schedule FRC on a node before the node starts communicating with its successor node. This is where a bubble exists. The bubble size is often large enough to run FRC for most microbatches (shown in the §3.9.1). In cases where the FRC cannot fit entirely into the bubble, we overlap FRC and FNC as much as we can. However, for the same microbatch, FRC^{n}_{n+1} depends on FNC_n and they cannot run in parallel. To resolve this dependency issue, we focus on different microbatches for FNC and FRC. That is, Bamboo schedules FNC_n for the k-th microbatch and FRC^{n}_{n+1} for its previous $(k-1)$-th microbatch to run in parallel. Since there is no dependency between them, their executions can overlap.

To reduce memory overhead, Bamboo follows a well-known principle to offload less frequently used tensors to CPU memory. Specially, since BRC is *not* performed in normal training passes, FRC's outputs and intermediate results are not needed until a preemption occurs and BRC is triggered. As a result, we swap out these data after FRC is done for each microbatch on each node. These data take the majority of FRC's memory consumption; swapping them out significantly reduces FRC's GPU memory usage [110]. However, we leave the redundant weights in GPU memory for efficient FRC because these weights are needed for FRC on each microbatch.

Lazy BRC and Recovery. BRC is executed by a *failover schedule* which a node runs when detecting its successor node fails. In particular, for the current iteration, all the lost gradients must be re-computed, while for the following iterations, all instructions of the victim node must be executed by its shadow node (until a reconfiguration occurs). Nodes that originally communicate with the victim node are transparently rerouted to the shadow node. The failover schedule is generated by merging the schedules of the victim and shadow

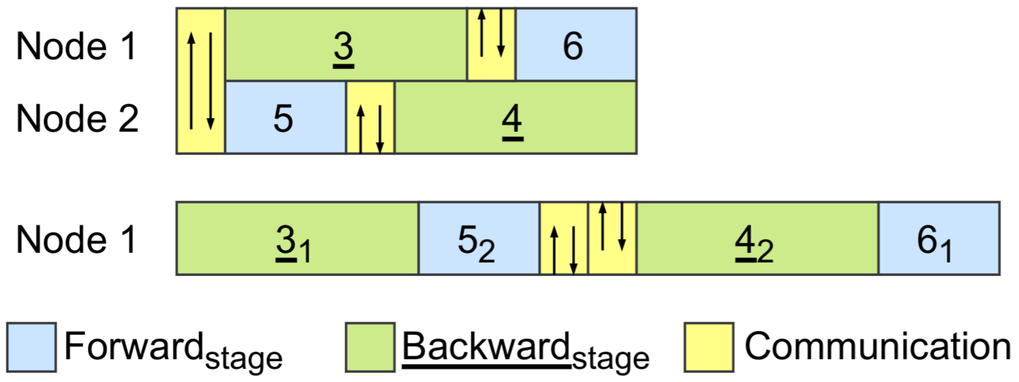


Figure 3.10: A example of merged instruction sequences in failover schedule. We use PipeDream's 1F1B schedule as shown Figure 3.1(c), and assume node 2 is the victim node and node 1 is the shadow node.

node. In particular, a schedule consists of a sequence of instructions and we divide it into two groups—(1) continuous communication instructions, which is placed at the head of a group and (2) computation instructions that can be executed without remote data dependencies.

When the two instruction groups (from the victim and shadow nodes) are merged, the instructions are interleaved with the following rules. (1) Communication instructions are still placed in the beginning of the merged groups. (2) Communications that used to be inter-node between the victim and the shadow are removed. (3) External communications from the victim node are first performed. (4) Computation instructions are ordered such that backward computation is always executed earlier; after the backward computation is done, the memory occupied by intermediate results is freed. Figure 3.10 shows an example of merged instruction sequences if node 2 is the victim node and node 1 is the shadow node.

Support for Multi-GPU Nodes. Bamboo's RC works for multi-GPU settings—this requires replicating all layers that belong to the GPUs of one node in the GPUs of its predecessor node. In other words, we use "group replicas" as opposed to individual replicas. However, in the presence of frequent preemptions, using multi-GPU would yield poorer performance—losing one node (with multiple GPUs) is equivalent to losing multiple nodes in the single-GPU setting. Our evaluation (§3.6) shows that it is much harder to allocate new multi-GPU nodes during training than single-GPU nodes.

Model	**Dataset**	D	P
ResNet-152 [49]	ImageNet [70]	4	8×1.5 (12)
VGG-19 [131]	ImageNet [70]	4	4×1.5 (6)
AlexNet [70]	Synthetic data	4	4×1.5 (6)
GNMT-16 [152]	WMT16 EN-De	4	4×1.5 (6)
BERT-Large [32]	Wikicorpus En [32]	4	8×1.5 (12)
GPT-2 [108]	Wikicorpus En [32]	4	8×1.5 (12)

Table 3.1: Our models, datasets, pipeline configurations.

Once Bamboo loses too many nodes or there are many idle nodes (i.e., new allocations) waiting to join the pipelines, Bamboo launches a reconfiguration. Details of the reconfiguration process can be found in Section §3.7.

3.6 Evaluation

Bamboo is implemented in ∼7K LoC as a standard Python library. We evaluated Bamboo by pretraining a range of popular vision and language models, as shown in Table 3.1. We used two tasks and four datasets in our experiments: (1) image classification, using the ImageNet-1K (ILSVRC12) [114] dataset and (2) translation, using the WMT16 English to German dataset for GNMT-16 and the Wikicorpus dataset [32] for BERT and GPT-2. For the first four (smaller) models that were also used in PipeDream [94] (which actually used smaller versions of these models), we took the values of D (the number of data-parallel pipelines) and P_{demand} (pipeline depth) from PipeDream [94]'s configurations.

As discussed earlier in §3.4, to avoid swapping Bamboo needs 1.5× more instances for each pipeline and hence each P reported in Table 3.1 equals $1.5\times P_{demand}$. For BERT and GPT2, we used 4 and 8×1.5=12 as D and P. We have also evaluated with another pipeline depth $P_h = P_{demand} \times \frac{Price_{demand}}{Price_{spot}}$; these results can be found in §3.6.2.

We trained these models on a spot cluster from EC2's p3 family where each instance has V100 GPU(s) with 16GB GPU memory and 61GB CPU memory. Each on-demand instance costs \$3.06/hr per GPU while the price of its spot counter-part (at the time of our experiments) is \$0.918/hr. Our evaluation uses two on-demand baselines: (1) p3 instances each with four V100 GPUs (Demand-M) and (2) p3 instances each with a single GPU (Demand-S). For both baselines, the pipeline configuration was the same and all nodes were obtained from one availability zone.

For all experiments, we trained each model to a target validation accuracy, which is a particular number of samples for the model. We did not train them to higher accuracies because large models take a huge amount of time to train (e.g., weeks) to reasonable accuracies; using such a large amount of resources (even spot instances) goes beyond our financial capabilities. Furthermore, Bamboo uses synchronous training where the time per iteration is fixed; hence, training for extended time would not change our results.

For on-demand instances, we used the largest per-GPU minibatch that fits in one GPU's memory—anything larger yields out-of-memory exceptions. This ensures that we hit peak achievable FLOPs on a single device. For data-parallel runs with n workers, the global minibatch size is $n \times g$ where g is the minibatch size. The global minibatch sizes we used are consistent with those used by the ML community and reported in the literature for these models. We used a per-GPU minibatch size of 256 per GPU for VGG-19, 512 for AlexNet, 2048 for ResNet-152, 32 for GNMT-16, 256 for BERT-Large, and 256 for GPT-2. For micro-batch size, we always selected a small value and tuned it for different models/configurations. We trained the vision models with an initial learning rate of 0.001, respectively, with a vanilla SGD optimizer [63]. For language models, we used the Adam optimizer [64] with an initial learning rate of $6e^{-3}$. We used half (fp16) precision in all our experiments.

3.6.1 Training Throughput and Costs

Overall Performance. To thoroughly and deterministically evaluate Bamboo's performance over spot instances under different preemption rates, we first ran a 48-node cluster (i.e., the configuration for ResNet, BERT, and GPT) and a 32-node cluster (i.e., for VGG, AlexNet, and GNMT) on AWS and collected a 24-hour preemption trace for each. On these traces, the *hourly preemption rate* varies significantly, ranging from no preemption all the way to 16 nodes preempted (33%), with an average rate of 4-6 nodes per hour (8-12%). To account for such changes, we extracted from each trace three segments, each with a different hourly preemption rate: 10%, 16%, and 33%. We used AWS' fleet manager to trigger

Model	System	Throughput	Cost (\$/hr)	Value
ResNet	Demand-M	30	97.92	0.31
	Demand-S	32	97.92	0.33
	Bamboo-M	[19.35, 15.69, 8.22]	[44.33, 40.01, 37.21]	[0.43, 0.39, 0.22]
	Bamboo-S	[**21.67**, 19.41, 12.13]	[**42.23**, 40.39, 36.72]	[**0.51**, 0.48, 0.33]
VGG	Demand-M	197	48.96	4.02
	Demand-S	167	48.96	3.41
	Bamboo-M	[93.34, 75.75, 64.22]	[21.31, 19.55, 18.43]	[4.38, 4.11, 3.48]
	Bamboo-S	[**153.31**, 124.88, 98.21]	[**20.19**, 19.28, 18.36]	[**7.59**, 6.48, 5.35]
AlexNet	Demand-M	359	48.96	7.33
	Demand-S	336	48.96	6.86
	Bamboo-M	[271.06, 207.43, 143.57]	[21.31, 19.55, 18.43]	[12.72, 10.61, 7.79]
	Bamboo-S	[**340.32**, 321.65, 280.42]	[**20.19**, 19.28, 18.36]	[**16.86**, 16.68, 15.27]
GNMT	Demand-M	27	48.96	0.55
	Demand-S	24	48.96	0.49
	Bamboo-M	[13.95, 10.82, 6.33]	[21.31, 19.55, 18.43]	[0.65, 0.55, 0.34]
	Bamboo-S	[**18.92**, 16.31, 8.8]	[**20.19**, 19.28, 18.36]	[**0.94**, 0.85, 0.48]
BERT	Demand-M	118	97.92	1.21
	Demand-S	108	97.92	1.10
	Bamboo-M	[71.22, 56.41, 41.68]	[44.33, 40.01, 37.21]	[1.61, 1.41, 1.12]
	Bamboo-S	[**98.87**, 83.70, 60.59]	[**42.23**, 40.39, 36.72]	[**2.34**, 2.07, 1.65]
GPT	Demand-M	32	97.92	0.32
	Demand-S	30	97.92	0.30
	Bamboo-M	[17.73, 14.00, 11.54]	[44.33, 40.01, 37.21]	[0.40, 0.35, 0.31]
	Bamboo-S	[**29.92**, 22.68, 13.78]	[**42.23**, 40.39, 36.72]	[**0.71**, 0.56, 0.38]

Table 3.2: Comparisons between training with DeepSpeed over on-demand instances and Bamboo over spot instances; throughput is defined as the number of samples per second. For Bamboo, we train each model three times, and their results are explicitly listed in the form of $[a, b, c]$ for the 10% (average), 16%, and 33% preemption rates, respectively.

preemptions by replaying these segments. Note that if we were to run Bamboo over the uncontrolled spot cluster, there would be no way to enable a direct comparison.

