Priority-based Parameter Propagation for Distributed Deep Neural Network Training

by

Anand Jayarajan

B.Tech, National Institute of Technology Calicut, India, 2012

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF

Master of Science

in

THE FACULTY OF GRADUATE AND POSTDOCTORAL STUDIES
(Computer Science)

The University of British Columbia
(Vancouver)

August 2019

The following individuals certify that they have read, and recommend to the Faculty of Graduate and Postdoctoral Studies for acceptance, the thesis entitled:

Priority-based Parameter Propagation
for
Distributed Deep Neural Network Training

submitted by **Anand Jayarajan** in partial fulfillment of the requirements for the degree of **Master of Science** in **Computer Science**.

Examining Committee:

Alexandra Fedorova, Electrical and Computer Engineering
Supervisor

Ivan Beschastnikh, Computer Science
Supervisor

Abstract

Data parallel training is commonly used for scaling distributed Deep Neural Network (DNN) training. However, the performance benefits are often limited by the communication-heavy parameter synchronization step. In this work, we take advantage of the domain specific knowledge of DNN training and overlap parameter synchronization with computation in order to improve the training performance. We make two key observations: (1) the optimal data representation granularity for the communication may differ from that used by the underlying DNN model implementation and (2) different parameters can afford different synchronization delays. Based on these observations, we propose a new synchronization mechanism called *Priority-based Parameter Propagation* (*P3*). *P3* synchronizes parameters at a finer granularity and schedules data transmission in such a way that the training process incurs minimal communication delay. We show that *P3* can improve the training throughput of ResNet-50, Sockeye and VGG-19 by as much as 25%, 38% and 66% respectively on clusters with realistic network bandwidth.

Lay Summary

Deep learning based models are replacing traditional techniques used in machine learning by providing superior prediction accuracy on fields like computer vision, speech recognition and machine translation. However, achieving such predictive power require the models to be iteratively trained on enormous amount of input data, usually of the order of Giga Bytes. Fortunately, DNN training algorithms like Stochastic Gradient Descent (SGD) are highly data parallel and training process can be sped up by distributing the workload on large cluster of machines. This transforms DNN training a communication bound workload from a computation-bound workload. In this work, we present optimizations to reduce the communication overhead by taking advantage of the domain specific knowledge about the training algorithm.

Preface

All of the work presented henceforth was conducted by lead investigator and primary author Anand Jayarajan during his internship at Vector Institute, Toronto and in collaboration with researchers Prof. Gennady Pekhimenko from the University of Toronto, Jinliang Wei and Dr. Garth Gibson from the Carnegie Mellon University and Prof. Alexandra Fedorova from the University of British Columbia. The lead investigator is responsible for all major areas of concept formation, data collection, evaluation and analysis, as well as the manuscript composition.

Table of Contents

Abstract . . . . **iii**

Lay Summary . . . . **iv**

Preface . . . . **v**

Table of Contents . . . . **vi**

List of Figures . . . . **viii**

Glossary . . . . **ix**

Acknowledgments . . . . **x**

1 Introduction **1**

1.1 Our Approach 4

2 Background **6**

3 Limitations of parameter synchronization **10**

4 Design and Implementation **14**

4.1 KVStore: Baseline system 16

4.2 *P3*: Implementation 17

5 Evaluation **19**

5.1 Methodology 19

5.2 Summary of the experiments 20
5.3 Bandwidth vs. throughput 20
5.4 Network utilization 22
5.5 Scalability 24
5.6 Training accuracy 25
5.7 Parameter slice size selection 26

6 Related Work 28

7 Conclusion and Future Work 31

Bibliography 33

A Additional Results 39
A.1 Comparison with other frameworks 39
A.2 P3 vs. Asynchronous SGD 40

List of Figures

Figure 1.1 Training iterations 3

Figure 2.1 Deep neural network structure 6
Figure 2.2 Parameter server architecture 8

Figure 3.1 Current parameter synchronization mechanism 11
Figure 3.2 Parameter distribution 12

Figure 4.1 Coarse and fine granularity 15
Figure 4.2 P3Worker and P3Server 17

Figure 5.1 Bandwidth v.s. Throughput 21
Figure 5.2 Network utilization of the baseline system 22
Figure 5.3 Network utilization of *P3* 23
Figure 5.4 Throughput scaling with different number of machines 24
Figure 5.5 *P3* v.s. DGC 25
Figure 5.6 Granularity v.s. Throughput 26

Figure A.1 Network utilization 39
Figure A.2 ASGD v.s. P3 40

Glossary

DNN	Deep Neural Network
SGD	Stochastic Gradient Descent
COHESA	Computing Hardware for Emerging Intelligent Sensory Applications
NSERC	National Sciences and Engineering Research Council of Canada
ML	Machine Learning
MPI	Message Passing Interface
FC	Fully Connected
AWS	Amazon Web Services
DGC	Deep Gradient Compression
WFBP	Wait-Free-Back-Propagation

Acknowledgments

Firstly, I would like to thank my advisor Prof. Alexandra Fedorova for introducing me to the emerging and exciting field of systems research for machine learning and giving me enough freedom to pursue any research problems of my interest. I'm also grateful to Prof. Gennady Pekhimenko (University of Toronto) for giving me guidance on the research field that was completely new to me. I want to extend my gratitude towards Dr. Garth Gibson for giving me the opportunity to do an internship at Vector Institute and for the immense support he provided during the course of this project. Most importantly, I like to thank my co-author Jinliang Wei for being an excellent mentor and a great friend.

I shout-out to all the members of the Networks, Systems, and Security lab especially to my co-advisor Prof. Ivan Beschastnikh. Even though I didn't get to work with him, he was always helpful and supportive during the two years I spent at the University of British Columbia. Finally, I thank Computing Hardware for Emerging Intelligent Sensory Applications (COHESA) for providing generous funding during my graduate years (COHESA is financed under the National Sciences and Engineering Research Council of Canada (NSERC) Strategic Networks grant number NETGP485577-15).