We trained ResNet, BERT, and GPT by replaying the three segments from the 48-node trace, and VGG, AlexNet, and GNMT by using the segements from the 32-node trace. These results are reported in Table 3.2. In addition to the time and monetary costs, we used a metric called *value*, which measures performance-per-dollar. Value is computed as $V = \frac{T}{C}$ where T is the training throughput, measured in terms of the number of samples per second, and C is the monetary cost per hour. Throughout the evaluation, we used both value and throughput as our metrics.

Our first observation is Demand-M slightly outperforms Demand-S due to reduced cross-node communication. However, the difference is marginal as the amount of data (i.e., only activations) transferred over the network is small. Bamboo-S significantly outperforms Bamboo-M (i.e., **1.4**× higher throughput and **1.5**× higher value) because (1) multi-GPU nodes are subject to more GPU failures with the same number of preemptions and (2) it is much harder to to allocate new nodes in a timely fashion.

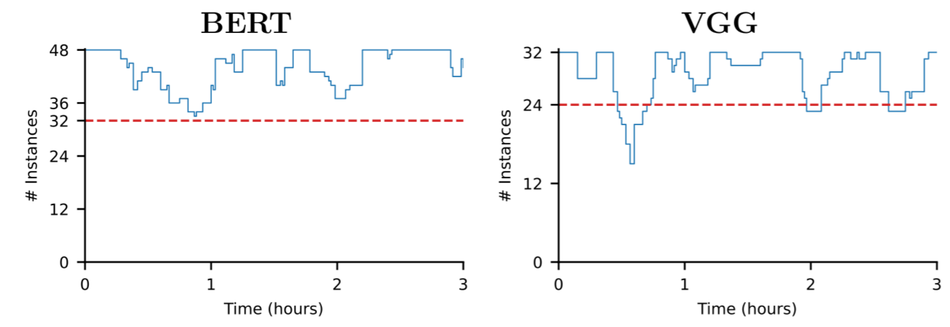
For Bamboo-S, the results in each bracket of the form [a, b, c] show Bamboo's performance under the three preemption rates. The higher the preemption rate, the worse Bamboo's throughput and value. Given that the average preemption rate is ∼10%, the first number in each bracket (highlighted) represents Bamboo's performance on the used spot cluster. On average, Bamboo's throughput (under the 10% preemption rate) is 15% lower than DeepSpeed running over $D \times P_{demand}$ instances. There are three major reasons.

First, the number of active instances in the spot cluster is actually lower than the requested size $D \times P$. For ResNet, for example, the average number of instances throughout the training is only 25.58 although the requested cluster size is 48 (and the on-demand cluster always has 32 nodes). The autoscaling group keeps attempting to add new instances but the total number of active instances only reaches the requested size for a small period of time.

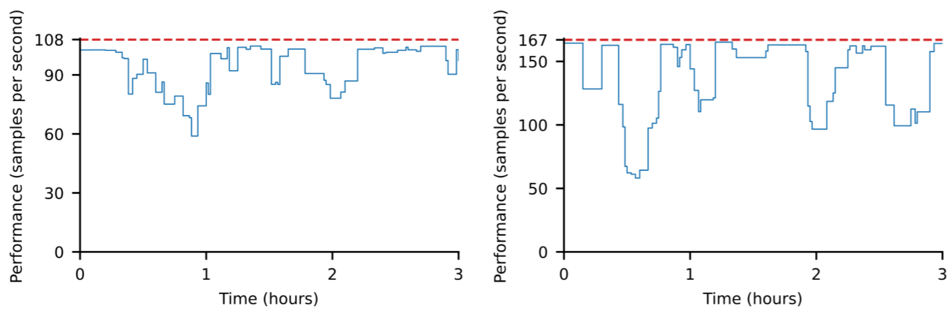
Second, Bamboo's reconfiguration contributes to reduced throughput—these overheads vary with environments and take an average of 7% of the total training time.

Third, the time for each iteration increases due to eager FRC. This is the major source of overhead for language models such as GPT-2. A detailed evaluation of RC's overhead can be found in §3.6.4.

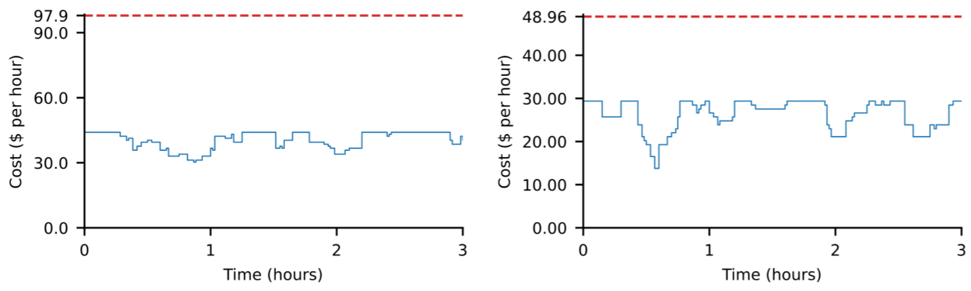
Despite the small throughput reduction, Bamboo delivers an overall of **1.95**× higher value compared to training with on-demand instances. The benefit in value remains clear for five models (ResNet, VGG, AlexNet, BERT and GPUT) even when the preemption rate increases to 33% (i.e., the worst-case segment of the collected trace).



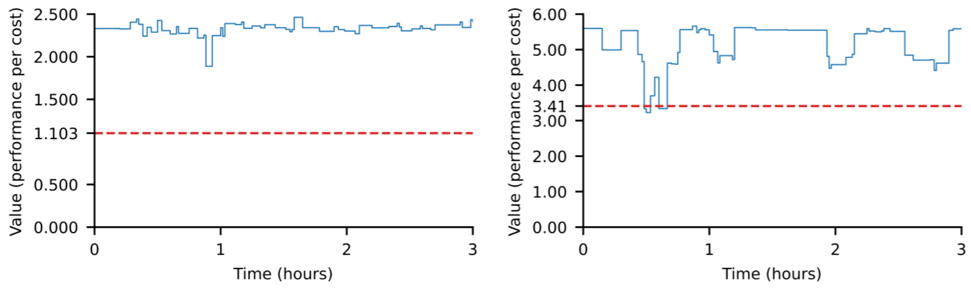
(a) Trace



(b) Training Throughput



(c) Monetary Cost



(d) Value

Figure 3.11: Bamboo's training performance for BERT (left) and VGG (right), compared to on-demand instances (red lines).

Prob.	**Prmt (#)**	**Inter. (hr)**	**Life (hr)**	**Fatal Fail. (#)**	**Nodes (#)**	**Thruput**	**Cost ($/hr)**	**Value**
0.01	8.50	2.08	15.20	0.06	45.18	87.99	41.11	2.10
0.05	48.15	0.44	10.14	0.23	43.65	76.35	39.73	1.90
0.10	99.77	0.23	6.71	0.29	41.69	72.12	37.94	1.88
0.25	276.52	0.10	3.13	1.04	35.80	60.12	32.58	1.82
0.50	709.83	0.06	1.49	5.98	26.96	40.37	24.53	1.59

Table 3.3: Results of simulating training BERT *until completion*; each preemption probability ran 1,000 times.

Prob.	**Thruput**	**Cost ($/hr)**	**Value**
0.01	54.87	90.73	0.60
0.05	50.66	87.43	0.58
0.10	49.18	83.23	0.59
0.25	40.59	71.24	0.57
0.50	26.24	53.05	0.49

Table 3.4: Simulation results of training BERT-large with pipeline depth P_h (which is $3.3 \times P_{demand}$).

To have a closer examination of Bamboo-S' training, we showed the traces for BERT-large and VGG-19, and plotted them in Figure 3.11. The two rows show (a) preemption traces (under the 10% rate), (b) training throughputs, (c) monetary costs, and (d) values, for BERT-large and VGG-19, respectively. Since Bamboo-M underperforms Bamboo-S, we focus on Bamboo-S in the rest of the evaluation.

3.6.2 Different Failure Models

The previous section demonstrated Bamboo running on real spot instances. This section demonstrates Bamboo's ability to affordably train large models across a wide range of failure models. To this end, we developed an offline simulation framework that takes as input (1) the preemption probability (including preemption frequency and the number of preemptions in each bulk), (2) per-iteration training time, and (3) Bamboo's recovery and reconfiguration time, automatically calculating training performance, costs, and values. Here we focus on BERT-large and simulated its training until completion.

We experimented using 5 different preemption probabilities (i.e., preemption rate per hour), and kept the preemption probability constant throughout the entire run (as opposed to re-playing traces). To mimic realistic spot instance creation and preemption, we randomly generated different creation probabilities per hour and also randomly picked zones for allocations. For each preemption probability, Table 3.3 reports the average numbers of preemptions, intervals (i.e., average time, in hours, between preemption events), average lifetime of an instance (in hours), average numbers of fatal failures (which require a restart from a checkpoint), average numbers of instances in the cluster, throughput (i.e., #samples per second), costs, and values, across 1,000 simulations.

Our simulations show that Bamboo's values match our real-world runs as just reported in §3.6.1. Further, regardless of the preemption probability, the value of Bamboo remains stable and is constantly higher than that of training with on-demand instances (which is 1.1). This is because most preemptions can be quickly recovered without introducing much overhead. The higher the preemption probability, the less the active instances running training jobs; this is the major source of the performance slowdown. However, the cost is reduced also proportionally, leading to stable values.

Simulation for P_h. To understand the tradeoff in choosing P, we experimented with another value of P for BERT-large: P_h, which is $\frac{3.06}{0.918} \times P_{demand}$. This configuration represents the *upper-bound* of the spot training resources that can be obtained within the cost of training with P_{demand} on-demand instances (while D remains unchanged). Note that in practice the number of active instances can barely reach the requested size and hence the cost of using a spot cluster of size $P_h \times D$ is often still much lower than training with an on-demand cluster of size $P_{demand} \times D$.

To avoid incurring a large monetary cost, we used the same simulator to run this experiment. These results are reported in Table 3.4. As shown, using P_h actually decreases both throughput (compared to 84 under P in Table 3.2) and value (due to significantly increased costs). This is because using too a large pipeline leads to poorer partitioning, underutilized resources and inferior performance.

3.6.3 Comparisons with Other Systems

We have reported the performance of training GPT-2 with asynchronous checkpointing and restart in Figure 3.3—the checkpointing-based approach spent only 23% on actual training, while Bamboo increases this percentage to **84%**. In fact, as shown in Table 3.3, even for the preemption rate of 0.5, there are only 5.98 fatal failures that would require checkpointing/restart under Bamboo. On the contrary, a checkpointing-based approach would need to restart the pipeline for every one of the 709.83 preemptions. Similarly, sample dropping significantly slows down the training when the preemption rate increases, as shown in Figure 3.4.

Varuna. Varuna [10] is a system developed concurrently with Bamboo to enable training on spot instances. As with other existing techniques, Varuna provides resilience with checkpointing. We set up Varuna on the same spot cluster on AWS EC2 as we used in § 3.6.1. We ran Varuna with a $D \times P$ pipeline (i.e., the same as on-demand instances) because Varuna does not use redundancies and hence not need to over-provision resources.

We trained BERT on Varuna with the same configurations, including the same datasets, model architectures, float precision, preemption rates, and hyperparameters. Varuna hang under the 33% preemption rate. For the 10% and 16% preemption rates, comparisons between Varuna and Bamboo-S are reported in Figure 3.12. As shown, Bamboo-S outperforms Varuna by **2.5**× and **2.7**× in throughput, respectively, under the 10% and 16% rates; and by **1.67**× and **1.64**×, in value, under these two rates. Note that value benefits are lower than throughput benefits due to Varuna's use of fewer instances.

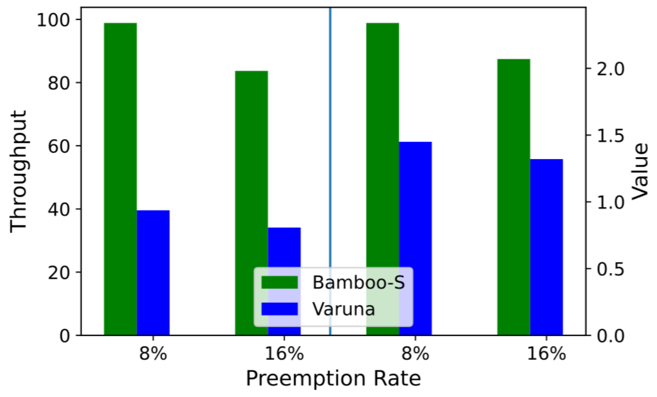


Figure 3.12: Throughput and value for Bamboo-S and Varuna running BERT at different preemption levels. Varuna hangs at the 33% preemption rate.

3.6.4 Microbenchmarks of Redundant Computation

To fully understand the overhead introduced by RC, we compared time and memory among three versions of RC: eager-FRC-lazy-BRC (EFLB, Bamboo's approach), eager-FRC-eager-BRC (EFEB), and lazy-FRC-lazy-BRC (LFLB), when training BERT and ResNet. Since the focus here is the RC overhead, we ran this experiment over on-demand instances.

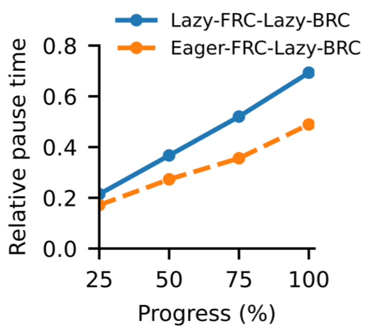
RC Setting	**BERT**	**ResNet**
Lazy-FRC-Lazy-BRC	7.01%	7.65%
Eager-FRC-Lazy-BRC (Bamboo)	19.77%	9.51%
Eager-FRC-Eager-BRC	71.51%	64.24%

Table 3.5: Time overhead with different RC settings.

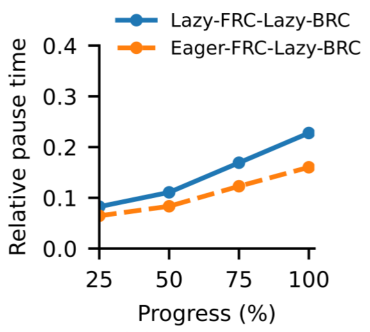
Table 3.5 reports RC's time overheads for the three RC settings. As expected, LFLB incurs the lowest per-iteration overhead because neither FRC nor BRC is performed with normal training iterations. The ∼7% overhead comes primarily from the extra code executed to prepare for a failover schedule. However, the recovery time is much longer under LFLB than the other two settings (discussed shortly). On the contrary, EFEB has the highest per-iteration overhead due to the eager execution of both FRC and BRC. The overhead incurred by EFLB, as used in Bamboo, is slightly higher than LFLB but much lower than EFEB. This is because eager FRC does not incur extra communication overhead and much its computation overhead can be hidden by scheduling it into the pipeline bubble and overlapping it with FNC.

Another interesting observation is the overhead for ResNet is lower than for BERT. This is because ResNet's layer partitioning is much more imbalanced than that of BERT (which is a transformer model where most the middle layers are equivalent). As a result, the bubble in ResNet's pipeline is much larger and hence it can accommodate a more significant fraction of FRC.

Eager FRC incurs an overall ∼1.5× overhead in GPU memory (that is why Bamboo recommends creating pipelines with 1.5× more nodes) while lazy FRC does not incur any memory overhead.



(a) BERT



(b) ResNet

Figure 3.13: Relative pause time for BERT and ResNet under different RC settings. Bamboo runs into a pause when a pipeline stops training and waits for the shadow node to recover the lost state on the victim node.

To understand the pause time under these different RC settings, Figure 3.13 shows the relative pause time (i.e., the actual pause time relative to the time of each training iteration without preemptions). As shown, lazy FRC reduces pause time by ∼35% despite the slightly higher per-iteration overhead it introduces. In summary, eager-FRC-lazy-BRC strikes the right balance between overhead and pause time.

3.7 Pipeline Reconfiguration

Reconfiguration introduces a much longer pause to the training process than recovering using RC. The goal of reconfiguration is to rebalance pipelines so they can withstand more failures as training progresses and continue to yield good performance. Reconfiguration also attempts to allocate more instances to maintain the cluster size. As shown in §3.3, asynchronous checkpointing is very efficient (but frequent restarting is not), and hence, Bamboo periodically checkpoints the model state. These checkpoints will not be used unless Bamboo restarts the training from a rare fatal failure (i.e., too many nodes are preempted so that training cannot continue).

Reconfiguration Triggering. Reconfiguration is triggered immediately when (1) consecutive preemptions occur simultaneously and (2) Bamboo determines that there is an urgent need to rebalance the pipelines at the end of an optimizer step. To do (2), the workers retrieve the cluster state from `etcd`, allowing them to see how many preemptions have occurred and in which pipeline they have occurred. They can also see how many workers are currently waiting to join the next rendezvous.

There are two main conditions for triggering reconfiguration at the end of an optimizer step: (a) the cluster has gained enough new nodes to reconstruct a new pipeline, and (b) Bamboo has encountered many preemptions and is close to a critical failure in the next step (e.g., encountering another preemption would cause us to suspend training), in which case we must pause the training to allocate more nodes.

Reconfiguration Policy. Bamboo attempts to maintain the pipeline depth P specified by the user. Therefore, our top priority at a reconfiguration is to reestablish a full pipeline of depth P. In this case, if we have had F failures and J ($> F$) nodes are waiting to join the cluster (i.e., new allocations arrive as Bamboo runs on the "spare tire"), we can fully recover all pipelines to depth P. The remaining $(J - F)$ nodes are placed in a standby queue to provide quick replacement upon future failures. However, if the number of nodes joining is smaller than F, we may end up having a number of N nodes such that $N\%P \neq 0$. In this case, instead of creating asymmetric pipelines (which complicates many operations), we move some nodes into the standby queue and decrease the total number of data-parallel pipelines. A final case is that the number of nodes joining, together with those in the standby queue, can form a new pipeline, and in this case we add a new pipeline to the system. In all these cases, the redundant layers are redistributed among the set of nodes participating in the updated pipelines.

How to Reconfigure. Once a reconfiguration is triggered, each node must be assigned a new stage (with new layers, state, and redundancies); it also needs to figure out if it will need to send or receive model and optimizer state from other nodes. Whichever nodes hits the rendezvous barrier first decides the new cluster configuration and puts the decision on `etcd` for all other nodes to read. To minimize the amount of data sent in layer transfer, Bamboo transfers layers in such a way that each node can reuse its old model and optimizer state as much as possible.

3.8 Support for Pure Data Parallelism

Bamboo supports pure data parallelism (without model partitioning). Due to space constraints, here we briefly discuss how it is supported. We use the same redundant computation strategy—Bamboo replicates the parameter and optimizer state of each node on a different node and uses these replicas as redundancies to provide quick recovery. For pure data parallelism, there is no bubble time to schedule RC. Eager FRC would be equivalent to overbatching (i.e., each node processes its original minibatch plus a redundant minibatch). To reduce the FRC overhead and make RC fit into the GPU memory constraints, we over-provision spot instances (by 1.5×, in the same way as discussed in §3.5) to make each node process a smaller batch.

Enabling eager FRC doubles the batch size. However, it results only in a ∼1.5× increase in the computation time due to the parallelism provided by GPUs. This overhead can be effectively reduced by slightly over-provisioning (1.5 $\times D$) nodes, increasing the degree of parallelism and decreasing the impact of overbatching. This enables us to run FRC eagerly without incurring much overhead (i.e., <10%).

3.9 Additional Experiments

3.9.1 Bubble Size

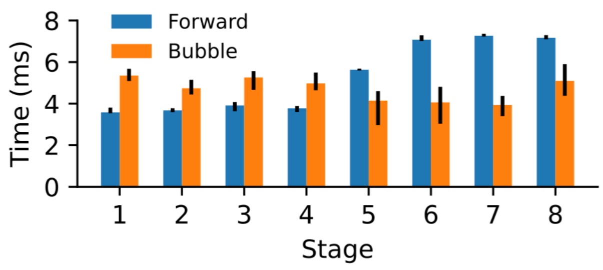


Figure 3.14: Comparison between bubble size and forward computation.

We measured the sizes of the pipeline bubble and forward computation of BERT with the same configuration as mentioned in Section 3.6, running on on-demand instances each with a single GPU. We manually inserted a barrier before each peer-to-peer communication, treating the time spent on the corresponding NCCL kernel as the bubble size. These results are reported in Figure 3.14.

To make memory evenly distributed across stages, more layers are placed on the last few stages. This explains the growth of forward computation. In this pipeline, for the first 4 stages, the bubble time is long enough to fit the entire FRC (i.e., the buble at stage 1 should run the forward computation for stage 2). For the last 4 stages, the bubble time is shorter than the forward computation time—it can still cover ∼60% of its FRC. The rest of the FRC on these nodes is run in parallel with their regular forward computation, as discussed in §3.5.2.

3.9.2 Cross-Zone Communication

Model	**Config**	**Throughput**	**Total Transferred Bytes**
BERT	Spread	148.923	16.39 GiB
BERT	Cluster	151.124	16.39 GiB
VGG19	Spread	160.12	11.213 GiB
VGG19	Cluster	165.77	11.213 GiB

Table 3.6: Comparison of throughput when running across availability zones compared to running within a single zone.

Because Bamboo allocates workers across availability zones to minimize the probability of reconfigurations, we measured We ran Bamboo in two configurations: (1) with nodes distributed across all zones (i.e., Spread) and (2) in a single availability zone with AWS' "Placement Group" option set to "Cluster" (i.e., Cluster), and measured their performance differences. As reported in Table 3.6, the differences between these two configurations are quite low (i.e., usually less than 5%). This demonstrates Bamboo's choice of assigning nodes from different availability zones as consecutive nodes in each pipeline has little impact on training performance.