Chapter 1

Introduction

Deep learning has revolutionized machine learning field with superior predictive power and adaptability to a wide range of applications such as computer vision [18], machine translation [40] and speech recognition [7]. However, Deep Neural Network (DNN)s are notoriously expensive to train because of their high computing power requirement and need to process large data set. One of the key enabling factors of the advances in deep learning was the availability of enormous computing power provided by application specific hardware accelerators like GPUs, FPGAs and ASICs [21, 28]. These devices can provide massive parallel processing power of the order trillions of floating point operations per second [29]. The highly parallel nature of DNN training algorithms makes it well suited to scale on such devices.

As modern DNNs keep demanding higher computing power and memory, single device training is no longer a sustainable solution because of their computing and memory resource limitations. This is where distributed training become relevant. Data parallel distribution with synchronous Stochastic Gradient Descent (SGD) is one of the most popular methods for accelerating the training by parallelizing data processing over a cluster of machines [12].

In data parallel training, the input data set is sharded among the worker machines and they train a shared DNN model iteratively by independently computing updates to the parameters of the model and synchronizing the updates at the end of each iteration. A single iteration on a worker machine involves three steps: (1) *forward propagation* to calculate the value of a loss function on a subset of the local data shard, (2) *backward propagation* to compute the gradients for every model parameter based on the computed loss, and finally (3) the *parameter synchronization* step to aggregate local gradients from all the worker machines and update the parameters using optimization algorithms like SGD [9].

In an iteration, each worker machine generates and synchronizes hundreds of megabytes of gradient values [3]. This often makes data parallel training a communication-bound workload [42]. Handling this traffic require high bandwidth networks like Gigabit Ethernet [14] or InfiniBand [33]. However, these technologies are yet to be widely adopted because of the high deployment cost. Moreover, the bandwidth requirement data parallel DNN training is increasing with the emergence of larger models and faster hardware accelerators as it leads to increase in data volume and transmission rate [29, 34]. Most of the major cloud providers and academic clusters have trouble in catering to such high bandwidth demands [26]. In this work, we propose solutions to scale data parallel training under limited bandwidth conditions.

Gradient compression is one of the popular approaches aimed at reducing the communication overhead. Since gradient values are generally represented as floating point numbers, it is challenging to achieve reasonable compression ratios from lossless compression techniques [10]. Instead, recent studies propose lossy compression techniques like gradient quantization [5, 32, 39] and sparse parameter synchronization [2, 25]. These methods, however, risk affecting the final convergence accuracy of the model because of the information loss that comes with the value approximation and stale parameter updates [23].

An orthogonal approach is to utilize the available network bandwidth more efficiently by leveraging domain specific opportunities in DNN training. The traffic generated during the training process is generally bursty because of iterative nature of the training algorithm. Some distributed Machine Learning (ML) frameworks attenuate these traffic bursts by overlapping communication with computation. Since the training computation is performed as a sequence of operations (*layers*), during backward propagation, the gradients for the parameters associated with each layer are generated one after another. Frameworks such as TensorFlow [1], MXNet [13] and Caffe2 [20], exploit this sequential layered structure to overlap parameter synchronization with backpropagation by issuing synchronization of each layer immediately after its gradients are computed.

In this thesis, we find new opportunities to better overlap communication and computation. Our first observation is that the domain specific knowledge of the DNN training algorithm allows us to better schedule parameter synchronization, not only based on when the gradients are generated, but also based on *when the data is consumed*. During training, the gradients of the layers are generated from final to initial layers and subsequently consumed in the reverse order in the next iteration. Figure 1.1 shows a snapshot of the training process containing the backward propagation of one iteration and the forward propagation of the next one. The temporal gap between gradients generated and consumed per layer are higher for final layers compared to the initial ones. Scheduling parameter synchronization using this information can help to overlap communication with both the forward and the backward propagation.

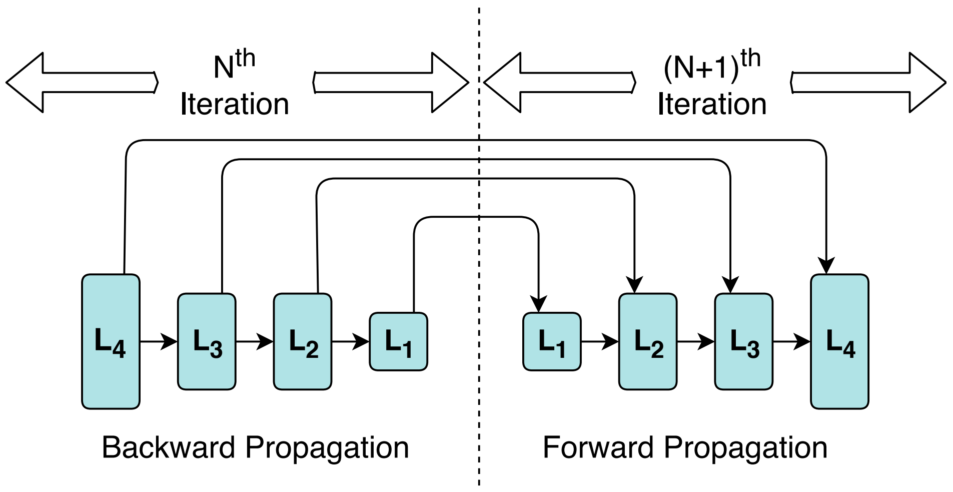


Figure 1.1: Training iterations

Secondly, the layer-wise granularity used by the underlying neural network implementation may not always be optimal for parameter synchronization. In our experiments, for certain heavy models (e.g., VGG [36], Sockeye [19]), parameter synchronization at a finer granularity improves the network utilization and reduces the communication delay.