3.9.3 Bamboo for Pure Data Parallelism

We ran two relatively small models such as VGG and ResNet using pure data parallelism with 8 workers (i.e., we partition the data but not the model). For Bamboo, we similarly over-provisioned 1.5× additional workers. We implemented another baseline *Checkpoint*, which periodically checkpoints model state for each worker and restarts the worker on another node when its original node is preemption. We used the same global batch size for these models as reported in §3.6. The comparisons between Bamboo, *Checkpoint*, and on-demand training are shown in Table 3.7.

Note that our implementation of *Checkpoint* assumes that there is always a standby node that is ready to join and load the checkpoint (which is a unrealistic over-approximation of the allocation model on any spot market); as such, the training cost remains unchanged and its throughput is reduced as the preemption rate increases.

Model	**System**	**Throughput**	**Cost ($/hr)**	**Value**
ResNet	Demand	24.51	24.48	1.01
	Checkpoint	[12.26, 8.42, 5.03]	[7.34, 7.34, 7.34]	[1.67, 1.15, 0.68]
	Bamboo	[**21.22**, 18.31, 12.31]	[**10.56**, 10.09, 9.18]	[**2.01**, 1.84, 1.34]
VGG	Demand	144.28	24.48	5.89
	Checkpoint	[83.21, 67.21, 45.31]	[7.34, 7.34, 7.34]	[11.33, 9.15, 6.17]
	Bamboo	[**125.59**, 96.51, 73.73]	[**10.56**, 10.09, 9.18]	[**11.89**, 9.56, 8.03]

Table 3.7: Comparison between pure data-parallel training over on-demand instances, a checkpoint-based approach on spot instances, Bamboo on spot instances. For *Checkpoint* and Bamboo, we trained each model three times, and their results are explicitly listed in the form of [*a*, *b*, *c*] for the 10% (average), 16%, and 33% preemption rates, respectively.

As shown, Bamboo outperforms *Checkpoint* by **1.64**× and **1.22**× in throughput and value. Both *Checkpoint* and Bamboo deliver a higher value than on-demand training (by **2**× and **1.79**×).

We make two observations on these numbers. First, Bamboo incurs a higher cost than *Checkpoint* due to resource over-provisioning. However, as discussed above, *Checkpoint* assumes the availability of standby nodes. In practice, guaranteeing such availability requires over-provisioning as well, but we did not take this into account when calculating costs (because it is hard to know exactly how many nodes we should over-provision). Hence, the cost and value reported for *Checkpoint* are the *lowerbound* and *upperbound* of those that can be achieved by any practical implementation of a checkpoint-based approach.

Second, *Checkpoint* works much better for pure data parallelism than for pipeline parallelism (as discussed in §3.3). This is because recovering from a checkpoint in pure data-parallel training is much easier than pipeline-parallel training where a pipeline reconfiguration process is needed for each restart.

3.10 Summary

With Bamboo, we have continued our look at how knowledge of ML workloads can be combined with understanding of heterogeneous cloud resources to increase the value provided by the system. By intelligently introducing computation redundancy into the system and exploiting the characteristics of pipeline parallelism, we were able to provide recovery for training in high failure environments that would otherwise be unusable. Specifically, we were able to provide nearly 2× the value of a full-priced on-demand baseline and 1.5× as much value as a checkpointing based approach designed to work with spot instances.

CHAPTER 4

Related Work

4.1 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* [19, 26, 28, 41, 117, 118, 133, 163], 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 [19, 41, 118] and distributed systems [25, 78, 163] have been developed for data-parallel training. Another parallelization strategy is to partition the work, often referred to as *model parallelism* [92] 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* [54, 61, 69, 94]. PipeDream [94] 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 [61] 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` [19] that broadcasts each worker's parameters to all other workers. Distributed systems including Dorylus use parameter servers [25, 78, 163], 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 [163] is an optimization of the BSP model.

Since synchronous training often introduces computation stalls, *asynchronous* training [19, 28] 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* [27, 94, 99, 141].

4.2 GNN Training and Graph Systems

As the GNN family keeps growing [31, 60, 75, 80, 151, 157–159, 161, 169], developing efficient and scalable GNN training systems becomes popular. GraphSage [43] uses graph sampling, NeuGraph [85] extends GNN training to multiple GPUs, and RoC [59] uses dynamic graph partitioning to achieve efficiency. Other systems that can scale to large graphs are all based on sampling [157, 161].

Programming frameworks such as DGL [30] 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.

4.3 Graph-Parallel Systems

There exists a body of work on scalable and efficient graph systems of many kinds: single-machine share-memory systems [37, 89, 90, 98, 130], disk-based out-of-core systems [7, 44, 72, 82, 86, 113, 138, 143, 145–147, 168, 171], and distributed systems [18, 21, 22, 40, 84, 88, 93, 112, 128, 139, 140, 142, 150, 165, 170]. 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.

4.4 Parallel Training of Large Models

There is a body of work on parallel training. Data parallelism [19, 28, 29, 61, 70, 78, 163, 163] is the most common parallelism model that partitions the dataset and trains on each partition. The learned weights are synchronized via either an all-reduce approach [19] or parameter servers [25, 78]. Model parallelism [29, 69, 101, 124, 129] partitions the operators in a DNN model across multiple GPU devices, with each worker evaluating and performing updates for only a subset of the model's parameters for all inputs. Recently, pipeline parallelism [54, 94, 134, 156] has been proposed to train large models by partitioning layers across workers and uses microbatches to saturate the pipeline. Popular DL training libraries such as DeepSpeed [109] and Megatron [96] support 3D parallelism, which combines data parallelism, model parallelism, and pipeline parallelism to train models at extremely large scale with improved compute and memory efficiency. Furthermore, DeepSpeed offers ZeRO-style data parallelism [110], which partitions model states across GPUs and uses communication collectives to gather individual parameters when needed during the training process, which offers better compute efficiency and scalability.

4.5 Elastic Training

Distributed training experiences frequent resource changes. There is a number of systems [46, 53, 56, 101, 106, 107] built to provide elasticity for training over changing resources. TorchElastic [106] is a PyTorch [102]-based tool that can dynamically kill or add data-parallel workers. Huang *et al.* [53] considers elasticity for declarative ML on MapReduce, which does not work for modern deep learning workloads. Litz [107] is a system that provides elasticity in the context of CPU-based machine learning using the parameter servers. Or *at al.* [101] presents an autoscaling system built on top of TensorFlow [6] and Horovod [119], which dynamically adapts the batch size and reuses existing processes upon resource changes. However, none of these works provide resilience for training over preemptible instances.

4.6 Exploiting Spot Instances

Proteus [46] exploits dynamic pricing on public clouds in order to lower costs for machine learning workloads through elasticity. Since Proteus does not explicitly consider modern deep learning workloads, Proteus simply reprocesses the input of a preempted node with another node. Varuna [10] is a system built concurrently with Bamboo for distributed training over spot instances. However, Varuna focuses on elasticity, not quick recovery from preemptions, based on an assumption that preemptions are not frequent. Bamboo, on the contrary, is designed specifically to deal with frequent preemptions.

There exists a body of work on enabling low latency and/or SLO guarantees when using preemptible spot instances. Tributary [45] is an elastic control system that exploits preemptible resources to reduce cost with SLO guarantees. Kingfisher [123] proposes a cost-aware resource acquisition scheme that uses integer linear programming to determine a service's resource footprint among a heterogeneous set of non-preemptible instances with fixed prices. Flint [120] is a system that runs batch-based data-intensive jobs on transient servers. SpotCheck [122] selects spot markets to acquire instances in while always bidding at a configurable multiple of the spot instance's corresponding on-demand price. BOSS [155] hosts key-value stores on spot instances by exploiting price differences across pools in different data-centers. ExoSphere [121] is a virtual cluster framework for spot instances. These systems are all orthogonal to Bamboo that is built specifically for deep learning training.

4.7 GPU Scheduling

There is also a large body of work on GPU scheduling [42, 76, 87, 95, 96, 105, 125, 154, 162, 166] for ML workloads. These techniques are orthogonal to Bamboo—they all focus on efficiency and throughput while Bamboo aims to perform redundant computation at a low cost.

CHAPTER 5

Future Directions and Conclusion

As the era of Machine Learning continues to progress and push into new and unforeseen areas, there is no doubt that new techniques will be developed to drive it forward. Unprecedented scale and new model architectures will allow ML researchers to push the boundaries of what applications are possible. These will lead to many new breakthroughs in artificial assistants, medical diagnosis, and many other fields, but they will also lead to continuously increasing costs, pushing the state of the art further and further out of reach of average users. Concurrently, the cloud appears to be increasingly heterogeneous in terms of both the hardware offered to users as well as the pricing schemes for that hardware, leading to new considerations and trade-offs for renting compute from the cloud. Already, we can see many companies offering customized silicon to improve the efficiency of computation, as well as advanced networking resource to speed up network bound workloads. Each of these comes with its own trade-off in terms of cost and the benefit to performance.

Many current and emerging workloads have the potential to benefit from this line of reasoning, to ensure that access to state of the art ML remains accessible and affordable while remaining performant and accurate.

- **Sparsely Gated Mixture of Experts** are an emerging type of network in which the input samples themselves can influence the computation of the model [47]. Specifically, a gating network learns which subset of "experts" or sub-networks should be activated based on the particular input sample. This allows for specialization but has dynamic and changing computation at every training step depending on which inputs are sampled, potentially leaving some subset of the experts idle. This can lead to over-provisioning, which seems to be a good fit for dynamically allocated compute resources, such as serverless, which can scale to the exact amount of computation needed.

- **Federated Learning** allows millions of clients to participate in training a model together over a very wide area [73]. For example, training a phone keyboard to better predict which word a user meant to type so that it can accurately correct them trains over millions of phones in a highly dynamic and uncertain environment. As they use LTE or 5G, these can be network constrained environments with constrained or variable network resources. The emerging smart-networking technologies have the potential to help accelerate this workload [74, 115] For example, switches have the ability to drop packets. By thoroughly understanding some notion of gradient utility (the importance of a gradient to the convergence of the model) we can drop certain gradients and prioritize others to help decongest the network while maintaining accurate training.

As we have seen, cost is becoming an increasingly important consideration in Machine Learning as the state of the art moves to a scale that is only achievable by companies with massive amounts of financial resources. This negatively impacts the field as a whole. In addition to fewer people being able to experiment with machine learning in new ways, there are also fewer researchers able to experiment with important problems like bias correction and assessing the potential harm machine learning can cause as it becomes more integrated into our daily lives. This dissertation took one step towards addressing this increasingly important issue, tackling the cost issue with both emerging types of neural networks as well as networks that have grown to be massive in size, in both cases leading to huge resource requirements for users.

With Dorylus, we were able to use cheap, scalable resources while taking advantage of specific model characteristic to hide any drawbacks of the cheaper resources. By utilizng the interleaving of graph and tensor workloads, we allowed for resource specialization and broke the workload up into discrete tasks for each resource used. In addition, we introduced asynchrony into the system to fully maximize the benefits of resource specialization and pipelining, allowing the system to fully overlap computation and communication. We also provided a theoretical analysis of the convergence guarantees this provides, as previous analyses of asynchronous ML did not consider feature staleness introduced by the Gather step in GNNs. We were able to increase the value compared to CPU-only and GPU-only backends for training GNNs by orders of magnitude on large, sparse graphs.

With Bamboo, we focused on bringing the benefits of cheap spot instances to training massive models with pipeline parallelism. By focusing on providing quick recovery through redundancy, we were able to overcome limitations that exist for checkpointing and approximation in high failure rate environments. We provided already existing backups in the pipeline to quickly recover in the event of a failure, reducing recovery time significantly. To overcome the overhead this could potentially add, we hid the redundant computation in the stalls inherent during pipeline parallel training. By combining these techniques, Bamboo decreased cost while maintaining performance, leading to an overall increase in value over both a full-priced, non-preemptible baseline and an existing system designed to work on spot instances.

APPENDIX A

Full Proof

To simplify the proof without introducing ambiguity, we refer to running asynchronous SGD on GNNs with asynchronous `Gather` as *asynchronous GNN training*, while running normal SGD with synchronous `Gather` as *synchronous GNN training*.

A.1 Proof of Theorem 1

Similar to the proof of Theorem 2 in [20], we prove Theorem 1 in 3 steps:

1. Lemma 1: Given a sequence of weights matrices $W^{(1)}, \ldots, W^{(N)}$ which are close to each other, approximate activations in asynchronous GNN training are close to the exact activations.

2. Lemma 2: Given a sequence of weights matrices $W^{(1)}, \ldots, W^{(N)}$ which are close to each other, approximate gradients in asynchronous GNN training are close to the exact gradients.

3. Theorem 1: Asynchronous GNN training generates weights that change slow enough for the gradient bias goes to zero, and thus the algorithm converges.

Let $\|A\|_\infty = \max_{i,j} |A(i,j)|$, we first state the following propositions:

Proposition 1.

- $\|AB\|_\infty \leq \text{col}(A)\|A\|_\infty\|B\|_\infty$, where $col(A)$ is the number of columns of matrix A.

- $\|A \odot B\|_\infty \leq \|A\|_\infty\|B\|_\infty$, where $\odot$ is element-wise product.

- $\|A+B\|_\infty \leq \|A\|_\infty + \|B\|_\infty$.

Let

$$C := \max\left\{\operatorname{col}(\hat{A}), \operatorname{col}\left(H^{(0)}\right), \ldots, \operatorname{col}\left(H^{(L)}\right),\right.$$
$$\left.\operatorname{col}\left(W^{(0)}\right), \ldots, \operatorname{col}\left(W^{(L)}\right)\right\}$$

be the largest number of columns of matrices we have in the proof, we have

- $\|AB\|_\infty \leq C\|A\|_\infty \|B\|_\infty$.

The proof can be found in Appendix C in [20].

Definition 1. (Mixed Matrix) We say matrix $\tilde{A}$ is a mixed matrix with N source matrices $\{A_1, \ldots, A_N\}$ iff. $\forall i, j,\ \exists! k \in [1, N]\ s.t.\ \tilde{A}(i,j) = A_k(i,j)$.
Conceptually, every element in a mixed matrix corresponds to one of the source matrices. And we also have the following proposition:

Proposition 2. Suppose that $\tilde{A}$ is a mixed matrices with $A_1, \ldots, A_N$ as its sources, then

$$\left\|\tilde{A}\right\|_\infty \leq \max_{k=1}^{N} \left\{\|A_k\|_\infty\right\}.$$

Proof. By Definition 1:

$$\begin{aligned}
\|\tilde{A}\|_\infty &= \max_{i,j} \left|\tilde{A}(i,j)\right| \\
&\leq \max_{i,j} \max_{k=1}^{N} |A_k(i,j)| \\
&= \max_{k=1}^{N} \max_{i,j} |A_k(i,j)| \\
&= \max_{k=1}^{N} \{\|A_k\|_\infty\}.
\end{aligned}$$

A.1.1 A single layer in GCN

As a base case, we first consider a single layer in a GCN model. In this case, at epoch i, the output activations H depends on weights W and input activations X only.
Under synchronous GNN training, we have

$$Z_i = \hat{A}X_iW_i, \quad H_i = \sigma(Z_i).$$

While under asynchronous GNN training with staleness bound S, we have

$$Z_{AS,i} = \hat{A}\tilde{X}_{AS,i}W_i, \quad H_{AS,i} = \sigma(Z_{AS,i}),$$

where $Z_{AS,i}$, $X_{AS,i}$, and $H_{AS,i}$ are corresponding approximate matrices of Z_i, X_i, and H_i. $\tilde{X}_{AS,i}$ is a mixed matrix of S stale activations $X_{AS,i-S+1}, \ldots, X_{AS,i}$.

Now we show that the approximate output activations under asynchronous GNN training are close to the exact ones when the weights change slowly during the training.

Proposition 3. Suppose that the activation $\sigma(\cdot)$ is ρ-Lipschitz, and for any series of T input activations and weights matrices $\{X_i, W_i\}_{i=1}^{T}$, where

1. matrices are bounded by some constant B: $\left\|\hat{A}\right\|_\infty \leq B$, $\|X_i\|_\infty \leq B$, $\|X_{AS,i}\|_\infty \leq B$, and $\|W_i\|_\infty \leq B$;

2. differences are bounded by ϵ: $\|X_{AS,i} - X_{AS,j}\|_\infty < \epsilon$, $\|X_{AS,i} - X_i\|_\infty < \epsilon$, and $\|W_i - W_j\|_\infty < \epsilon$.

Then there exists K that depends on C, B, and ρ, s.t. for all $S < i, j \leq T$, where S is the staleness bound:

1. The approximate outputs won't change too fast: $\|Z_{AS,i} - Z_{AS,j}\|_\infty < K\epsilon$, and $\|H_{AS,i} - H_{AS,j}\|_\infty < K\epsilon$,

2. The approximate outputs are close to the exact outputs: $\|Z_{AS,i} - Z_i\|_\infty < K\epsilon$, and $\|H_{AS,i} - H_i\|_\infty < K\epsilon$.

Proof. After the S warm up training epochs, we know for all $i > S$, $\tilde{X}_{AS,i}$ consists with either latest neighbor activations (i.e., $X_{AS,i}$) or activations from some previous epochs (i.e., $X_{AS,i-1}, \dots, X_{AS,i-S+1}$), and

$$\begin{aligned}\left\|\tilde{X}_{AS,i} - X_{AS,i}\right\|_\infty \leq & \max_{s\in[i-S+1,i)} \|X_{AS,i} - X_{AS,s}\|_\infty \\ \leq & \epsilon.\end{aligned}$$

By the triangle inequality,

$$\begin{aligned}\left\|\tilde{X}_{AS,i} - X_i\right\|_\infty &< 2\epsilon, \forall i \in (S, T] \\ \left\|\tilde{X}_{AS,i} - \tilde{X}_{AS,j}\right\|_\infty &< 3\epsilon, \forall i, j \in (S, T]\end{aligned}$$

By Prop. 2 and $\|X_{AS,i}\|_\infty \leq B$, we have $\left\|\tilde{X}_{AS,i}\right\|_\infty \leq B$. Thus, $\forall i, j \in (S, T]$, we have

$$\begin{aligned}
&\|Z_{AS,i} - Z_{AS,j}\|_\infty \\
=&\left\|\hat{A}\tilde{X}_{AS,i}W_i - \hat{A}\tilde{X}_{AS,j}W_j\right\|_\infty \\
=&\left\|\hat{A}\left(\tilde{X}_{AS,i} - \tilde{X}_{AS,j}\right)W_i + \hat{A}\tilde{X}_{AS,j}(W_i - W_j)\right\|_\infty \\
\leq&C^2\left(\left\|\hat{A}\right\|_\infty \left\|\tilde{X}_{AS,i} - \tilde{X}_{AS,j}\right\|_\infty \|W_i\|_\infty \right.\\
&\left. + \left\|\hat{A}\right\|_\infty \left\|\tilde{X}_{AS,j}\right\|_\infty \|W_i - W_j\|_\infty\right) \\
\leq&C^2(3\epsilon B^2 + \epsilon B^2) \\
=&4C^2B^2\epsilon,
\end{aligned}$$

and

$$\begin{aligned}
\|Z_{AS,i} - Z_i\|_\infty =&\left\|\hat{A}\tilde{X}_{AS,i}W_i - \hat{A}X_iW_i\right\|_\infty \\
=&\left\|\hat{A}\left(\tilde{X}_{AS,i} - X_i\right)W_i\right\|_\infty \\
\leq&C^2\left\|\hat{A}\right\|_\infty \left\|\tilde{X}_{AS,i} - X_i\right\|_\infty \|W_i\|_\infty \\
\leq&2C^2B^2\epsilon.
\end{aligned}$$

By the Lipschitz continuity of $\sigma(\cdot)$,

$$\begin{aligned}
\|H_{AS,i} - H_{AS,j}\|_\infty =&\|\sigma(Z_{AS,i}) - \sigma(Z_{AS,j})\|_\infty \\
\leq&\rho\|Z_{AS,i} - Z_{AS,j}\| \\
\leq&4\rho C^2B^2\epsilon, \\
\|H_{AS,i} - H_i\|_\infty =&\|\sigma(Z_{AS,i}) - \sigma(Z_i)\|_\infty \\
\leq&\rho\|Z_{AS,i} - Z_i\| \\
\leq&2\rho C^2B^2\epsilon.
\end{aligned}$$

We set $K = \max\{4C^2B^2\epsilon, 4\rho C^2B^2\epsilon\}$ thus all the differences are bounded by $K\epsilon$.

A.1.2 Lemma 1: Activations of Multi-layer GCNs

Now we generalize the conclusion to multi-layer GCNs by applying Prop. 3 layer by layer. The forward computation of a multi-layer GCN can be expressed as follows:
Under synchronous GNN training,

$$\begin{aligned} Z_i^{(l+1)} =& \hat{A} H_i^{(l)} W_i^{(l)} \qquad l = 1, \ldots, L-1 \\ H_i =& \sigma\left(Z_i^{(l+1)}\right) \qquad l = 1, \ldots, L-1 \end{aligned} \tag{A.1}$$

Under asynchronous GNN training,

$$\begin{aligned} Z_{AS,i}^{(l+1)} =& \hat{A} \tilde{H}_{AS,i}^{(l)} W_i^{(l)} \qquad l = 1, \ldots, L-1 \\ H_{AS,i}^{(l+1)} =& \sigma\left(Z_{AS,i}^{(l+1)}\right) \qquad l = 1, \ldots, L-1 \end{aligned} \tag{A.2}$$

The following lemma bounds the approximation error of output activations of a multi-layer GCN under asynchronous GNN training. The Lemma shows that the approximation error is bounded by the change rate of model weights only, regardless of the staleness bound S.