Based on these observations, we propose a new synchronization mechanism called *Priority-based Parameter Propagation* (*P3*).

1.1 Our Approach

P3 consists of two key components: (1) *Parameter Slicing*: *P3* splits the layers into smaller slices and synchronize them independently. (2) *Priority-based Update*: *P3* synchronizes the parameter slices based on their priority, where the priority of a slice is defined by when it is required again in the subsequent iteration. During backpropagation, *P3* always allocates network cycles to the highest priority slices in the queue, preempting synchronization of the slices from a previous lower priority layer if necessary.

P3 offers the following advantages over state-of-the-art parameter synchronization mechanisms [25, 41]. (1) *P3* can provide improved training performance under limited bandwidth conditions by better overlapping communication with computation and utilizing the available network bandwidth more efficiently. (2) *P3* is model-agnostic, its implementation requires minimal programming effort, and all required changes are localized within the framework. (3) *P3* always communicates full gradients and does not affect model convergence.

In summary, this paper makes the following contributions:

- We show that parameter synchronization at layer-wise granularity can cause suboptimal resource utilization in heavy models (e.g., VGG, Sockeye). We also show that the parameter synchronization can be scheduled better to efficiently use the available network bandwidth by taking into account not only the information on when the gradients are generated, but also when they are consumed.

- We present a new parameter synchronization mechanism called *Priority-based Parameter Propagation* (*P3*), which uses parameter slicing and priority-based updates to reduce communication overhead. We demonstrate that *P3* has better resiliency towards bandwidth limitations compared to other state-of-the-art synchronization mechanisms [41].

- We implement and open source *P3*[1] on MXNet [13], a popular distributed

[1] https://github.com/anandj91/p3

ML framework, and evaluate its performance against the standard MXNet implementation as the baseline. We observe that, *P3* improves the training performance of several state-of-the-art models like ResNet-50 [18], Sockeye [19] and VGG-19 [36] by as much as 25%, 38%, and 66% respectively.

Chapter 2

Background

The fundamental building blocks of DNNs are mathematical operations, such as convolution, matrix multiplication, and activation functions. These operations perform certain transformations ($f_\theta(x)$) on an input vector (x) using the parameters (θ) associated with the function. A DNN is defined by a sequence of such operations (layers). In Figure 2.1, the initial layer takes the application-specific data samples as input and produces a prediction as an output vector at the output layer. The goal of the training algorithm is to find the parameter values which can make the most accurate predictions.

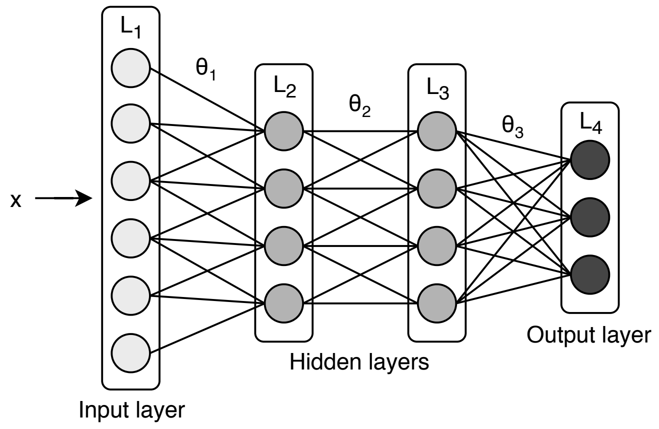


Figure 2.1: Deep neural network structure

DNN training usually starts from a random parameter initialization and iterates by randomly sampling input vectors from the training dataset. On every iteration, the DNN computes the output vector on the inputs and calculates the error associated with the prediction (loss) by feeding the output vector to a loss function. This step is called *forward propagation*. After that, a *backward propagation* step is performed to calculate the error contribution of each parameter by computing gradients of all the layers with respect to the loss. The backward propagation method for calculating gradients is based on the chain rule of derivatives and is therefore performed in the reverse order of forward propagation, i.e., gradients of the final layer are calculated first and the process moves back towards the input layer, hence the name backward propagation [31]. Once the gradients are calculated, the parameters are updated using an optimization algorithm like SGD [9]. The iterations are repeated several times over the training dataset until the model converges to an acceptable prediction accuracy.

Each iteration is compute-intensive, which makes the training process time consuming. The total training time can be reduced by distributing the workload onto multiple machines by taking advantage of the data parallel nature of the SGD algorithm. Data parallel training [22] involves multiple workers simultaneously training a shared DNN with the training dataset distributed equally among them. On each iteration, workers independently calculate the gradients locally for a common parameter value assignment but on different input data samples. Then, the gradients are aggregated in a synchronous fashion for performing parameter updates. This method is called a *synchronous* SGD algorithm [12].

Parameter server architecture [24] is one of the most popular methods used in practice for parameter synchronization and it is widely supported in most of the distributed ML frameworks (e.g., MXNet [13], TensorFlow [1], Caffe2 [20]). The parameter server keeps track of the up-to-date values of all the model parameters. Before every iteration, each worker machine reads the latest parameter values (θ) from the parameter server and locally computes gradients for the inputs sampled from its data shard. The workers then send the local gradients ($\triangledown$) to the parameter server. The parameter server waits until it receives gradient updates from all worker machines, then aggregates the gradients together and updates the parameters for the next iteration.

Figure 2.2 shows parameter server-based data parallel training in a four-node cluster. The communication between worker machine and parameter server is usually over a network and often becomes the bottleneck in achieving linear scalability in data parallel training [35, 42].

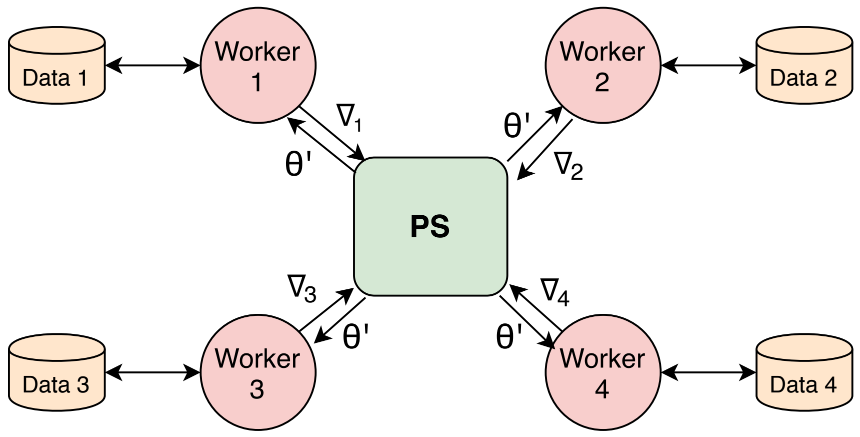


Figure 2.2: Parameter server architecture

The state-of-the-art ML framework MXNet is designed specifically for making data parallel training efficient and easy to execute. It comes with a built-in implementation of a parameter server called *KVStore*. In MXNet, worker machines send gradients of a layer to the *KVStore* as soon as they are calculated, and issue parameter pull requests once all the other workers have finished sending the gradients for that layer. This aggressive parameter synchronization makes data parallel training efficient in MXNet.

TensorFlow, on the other hand, is designed as a more generic ML framework. Hence it does not have an explicit parameter server implementation. However, a parameter server can be implemented on top of the graph computation framework provided by TensorFlow. Since the parameter server is a part of the computation graph, the communication between the worker subgraph and parameter server subgraph is handled by the framework itself. TensorFlow automatically places *Send* and *Recv* operations on the edges of the computation graph that crosses the device boundaries. Similar to MXNet, the worker subgraph executes the send operation as soon as the gradients are computed. However, since every training iteration is a separate graph execution, the parameter pull request is not issued until the start of the next iteration. This disconnection in sending gradients and receiving parameter updates could cause underutilization of bidirectional network bandwidth.

Despite small differences described above, we observe that state-of-the-art ML frameworks (e.g., MXNet, TensorFlow, Caffe2.) have two common characteristics. For performance reasons, the operations in the DNN implementation usually perform computations on large data representations. Therefore, the gradients for all the parameters within a layer are usually generated in a single shot. We observe that, because the gradients are generated at the layer level granularity, frameworks perform parameter synchronization at the same granularity as well. We also observe that since the DNN implementation is written as a dependency graph in these frameworks, the gradients of the layers are sent out to the parameter server over the network as soon as the backward propagation of that layer has completed. In this work, we analyze the limitations associated with these two characteristics of ML frameworks and propose a mechanism for more efficient parameter synchronization.

Apart from parameter server architecture, there are other mechanisms used for gradient aggregation. For example, there are many variations of Message Passing Interface (MPI) *all_reduce* operation specifically designed for ML workloads [8, 15]. In this work, we implement *P3* over the parameter server architecture in MXNet. However, we believe, *P3* design principles (namely, parameter slicing and priority-based propagation) are general enough to be applied to any gradient aggregation method.

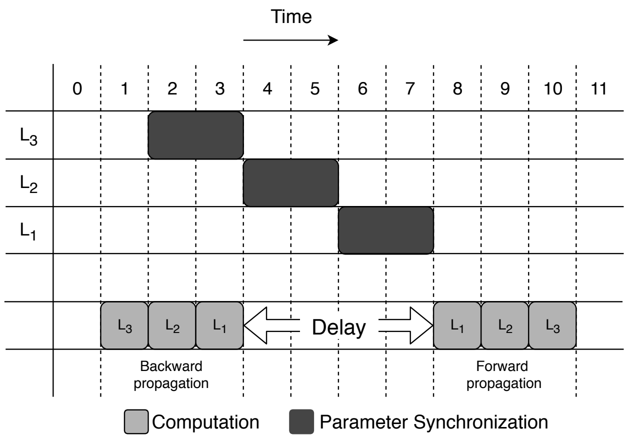
Chapter 3

Limitations of parameter synchronization

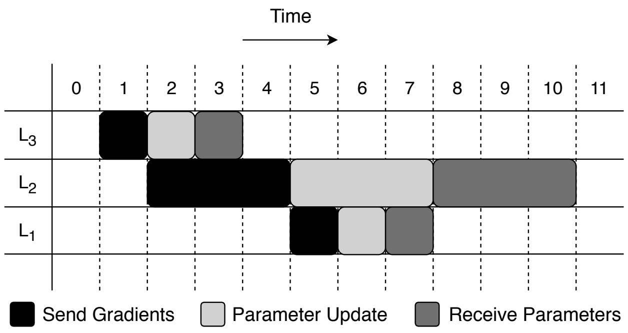
Current parameter synchronization mechanisms have major limitations in effectively utilizing available network bandwidth for two main reasons. First is because of the aggressive synchronization performed by the frameworks where the gradients of the layers are sent to the parameter server immediately after finishing the backward propagation of that layer. Since the backward propagation progresses from the final to the initial layer, the gradients are also generated and propagated in that order. However, the next forward propagation can only start after receiving the updated parameters of the *first layer*. We observe that, under limited bandwidth, gradient propagation of the final layers can induce queuing delay onto the gradient propagation of the initial layers and subsequently delay the next iteration. This prevents the communication from overlapping with the forward propagation.