Lemma 1. Suppose that all the activations are ρ-Lipschitz. Given any series of T inputs and weights matrices $\left\{H_i^{(0)}, W_i\right\}_{i=1}^{T}$, where

1. $\left\|\hat{A}\right\|_\infty \leq B$, $\left\|H_i^{(0)}\right\|_\infty \leq B$, and $\|W_i\|_\infty \leq B$,
2. $\|W_i - W_j\|_\infty < \epsilon$,

there exists some constant K that depends on C, B, and ρ s.t. $\forall i > LS$,

1. $\left\|H_i^{(l)} - H_{AS,i}^{(l)}\right\|_\infty < K\epsilon, \quad l = 1, \ldots, L,$
2. $\left\|Z_i^{(l)} - Z_{AS,i}^{(l)}\right\|_\infty < K\epsilon, \quad l = 1, \ldots, L.$

Proof. By Prop. 3 and Eq. (A.1,A.2), for all $i > LS$, there exists a constant $K^{(1)}$ s.t. $\left\|H_i^{(1)} - H_{AS,i}^{(1)}\right\|_\infty < K^{(1)}\epsilon$ and $\left\|Z_i^{(1)} - Z_{AS,i}^{(1)}\right\|_\infty < K^{(1)}\epsilon$.

By applying Prop. 3 repeatedly for all L layers, we will get $K^{(1)}, K^{(2)}, \dots, K^{(L)}$, s.t.

$$\left\|H_i^{(2)} - H_{AS,i}^{(2)}\right\|_\infty < K^{(1)}K^{(2)}\epsilon, \dots, \left\|H_i^{(L)} - H_{AS,i}^{(L)}\right\|_\infty < K\epsilon,$$

and

$$\left\|Z_i^{(2)} - Z_{AS,i}^{(2)}\right\|_\infty < K^{(1)}K^{(2)}\epsilon, \dots, \left\|Z_i^{(L)} - Z_{AS,i}^{(L)}\right\|_\infty < K\epsilon,$$

where $K = \prod_{l=1}^{L} K^{(i)}$, for all $i > LS$.

A.1.3 Lemma 2: Gradients of Multi-layer GCNs

Denote $f(y, z)$ as the cost function that takes the model prediction y and the ground truth z. The gradients in the backpropagation can be computed as follows:
Under synchronous training,

$$\begin{aligned}
\nabla_{H^{(l)}} f &= \hat{A}^T \nabla_{Z^{(l+1)}} f W^{(l)^T} & l &= 1, \dots, L-1 \\
\nabla_{Z^{(l)}} f &= \sigma'\left(Z^{(l)}\right) \odot \nabla_{H^{(l)}} f & l &= 1, \dots, L \\
\nabla_{W^{(l)}} f &= \left(\hat{A}H^{(l)}\right)^T \nabla_{Z^{(l+1)}} f & l &= 0, \dots, L-1
\end{aligned} \tag{A.3}$$

Under asynchronous training,

$$\begin{aligned}
\nabla_{H_{AS}^{(l)}} f_{AS} &= \hat{A}^T \nabla_{Z_{AS}^{(l+1)}} f_{AS} W^{(l)^T} & l &= 1, \dots, L-1 \\
\nabla_{Z_{AS}^{(l)}} f_{AS} &= \sigma'\left(Z_{AS}^{(l)}\right) \odot \nabla_{H_{AS}^{(l)}} f_{AS} & l &= 1, \dots, L \\
\nabla_{W^{(l)}} f_{AS} &= \left(\hat{A}H_{AS}^{(l)}\right)^T \nabla_{Z_{AS}^{(l+1)}} f_{AS} & l &= 0, \dots, L-1
\end{aligned} \tag{A.4}$$

Denote the final loss function as $\mathcal{L}(W_i)$. Let $g_i(W_i) = \nabla\mathcal{L}(W_i)$, which is the gradient of $\mathcal{L}$ with respect to W_i under synchronous GNN training in epoch i. And let $g_{AS,i}(W_i) = \nabla\mathcal{L}_{AS,i}(W_i)$, which is the corresponding gradients of approximate $\mathcal{L}$ under asynchronous GNN training in epoch i. To simplify the proof, we construct the weight update sequences with layer-wise weight updates instead of epoch-wise weight updates. But either of them works for the proof. The following lemma bounds the difference between gradients under asynchronous GNN training and exact ones.

Lemma 2. Suppose that $\sigma(\cdot)$ and $\nabla_z f(y,z)$ are ρ-Lipschitz, and $\|\nabla_z f(y,z)\|_\infty \leq B$. For the given inputs matrix $H^{(0)}$ from a fixed dataset and any series of T weights matrices $\{W_i\}_{i=1}^T$, s.t.,

1. $\|W_i\|_\infty \leq B, \left\|\hat{A}\right\| \leq B,$ and $\|\sigma'(Z_{AS,i})\|_\infty \leq B,$
2. $\|W_i - W_j\|_\infty < \epsilon, \forall i, j,$

then there exists K that depends on C, B, and ρ s.t.

$$\|g_{AS,i}(W_i) - g_i(W_i)\|_\infty \leq K\epsilon, \forall i > LS.$$

Proof. By the Lipschitz continuity of ∇f and Lemma 1, for the final layer L, we have

$$\begin{aligned} \exists K^{(L)}, s.t.\ \left\|\nabla_{Z_{AS}^{(L)}} f_{AS} - \nabla_{Z^{(L)}} f\right\|_\infty \leq & \rho \left\|Z_{AS}^{(L)} - Z^{(L)}\right\|_\infty \\ \leq & \rho K^{(L)} \epsilon. \end{aligned} \tag{A.5}$$

Besides, by the Lipschitz continuity of $\sigma(\cdot)$ and Lemma 1,
$\exists \dot{K},\ s.t.\ \forall l \in [1, L], \left\|\sigma'\left(Z_{AS}^{(l)}\right) - \sigma'\left(Z^{(l)}\right)\right\|_\infty \leq \rho \dot{K} \epsilon.$
We prove by induction on l to bound the difference of $\nabla_{Z_{AS}} f_{AS}$ and $\nabla_Z f$ layer by layer in the back-propagation order, i.e., we will prove

$$\exists K^{(l)}, \forall l \in [1, L], \left\|\nabla_{Z_{AS}^{(l)}} f_{AS} - \nabla_{Z^{(l)}} f\right\|_\infty \leq K^{(l)} \epsilon. \tag{A.6}$$

Base case: by Eq. A.5, the statement holds for $l = L$, where $K^{(L)} = \rho \dot{K}$.
Induction Hypothesis (IH):

$$\forall l' > l, \left\|\nabla_{Z_{AS}^{(l')}} f_{AS} - \nabla_{Z^{(l')}} f\right\|_\infty \leq K^{(l')} \epsilon.$$

Then for layer l, by Eq. (A.3,A.4) and the induction hypothesis,

$$\begin{aligned}
&\left\|\nabla_{Z_{AS}^{(l)}} f_{AS} - \nabla_{Z^{(l)}} f\right\|_\infty \\
=&\left\|\sigma'\left(Z_{AS}^{(l)}\right) \odot \hat{A}^T \nabla_{Z_{AS}^{(l+1)}} f_{AS} W^{(l)^T}\right. \\
&\left.-\sigma'\left(Z^{(l)}\right) \odot \hat{A}^T \nabla_{Z^{(l+1)}} f W^{(l)^T}\right\|_\infty \\
\leq & C\left\{\left\|\left[\sigma'\left(Z_{AS}^{(l)}\right) - \sigma'\left(Z^{(l)}\right)\right] \odot \left[\hat{A}^T \nabla_{Z_{AS}^{(l+1)}} f_{AS}\right]\right\|_\infty \left\|W^{(l)^T}\right\|_\infty\right. \\
&\left.+\left\|\sigma'\left(Z^{(l)}\right) \odot \left[\hat{A}^T \left(\nabla_{Z_{AS}^{(l+1)}} f_{AS} - \nabla_{Z^{(l+1)}} f\right)\right]\right\|_\infty \left\|W^{(l)^T}\right\|_\infty\right\} \\
\leq & C^2\left\{\left\|\sigma'\left(Z_{AS}^{(l)}\right) - \sigma'\left(Z^{(l)}\right)\right\|_\infty \left\|\hat{A}^T\right\|_\infty \left\|\nabla_{Z_{AS}^{(l+1)}} f_{AS}\right\|_\infty \left\|W^{(l)^T}\right\|_\infty\right. \\
&\left.+\left\|\sigma'\left(Z^{(l)}\right)\right\|_\infty \left\|\hat{A}^T\right\|_\infty \left\|\nabla_{Z_{AS}^{(l+1)}} f_{AS} - \nabla_{Z^{(l+1)}} f\right\|_\infty \left\|W^{(l)^T}\right\|_\infty\right\} \\
\leq & C^2 \rho \dot{K} \epsilon B^3 + C^2 B^2 \rho K^{(l+1)} \epsilon B \\
=&\rho\left(\dot{K} + K^{(l+1)}\right) B^3 C^2 \epsilon.
\end{aligned}$$

Thus we set $K^{(l)} = \rho\left(\dot{K} + K^{(l+1)}\right) B^3 C^2$ and equation (A.6) holds.

Similarly, we can also bound the difference of $\nabla_W f_{AS}$ and $\nabla_W f$ in each layer with K_W:

$$\exists K_W^{(l)}, \forall l \in [0, L-1], \|\nabla_{W^{(l)}} f_{AS} - \nabla_{W^{(l)}} f\|_\infty \leq K_W^{(l)} \epsilon. \tag{A.7}$$

By inequalities (A.6, A.7), we have

$$\begin{aligned}
&\|g_{AS,i}(W_i) - g_i(W_i)\|_\infty \\
\leq &\max_{l \in [0, L-1]} \|\nabla_{W^{(l)}} f_{AS} - \nabla_{W^{(l)}} f\|_\infty \\
\leq & K\epsilon,
\end{aligned}$$

where $K = \max_{l \in [0, L-1]} \left\{K_W^{(l)}\right\}$.

A.1.4 Proof of Theorem 1

innercustomthmSuppose 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) 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. 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 GCN 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 $\gamma = min\left\{\frac{1}{\rho}, \frac{1}{\sqrt{N}}\right\}$.