Figure 3.1a shows the parameter synchronization of a 3-layer DNN, where the forward and backward propagation of each layer takes one time unit and parameter synchronization takes two time units. With aggressive synchronization, the total delay between the two iterations is twice the time taken for synchronizing the first layer because of the additional queuing delay induced by the previous layers. Moreover, during forward propagation the network remains idle.

This effect becomes even more noticeable when the communication time required for individual layers varies due to the presence of Fully Connected (FC)



(a) Aggressive synchronization

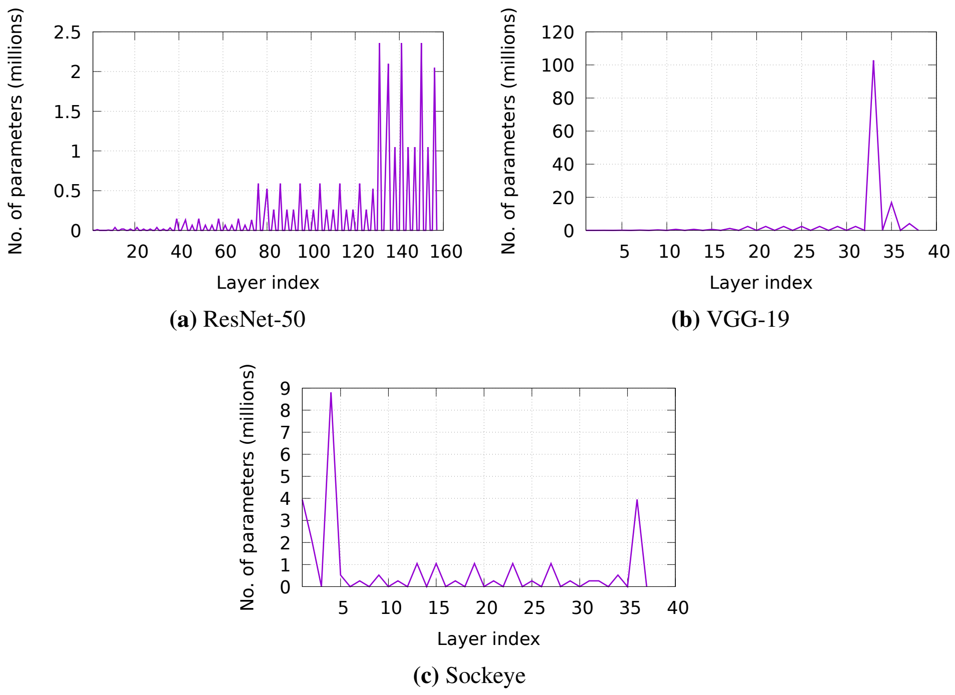


(b) Layer level granularity

Figure 3.1: Current parameter synchronization mechanism

layers in the DNN, as the synchronization time needed for such dense layers is relatively higher. Figure 3.2 shows the parameter distribution of two popular image classification models: ResNet-50, VGG-19, and a machine translation model: Sockeye. The skewed parameter size distribution is a general trend in image classification models where the final FC layers are usually heavier and can potentially induce higher queuing delay for the lighter initial convolution layers.

The second limitation associated with current parameter synchronization mechanisms comes from conducting synchronization at a layer-level granularity. The communication time of parameter synchronization consists of three parts: (1) *gradient propagation* time for the worker machine to send the gradients to the param-



(a) ResNet-50 **(b)** VGG-19 **(c)** Sockeye

Figure 3.2: Parameter distribution

eter server, (2) *parameter update* time taken by the parameter server to aggregate the gradients and update the parameters, and (3) *parameter propagation* time taken by the parameter server to send the updated parameters back to worker machine(s). As we describe in Section 2, current distributed ML frameworks overlap gradient propagation of one layer with the backward propagation of the next one. On top of this, at the parameter server side, the gradient propagation of a layer is overlapped with the parameter update of the previous layer. This type of communication-computation pipelining is effective only if the size of the layers are more or less uniform. Unfortunately, this is usually not the case. For example, Figure 3.2b shows that VGG-19 contains a single FC layer which has 71.5% of all the parameters in the entire network. We observe that the disproportionately heavy layers like this can negatively affect the utilization of network bidirectional bandwidth.

This effect is explained in Figure 3.1b using the previous example of parameter synchronization of the 3-layered DNN. In this case, gradient propagation, parameter update, and parameter propagation of the second layer take three times as long as that of the first and third layers. Because of this imbalance, the communication delay in this model is dominated by the second layer. The parameter synchronization of the first and the third layer can only be partially overlapped with the second layer. As seen in the example, this severely underutilizes the computing resources and bidirectional bandwidth by spending the last three time steps just for receiving parameter updates from the parameter server.

From the above observations we draw two major conclusions. First, the application domain-specific knowledge of DNNs can be utilized to schedule communication not only based on the data is generated in the backward propagation, but also based on when the data is consumed in the subsequent forward propagation. Scheduling parameter synchronization based on this information and sending the gradients conservatively could reduce the delay by better overlapping communication with both the forward and the backward propagation. Second, the optimal granularity required for parameter synchronization may differ from the one used for data representation by the underlying model implementation. Synchronizing parameters at a finer granularity can better utilize the available computing and networking resources as we empirically show in Section 5.4.