Proof. We assume the asynchronous GNN training has run for LS epochs with the initial weights W_1 as a warm up, and Lemma 2 holds. Denote $\delta_i = g_{AS,i}\left(W_i\right) - \nabla\mathcal{L}\left(W_i\right) = g_{AS,i}\left(W_i\right) - g_i\left(W_i\right)$. By the smoothness of $\mathcal{L}$ we have

$$\begin{aligned}
&\mathcal{L}\left(W_{i+1}\right) \\
\leq&\mathcal{L}\left(W_i\right) + \left\langle\nabla\mathcal{L}\left(W_i\right), W_{i+1}-W_i\right\rangle + \frac{\rho}{2}\gamma^2\left\|g_{AS,i}\left(W_i\right)\right\|_F^2 \\
=&\mathcal{L}\left(W_i\right) - \gamma\left\langle\nabla\mathcal{L}\left(W_i\right), g_{AS,i}\left(W_i\right)\right\rangle + \frac{\rho}{2}\gamma^2\left\|g_{AS,i}\left(W_i\right)\right\|_F^2 \\
=&\mathcal{L}\left(W_i\right) - \gamma\left\langle\nabla\mathcal{L}\left(W_i\right), \delta_i\right\rangle - \gamma\left\|\nabla\mathcal{L}\left(W_i\right)\right\|_F^2 \\
&+\frac{\rho}{2}\gamma^2\left(\left\|\delta_i\right\|_F^2 + \left\|\nabla\mathcal{L}\left(W_i\right)\right\|_F^2 + 2\left\langle\delta_i, \nabla\mathcal{L}\left(W_i\right)\right\rangle\right) \\
=&\mathcal{L}\left(W_i\right) - \left(\gamma-\rho\gamma^2\right)\left\langle\nabla\mathcal{L}\left(W_i\right), \delta_i\right\rangle \\
&-\left(\gamma-\frac{\rho\gamma^2}{2}\right)\left\|\nabla\mathcal{L}\left(W_i\right)\right\|_F^2 + \frac{\rho}{2}\gamma^2\left\|\delta_i\right\|_F^2.
\end{aligned} \tag{A.8}$$

We firstly bound the inner product term $\langle \nabla \mathcal{L}(W_i), \delta_i \rangle$. For all i, consider the sequence of $LS+1$ weights: $\{W_{i-LS}, \ldots, W_i\}$:

$$\begin{aligned}
&\max_{i-LS \leq j,k \leq i} \|W_j - W_k\|_\infty \\
\leq &\sum_{j=i-LS}^{i-1} \|W_j - W_{j+1}\|_\infty \\
= &\sum_{j=i-LS}^{i-1} \gamma \|g_{AS,i}(W_j)\|_\infty \\
\leq &\sum_{j=i-LS}^{i-1} \gamma G = LSG\gamma.
\end{aligned}$$

By Lemma 2, there exists $\hat{K} > 0$, s.t.

$$\|\delta_i\|_\infty = \|g_{AS,i}(W_i) - g_i(W_i)\|_\infty \leq \hat{K} LSG\gamma, \forall i > 0.$$

Assume that W is D-dimensional, we have

$$\begin{aligned}
\langle \nabla \mathcal{L}(W_i), \delta_i \rangle \leq &D^2 \|\nabla \mathcal{L}(W_i)\|_\infty \|\delta_i\|_\infty \\
\leq &\hat{K} LSD^2G^2\gamma \\
\leq &K\gamma,
\end{aligned}$$

and

$$\begin{aligned}
\|\delta_i\|_F^2 \leq &D^2 \|g_{AS,i}(W_i) + \nabla \mathcal{L}(W_i)\|_\infty^2 \\
\leq &D^2 \|g_{AS,i}(W_i)\|_\infty^2 + D^2 \|\nabla \mathcal{L}(W_i)\|_\infty^2 \\
\leq &2D^2G^2 \\
\leq &K,
\end{aligned}$$

where $K = \max\left\{\hat{K}LSD^2G^2\gamma, 2D^2G^2\right\}$. Apply these two inequality to equation A.8, we get

$$\begin{aligned}\mathcal{L}\left(W_{i+1}\right) \leq & \mathcal{L}\left(W_i\right) + \left(\gamma - \rho\gamma^2\right) K\gamma \\ & - \left(\gamma - \frac{\rho\gamma^2}{2}\right) \left\|\nabla\mathcal{L}\left(W_i\right)\right\|_F^2 + \rho K\gamma^2/2\end{aligned}$$

Summing up the above inequalities for all i and rearranging the the terms, we get

$$\begin{aligned}&\left(\gamma - \frac{\rho\gamma^2}{2}\right) \sum_{i=1}^{N} \left\|\nabla\mathcal{L}\left(W_i\right)\right\|_F^2 \\ &\leq \mathcal{L}\left(W_1\right) - \mathcal{L}\left(W^*\right) + KN\left(\gamma - \rho\gamma^2\right)\gamma + \frac{\rho K}{2} N\gamma^2.\end{aligned}$$

Divide both sides of the summed inequality by $N\left(\gamma - \frac{\rho\gamma^2}{2}\right)$, and take $\gamma = \min\left\{\frac{1}{\rho}, \frac{1}{\sqrt{N}}\right\}$,

$$\begin{aligned}&\mathbb{E}_{R\sim P_R}\left\|\nabla\mathcal{L}\left(W_R\right)\right\|_F^2 = \frac{1}{N}\sum_{i=1}^{N}\left\|\nabla\mathcal{L}\left(W_i\right)\right\|_F^2 \\ \leq & \frac{\mathcal{L}\left(W_1\right) - \mathcal{L}\left(W^*\right) + KN\left(\gamma - \rho\gamma^2\right)\gamma + \frac{\rho K}{2}N\gamma^2}{N\gamma(2-\rho\gamma)} \\ \leq & \frac{\mathcal{L}\left(W_1\right) - \mathcal{L}\left(W^*\right) + KN\left(\gamma - \rho\gamma^2\right)\gamma + \frac{\rho K}{2}N\gamma^2}{N\gamma} \\ \leq & \frac{\mathcal{L}\left(W_1\right) - \mathcal{L}\left(W^*\right)}{N\gamma} + K\gamma(1-\rho\gamma) + \rho K\gamma \\ \leq & \frac{\mathcal{L}\left(W_1\right) - \mathcal{L}\left(W^*\right)}{\sqrt{N}} + K\gamma + \rho K/\sqrt{N} \\ \leq & \frac{\mathcal{L}\left(W_1\right) - \mathcal{L}\left(W^*\right) + K + \rho K}{\sqrt{N}}.\end{aligned}$$

Particularly, we have $\mathbb{E}_{R\sim P_R}\left\|\nabla\mathcal{L}\left(W_R\right)\right\|_F^2 \to 0$ when $N \to \infty$, which implies that the gradient under asynchronous GNN training is asymptotically unbiased. For simplicity purposes, we only prove the convergence of asynchronous GNN training for GCN here. However, the proof can be easily generalized to many other GNN models as long as they share similar properties in Lemma 1 and Lemma 2.

Bibliography

[1] Openai's gpt-3 language model: A technical overview. `https://lambdalabs.com/blog/demystifying-gpt-3/`.

[2] Understanding searches better than ever before. `https://blog.google/products/search/search-language-understanding-bert/`.

[3] Video analytics towards vision zero. `https://bellevuewa.gov/sites/default/files/media/pdf_document/2020/VideoAnalyticsTowardsVisionZero-TrafficVideoAnalytics-12262019.pdf`.

[4] Kubernetes: An open-source system for automating deployment, scaling, and management of containerized applications. https://kubernetes.io/, 2021.

[5] Operating etcd clusters for Kubernetes. https://kubernetes.io/docs/tasks/administer-cluster/configure-upgrade-etcd/, 2021.

[6] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283, 2016.

[7] 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.

[8] M. Allamanis, M. Brockschmidt, and M. Khademi. Learning to represent programs with graphs. In *6th International Conference on Learning Representations, ICLR 2018*, 2018.

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

[10] S. Athlur, N. Saran, M. Sivathanu, R. Ramjee, and N. Kwatra. Varuna: Scalable, low-cost training of massive deep learning models, 2021.

[11] AWS. Amazon ec2 spot instances pricing. https://aws.amazon.com/ec2/spot/pricing/, 2021.

[12] 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.

[13] Z. Bai, Z. Zhang, Y. Zhu, and X. Jin. PipeSwitch: Fast pipelined context switching for deep learning applications. In *OSDI*, pages 499–514, 2020.

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

[15] S. Boucher, A. Kalia, D. G. Andersen, and M. Kaminsky. Lightweight preemptible functions. In *USENIX ATC*, pages 465–477, 2020.

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

[17] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei. Language models are few-shot learners, 2020.

[18] 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.

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

[20] 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.

[21] 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.

[22] 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.

[23] T. Chen, B. Xu, C. Zhang, and C. Guestrin. Training deep nets with sublinear memory cost, 2016.

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

[25] 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.

[26] 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.

[27] 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 *Proceedings of 2014 USENIX Annual Technical Conference, Philadelphia, PA, June 2014 (ATC '14)*, pages 37–48. USENIX Association, June 2014.

[28] 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.

[29] J. Dean, G. S. Corrado, R. Monga, K. Chen, M. Devin, Q. V. Le, M. Z. Mao, M. Ranzato, A. Senior, P. Tucker, K. Yang, and A. Y. Ng. Large scale distributed deep networks. In *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*, NIPS'12, page 1223–1231, Red Hook, NY, USA, 2012. Curran Associates Inc.

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

[31] 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.

[32] J. Devlin, M. Chang, K. Lee, and K. Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. *CoRR*, abs/1810.04805, 2018.

[33] 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.

[34] S. Fan, Y. Rong, C. Meng, Z. Cao, S. Wang, Z. Zheng, C. Wu, G. Long, J. Yang, L. Xia, L. Diao, X. Liu, and W. Lin. DAPPLE: A pipelined data parallel approach for training large models. In *PPoPP*, pages 431–445, 2021.

[35] W. Fedus, B. Zoph, and N. Shazeer. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity, 2021.

[36] 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.

[37] S. Fushimi, M. Kitsuregawa, and H. Tanaka. An overview of the system software of A parallel relational database machine GRACE. In *VLDB'86 Twelfth International Conference on Very Large Data Basess*, pages 209–219, 1986.

[38] A. Gaut, T. Sun, S. Tang, Y. Huang, J. Qian, M. ElSherief, J. Zhao, D. Mirza, E. Belding, K.-W. Chang, and W. Y. Wang. Towards understanding gender bias in relation extraction. In *ACL*, 2020.

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

[40] 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.

[41] 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.

[42] J. Gu, M. Chowdhury, K. G. Shin, Y. Zhu, M. Jeon, J. Qian, H. Liu, and C. Guo. Tiresias: A gpu cluster manager for distributed deep learning. In *NSDI*, pages 485–500, 2019.

[43] W. L. Hamilton, Z. Ying, and J. Leskovec. Inductive representation learning on large graphs. In *Annual Conference on Neural Information Processing Systems 2017*, pages 1024–1034, 2017.

[44] 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.

[45] A. Harlap, A. Chung, A. Tumanov, G. R. Ganger, and P. B. Gibbons. Tributary: spot-dancing for elastic services with latency SLOs. In *USENIX ATC*, pages 1–14, 2018.

[46] A. Harlap, A. Tumanov, A. Chung, G. R. Ganger, and P. B. Gibbons. Proteus: agile ML elasticity through tiered reliability in dynamic resource markets.

[47] J. He, J. Qiu, A. Zeng, Z. Yang, J. Zhai, and J. Tang. Fastmoe: A fast mixture-of-expert training system. *CoRR*, abs/2103.13262, 2021.

[48] K. He, X. Zhang, S. Ren, and J. Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *2015 IEEE International Conference on Computer Vision (ICCV)*, pages 1026–1034, 2015.

[49] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *CVPR*, pages 770–778, 2016.

[50] 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.

[51] 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.

[52] 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 *Proceedings of the 26th International Conference on Neural Information Processing Systems - Volume 1*, NIPS'13, page 1223–1231, Red Hook, NY, USA, 2013. Curran Associates Inc.

[53] B. Huang, M. Boehm, Y. Tian, B. Reinwald, S. Tatikonda, and F. R. Reiss. Resource elasticity for large-scale machine learning. In *SIGMOD*, pages 137–152, 2015.

[54] 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.

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

[56] C. Hwang, T. Kim, S. Kim, J. Shin, and K. Park. Elastic resource sharing for distributed deep learning. In *NSDI*, pages 721–739, 2021.

[57] 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.

[58] M. Jeon, S. Venkataraman, A. Phanishayee, J. Qian, W. Xiao, and F. Yang. Analysis of Large-Scale Multi-Tenant GPU clusters for DNN training workloads. In *USENIX ATC*, pages 947–960, 2019.

[59] 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.

[60] 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.

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

[62] 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.

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

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

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

[66] M. Kirisame, S. Lyubomirsky, A. Haan, J. Brennan, M. He, J. Roesch, T. Chen, and Z. Tatlock. Dynamic tensor rematerialization, 2020.

[67] 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.

[68] 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.

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

[70] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 25. Curran Associates, Inc., 2012.

[71] 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.

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

[73] F. Lai, X. Zhu, H. V. Madhyastha, and M. Chowdhury. Oort: Efficient federated learning via guided participant selection. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pages 19–35. USENIX Association, July 2021.

[74] C. Lao, Y. Le, K. Mahajan, Y. Chen, W. Wu, A. Akella, and M. Swift. ATP: In-network aggregation for multi-tenant learning. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 741–761. USENIX Association, Apr. 2021.

[75] 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.

[76] Y. Lee, A. Scolari, B.-G. Chun, M. D. Santambrogio, M. Weimer, and M. Interlandi. PRETZEL: Opening the black box of machine learning prediction serving systems. In *OSDI*, pages 611–626, 2018.

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

[78] 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 *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 583–598, 2014.

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

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

[81] H. Lin, H. Zhang, Y. Ma, T. He, Z. Zhang, S. Zha, and M. Li. Dynamic Mini-batch SGD for Elastic Distributed Training: Learning in the Limbo of Resources. *CoRR*, 2019.

[82] 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.

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

[84] 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.

[85] 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.

[86] 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.

[87] K. Mahajan, A. Balasubramanian, A. Singhvi, S. Venkataraman, A. Akella, A. Phanishayee, and S. Chawla. Themis: Fair and efficient GPU cluster scheduling. In *NSDI*, pages 289–304, 2020.

[88] 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.

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

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

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

[92] 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.

[93] 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.

[94] 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.

[95] D. Narayanan, K. Santhanam, F. Kazhamiaka, A. Phanishayee, and M. Zaharia. Heterogeneity-aware cluster scheduling policies for deep learning workloads. In *OSDI*, 2020.

[96] D. Narayanan, M. Shoeybi, J. Casper, P. LeGresley, M. Patwary, V. Korthikanti, D. Vainbrand, P. Kashinkunti, J. Bernauer, B. Catanzaro, A. Phanishayee, and M. Za-

haria. Efficient large-scale language model training on gpu clusters using Megatron-LM. In *SC*, 2021.

[97] A. Newell, D. Skarlatos, J. Fan, P. Kumar, M. Khutornenko, M. Pundir, Y. Zhang, M. Zhang, Y. Liu, L. Le, B. Daugherty, A. Samudra, P. Baid, J. Kneeland, I. Kabiljo, D. Shchukin, A. Rodrigues, S. Michelson, B. Christensen, K. Veeraraghavan, and C. Tang. RAS: Continuously optimized region-wide datacenter resource allocation. In *SOSP*, pages 505–520, 2021.

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

[99] 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.

[100] NVIDIA. cusparse. https://docs.nvidia.com/cuda/cusparse/index.html, 2020.

[101] A. Or, H. Zhang, and M. Freedman. Resource elasticity in distributed deep learning. In I. Dhillon, D. Papailiopoulos, and V. Sze, editors, *MLSys*, volume 2, pages 400–411, 2020.

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

[103] D. A. Patterson, G. Gibson, and R. H. Katz. A case for redundant arrays of inexpensive disks (RAID). In *SIGMOD*, pages 109–116, 1988.

[104] 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.

[105] Y. Peng, Y. Bao, Y. Chen, C. Wu, and C. Guo. Optimus: An efficient dynamic resource scheduler for deep learning clusters. In *EuroSys*, 2018.

[106] PyTorch Developers. TorchElastic, 2021.

[107] A. Qiao, A. Aghayev, W. Yu, H. Chen, Q. Ho, G. A. Gibson, and E. P. Xing. Litz: Elastic framework for High-Performance distributed machine learning. In *USENIX ATC 18*, pages 631–644, 2018.

[108] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever. Language models are unsupervised multitask learners. 2019.

[109] S. Rajbhandari, J. Rasley, O. Ruwase, and Y. He. Zero: Memory optimization towards training A trillion parameter models. *CoRR*, abs/1910.02054, 2019.

[110] S. Rajbhandari, J. Rasley, O. Ruwase, and Y. He. Zero: Memory optimizations toward training trillion parameter models. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–16. IEEE, 2020.

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

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

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

[114] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. S. Bernstein, A. C. Berg, and L. Fei-Fei. ImageNet large scale visual recognition challenge. *CoRR*, abs/1409.0575, 2014.

[115] A. Sapio, M. Canini, C.-Y. Ho, J. Nelson, P. Kalnis, C. Kim, A. Krishnamurthy, M. Moshref, D. Ports, and P. Richtarik. Scaling distributed machine learning with In-Network aggregation. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 785–808. USENIX Association, Apr. 2021.

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

[117] 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.

[118] 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.

[119] A. Sergeev and M. D. Balso. Horovod: fast and easy distributed deep learning in tensorflow. *CoRR*, abs/1802.05799, 2018.

[120] P. Sharma, T. Guo, X. He, D. E. Irwin, and P. J. Shenoy. Flint: Batch-Interactive Data-Intensive Processing on Transient Servers.

[121] P. Sharma, D. Irwin, and P. Shenoy. Portfolio-driven resource management for transient cloud servers. In *SIGMETRICS*, page 59, 2017.

[122] P. Sharma, S. Lee, T. Guo, D. Irwin, and P. Shenoy. SpotCheck: Designing a derivative IaaS cloud on the spot market. In *EuroSys*, 2015.

[123] U. Sharma, P. Shenoy, S. Sahu, and A. Shaikh. A cost-aware elasticity provisioning system for the cloud. In *ICDCS*, pages 559–570, 2011.

[124] N. Shazeer, Y. Cheng, N. Parmar, D. Tran, A. Vaswani, P. Koanantakool, P. Hawkins, H. Lee, M. Hong, C. Young, R. Sepassi, and B. A. Hechtman. Mesh-TensorFlow: Deep Learning for Supercomputers.

[125] H. Shen, L. Chen, Y. Jin, L. Zhao, B. Kong, M. Philipose, A. Krishnamurthy, and R. Sundaram. Nexus: A gpu cluster engine for accelerating dnn-based video analysis. In *SOSP*, pages 322–337, 2019.

[126] E. Sheng, K.-W. Chang, P. Natarajan, and N. Peng. Towards controllable biases in language generation. In *EMNLP-Finding*, 2020.

[127] E. Sheng, K.-W. Chang, P. Natarajan, and N. Peng. Societal biases in language generation: Progress and challenges. In *ACL*, 2021.

[128] 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.

[129] M. Shoeybi, M. Patwary, R. Puri, P. LeGresley, J. Casper, and B. Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism, 2020.

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

[131] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. In Y. Bengio and Y. LeCun, editors, *ICLR*, 2015.

[132] K. S. Tai, R. Socher, and C. D. Manning. Improved semantic representations from tree-structured long short-term memory networks. *CoRR*, abs/1503.00075, 2015.

[133] 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.

[134] 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. In *OSDI*, pages 495–514, 2021.

[135] J. Thorpe, P. Zhao, J. Eyolfson, Y. Qiao, Z. Jia, M. Zhang, R. Netravali, and G. H. Xu. Bamboo: Making preemptible instances resilient for affordable training of large dnns, 2022.

[136] I. Turc, M. Chang, K. Lee, and K. Toutanova. Well-Read Students Learn Better: The Impact of Student Initialization on Knowledge Distillation. *CoRR*, 2019.

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

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

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

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

[141] 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.

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

[143] 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.

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

[145] 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.

[146] 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.

[147] 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.

[148] 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. Smola, and Z. Zhang. Deep graph library: Towards efficient and scalable deep learning on graphs, 2019.

[149] T. Wang, J. Huan, and B. Li. Data dropout: Optimizing training data for convolutional neural networks. In *ICTAI*, pages 39–46, 2018.

[150] 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.

[151] S. Wu, Y. Tang, Y. Zhu, L. Wang, X. Xie, and T. Tan. Session-based recommendation with graph neural networks. *Proceedings of the AAAI Conference on Artificial Intelligence*, 33:346–353, Jul 2019.

[152] Y. Wu, M. Schuster, Z. Chen, Q. V. Le, M. Norouzi, W. Macherey, M. Krikun, Y. Cao, Q. Gao, K. Macherey, J. Klingner, A. Shah, M. Johnson, X. Liu, L. Kaiser, S. Gouws, Y. Kato, T. Kudo, H. Kazawa, K. Stevens, G. Kurian, N. Patil, W. Wang, C. Young, J. Smith, J. Riesa, A. Rudnick, O. Vinyals, G. Corrado, M. Hughes, and J. Dean. Google's neural machine translation system: Bridging the gap between human and machine translation. *CoRR*, abs/1609.08144, 2016.

[153] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, and P. S. Yu. A comprehensive survey on graph neural networks, 2019.

[154] W. Xiao, S. Ren, Y. Li, Y. Zhang, P. Hou, Z. Li, Y. Feng, W. Lin, and Y. Jia. AntMan: Dynamic scaling on GPU clusters for deep learning. In *OSDI*, pages 533–548, 2020.

[155] Z. Xu, C. Stewart, N. Deng, and X. Wang. Blending on-demand and spot instances to lower costs for in-memory storage. In *INFOCOM*, pages 1–9, 2016.

[156] B. Yang, J. Zhang, J. Li, C. Ré, C. R. Aberger, and C. D. Sa. PipeMare: Asynchronous Pipeline Parallel DNN Training. In *MLSys*, 2019.

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

[158] 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.

[159] 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.

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

[161] 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.

[162] H. Zhang, L. Stafman, A. Or, and M. J. Freedman. SLAQ: Quality-driven scheduling for distributed machine learning. In *SoCC*, pages 390–404, 2017.

[163] 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.

[164] 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.

[165] 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.

[166] Q. Zhang, R. Zhou, C. Wu, L. Jiao, and Z. Li. Online scheduling of heterogeneous distributed machine learning jobs. In *MobiHoc*, pages 111–120, 2020.

[167] M. K. Zhang Xianyi. Openblas. https://www.openblas.net, 2019.

[168] 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.

[169] J. Zhou, G. Cui, Z. Zhang, C. Yang, Z. Liu, L. Wang, C. Li, and M. Sun. Graph neural networks: A review of methods and applications, 2018.

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

[171] 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.