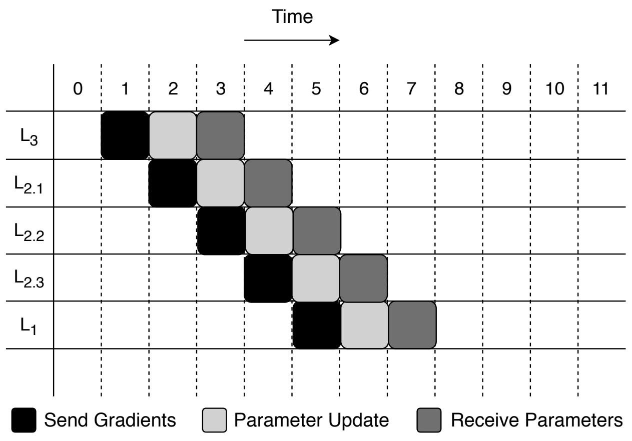
Chapter 4

Design and Implementation

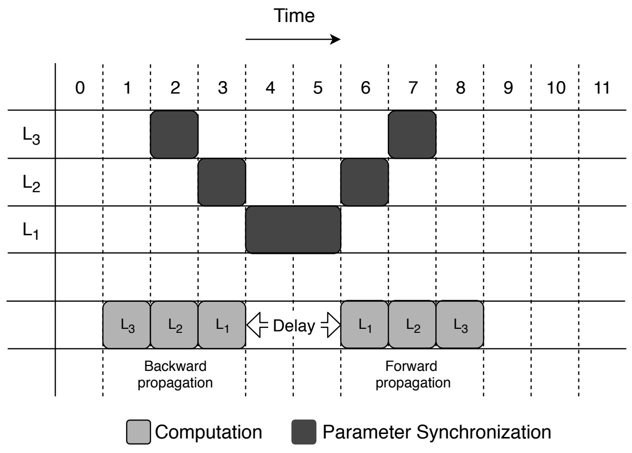
Based on the observations in Chapter 3, we propose a new method for parameter synchronization called Priority-based Parameter Propagation (*P3*). As explained in Section 1.1, *P3* has two core components: (1) *parameter slicing*, and (2) *priority-based update*.

Once the gradients are computed, gradient aggregation and updates of each parameter can be performed independent of each other. We take advantage of this property of the SGD algorithm for parameter slicing optimization. *P3* splits the layers into smaller slices of parameters and each of these slices are then independently synchronized. In Figure 4.1a, applying parameter slicing optimization on the second layer achieves better overlap between data transmission and parameter update. Moreover, the bidirectional bandwidth is completely utilized as the synchronization of slices are perfectly pipelined. In this example, parameter slicing reduces the communication cost by 30%.

After splitting the layers into smaller pieces, *P3* assigns priorities to each slice. The slices inherit priority values from their parent layers. We determine layers' priorities based on the order in which they are processed in the forward propagation. The first layer gets the highest priority and the priority decrements moving towards to the end, with final layer having the lowest priority. During backward propagation, parameter synchronization of the slices are issued based on their priorities as illustrated in Figure 4.1b. In this example, with the priority-based update, the delay between the two iterations has been reduced by half and the communication



(a) Fine granularity



(b) Priority based synchronization

Figure 4.1: Coarse and fine granularity

is evenly overlapped with both the forward and the backward propagation.

We implemented *P3* by modifying the parameter server module in MXNet called KVStore. Below, we first explain how the baseline KVStore works, and then we describe our modifications to support *P3*.

4.1 KVStore: Baseline system

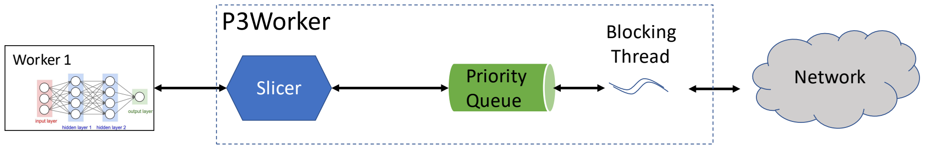
KVStore is a wrapper implemented on top of the light-weight parameter server called *ps-lite* [24]. KVStore has two components: *KVWorker*, which runs locally to the worker machine as part of the training process and a separate server process, called *KVServer*. KVServer receives gradients from KVWorkers, aggregates them, and updates the parameters while ensuring data consistency. The parameters are stored as key-value pairs, where the key is the index of a layer and the value is an array of floating points each corresponding to the parameter values of that layer. For load balancing purposes, more than one KVServer can be used for the training with the parameters equally sharded between them. For better resource utilization, a common practice is to run one KVServer on every machine along with the worker process.

Before starting the training process, KVStore initializes and distributes the parameters of all the layers among the KVServers. KVStore follows a simple heuristic for fair distribution of parameters. Layers with size smaller than a fixed threshold are assigned to a randomly chosen KVServer. Parameters of larger layers are split equally among the KVServers. This is different from parameter slicing used in *P3* (explained in Section 4.2). The threshold is a configurable parameter and is set to 10^6 parameters by default.

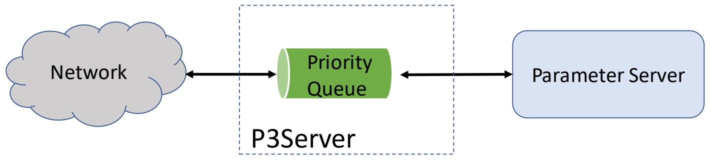
KVServer exposes two interfaces to KVWorker for sending gradients and requesting updated parameters: a *Push* request and a *Pull* request. During training, MXNet issues a parameter synchronization request for a layer to the KVServer through the KVWorker as soon as the backpropagation of that layer has finished. KVWorker serializes (and fragments in case of large layers) the gradient matrix and issues a Push request to the corresponding KVServer(s). KVServer waits until it has received gradient updates from all the workers for that key-value pair. Once all the updates have been received, KVServer aggregates the gradients and updates the parameters.

Once the parameters are updated, KVServer notifies all the workers. When KVWorker receives a notification, it immediately issues a Pull request to the KVServer(s) for the corresponding updated parameter values. KVServer then sends the latest parameter values in response, and KVWorker (reconstructs the parameter array for large layers) updates the local parameter values for the next iteration. MXNet overlaps the parameter synchronization with backward propagation by asynchronously issuing Push requests for the layers whose gradients are ready to be propagated.

4.2 *P3*: Implementation



(a) P3Worker



(b) P3Server

Figure 4.2: P3Worker and P3Server

To implement *P3*, we modified KVWorker and KVServer into *P3*Worker and *P3*Server as shown in Figure 4.2. On the worker side (Figure 4.2a), when a parameter synchronization is issued, *P3*Worker splits the gradient matrix of the layer based on a predefined size threshold (explained in more details in Section 5.7). Unlike in KVStore, this threshold defines the maximum granularity into which layers can be split. This is the parameter slicing part of *P3*. Each of these slices are assigned to *P3*Servers in a round-robin fashion.

The priority-based gradient propagation is implemented using a producer-consumer mechanism through a priority queue. After parameter slicing, the producer part of *P3*Worker assigns priorities to the individual slices and pushes them into the priority queue. A separate consumer thread in the *P3*Worker continuously polls the highest priority slice from the queue and sends the slice to the *P3*Server through the network with its priority added to the packet header. The consumer thread uses blocking network calls. Hence the rate at which the priority queue polled is automatically adjusted based on networking delay. This simple producer-consumer model makes sure that the network does not experience bursty traffic flow from the *P3*Worker, and that the backward propagation is not hindered at the worker side. Also the slice with the highest priority in the queue always gets the first preference for transmission.

We also introduce similar producer-consumer mechanism at the receiver in the *P3*Server in order to deal with the in-network delays as illustrated in Figure 4.2b. The packets received at the *P3*Server are pushed into a priority queue along with the priority assigned by the *P3*Worker. A server consumer thread then polls from this queue and processes the packets the same way as in a KVServer. Prioritization at the *P3*Server ensures highest priority parameters are processed first.

Apart from these modifications, we remove the explicit update notification and pull requests from the KVServer. *P3*Server immediately broadcasts the updated parameters to all workers once it has received all of the updates. Since workers always issue a pull request after every push, this change does not affect the correctness of the training algorithm. This modification was necessary, because otherwise MXNet only issues a pull request once it has received the update notification for all the slices of a layer. Eliminating this helped to improve the bidirectional bandwidth utilization.

Chapter 5

Evaluation

5.1 Methodology

We have evaluated the *P3* implementation on three image classification models: ResNet-50 [18], InceptionV3 [37], VGG-19 [36] and on a machine translation model, Sockeye [19]. We chose the standard MXNet KVStore implementation described in Section 4.1 as the baseline in all performance evaluation experiments.

Since *P3* implementation does not interfere with the model implementation or the training algorithm, the model convergence is not affected in any way. This means that the baseline and the *P3* would follow the same training curve for a given hyper-parameter set. The improvement in training performance is completely determined by the rate at which input data is processed. Therefore the primary performance comparison metric we use is the training throughput, which is the number of total training samples processed by the worker machines in one second. We measure the throughput after training the models for a few iterations until the throughput has become stable and then averaged over thousand iterations. In all the experiments, we set the number of KVServers/*P3*Servers equal to the number of worker machines.

5.2 Summary of the experiments

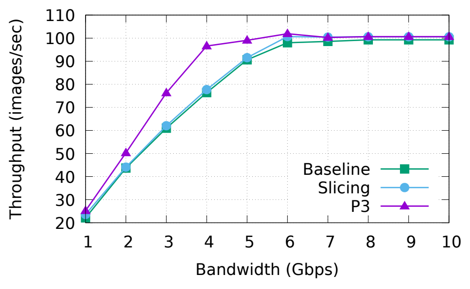
We conduct performance evaluation of *P3* in three different experiments. Section 5.3 shows *P3* 's resiliency towards the bandwidth constraints in the network. We perform this experiment by training the model on a four machine cluster, each equipped with one Nvidia P4000 GPU [28] and interconnected with a 100Gbps InfiniBand network [27]. We measure throughput variation while artificially limiting the network interface transmission rate. This simulates more realistic networking infrastructure in modern cloud services where bandwidth is usually between 1Gbps and 25Gbps [16]. Section 5.4 shows how well *P3* utilizes the available bandwidth and reduces the network idle time. Finally, in Section 5.5 we test the scalability of *P3* on different cluster sizes. This experiment is conducted on Amazon Web Services (AWS) [6] using g3.4xlarge machine instances on a 10Gbps network.

In Section 5.6, we compare the convergence accuracy for models trained using *P3* and compression based techniques. For this comparison study, we pick the state-of-the-art compression technique Deep Gradient Compression (DGC) [25]. We implemented DGC on MXNet based on the details provided in the original paper and information collected from the authors. In addition to these experiments, we have also evaluated the effects of different parameter slice sizes on the training throughput in Section 5.7. We have also included additional results in Appendix A.

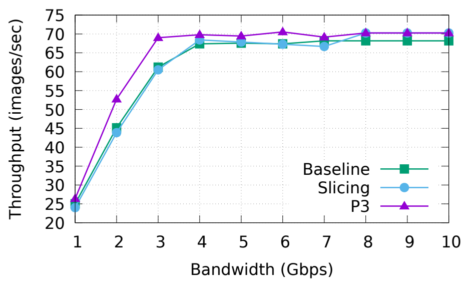
5.3 Bandwidth vs. throughput

In this experiment, we measure the training throughput of ResNet-50, InceptionV3, VGG-19, and Sockeye on a tightly controlled four-machine cluster by setting different transmission rates on the network interface on all the machines using Linux's *tc qdisc* utility [4]. Figure 5.1 compares the throughput from *P3* against the baseline system for different network bandwidths between 1Gbps and 30Gbps. We also measured the performance benefits achieved from the parameter slicing optimization alone.

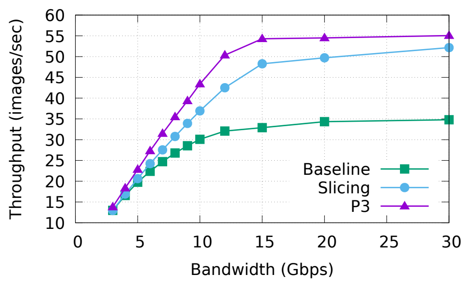
In Figure 5.1a and 5.1b, both baseline and *P3* give similar training performance when the network bandwidth is sufficiently high (over 6Gbps and 4Gbps respectively) for scaling these models on four machines. However, the baseline



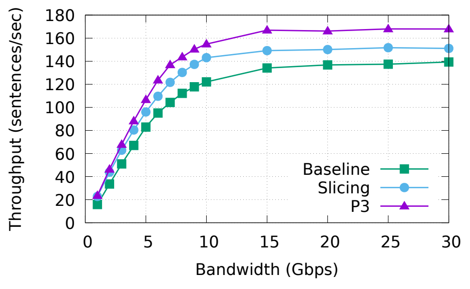
(a) ResNet-50



(b) InceptionV3



(c) VGG-19



(d) Sockeye

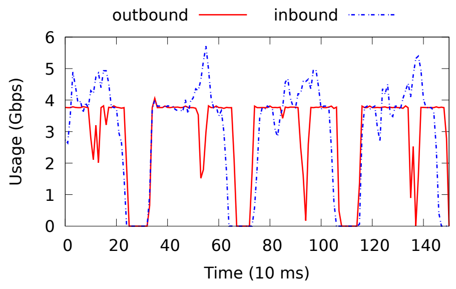
Figure 5.1: Bandwidth v.s. Throughput

throughput starts to drop in ResNet-50 below 6Gbps. At the same time, *P3* maintains the linear throughput until the bandwidth drops below 4Gbps. This is because *P3* reduces the peak bandwidth required for the model by efficiently overlapping communication with the computation. At 4Gbps, *P3* provides 26% more throughput than the baseline. For InceptionV3, the maximum speed up obtained is 18%. It is interesting to note that these models do not benefit from parameter slicing, as the layer sizes are relatively small in these DNNs (see Figure 3.2a).

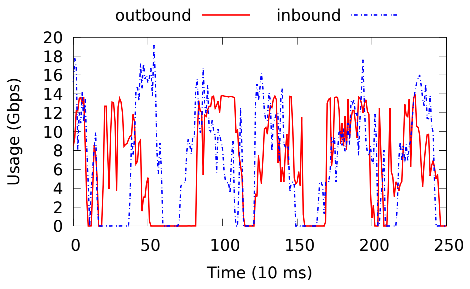
Figure 5.1c and 5.1d show the throughput of VGG-19 and Sockeye respectively. These models contain one or more very large layers (Figure 3.2b and 3.2c), and because of the presence of these large layers, the parameter slicing optimization alone is giving considerable improvement in performance. At 30Gbps, parameter slicing can provide 49% speedup on VGG-19. The speedup is further improved with *P3* by as much as 66% at 15Gbps. Sockeye is a special case among other models. Unlike image classification models, the heaviest layer in this model is the initial layer. In Figure 5.1d, Sockeye performance has improved by a maximum of 38% with *P3*.

We observe that *P3* always performs better than the baseline with relatively higher performance benefits when bandwidth is limited. Since *P3* reduces the peak network bandwidth required for the training, it is more suitable than baseline on a shared network cluster where effective bandwidth available for a single training process is much lower than the maximum capacity of the network. However, the speed-up diminishes when the network bandwidth is too low. This is because the communication time is significantly higher than computation and there is little room for improvement by overlapping communication and computation.

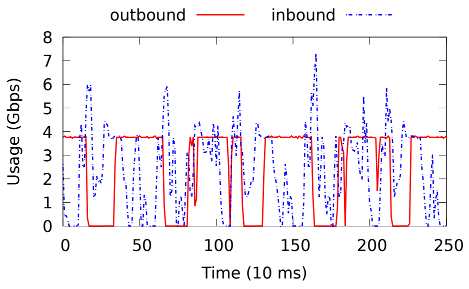
5.4 Network utilization



(a) ResNet-50 at 4Gbps



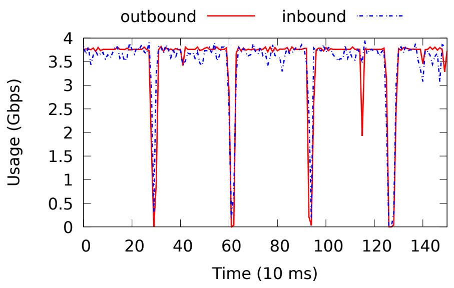
(b) VGG-19 at 15Gbps



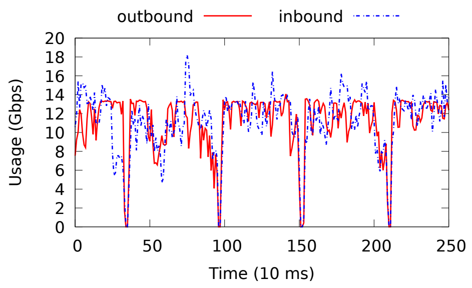
(c) Sockeye at 4Gbps

Figure 5.2: Network utilization of the baseline system

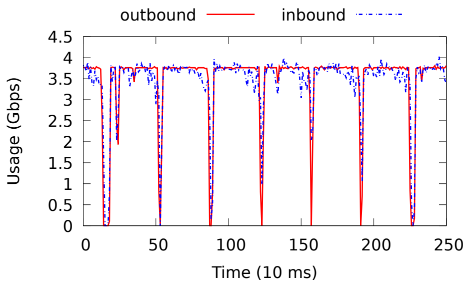
This experiment compares the network utilization of *P3* with the baseline sys-



(a) ResNet-50 at 4Gbps



(b) VGG-19 at 15Gbps



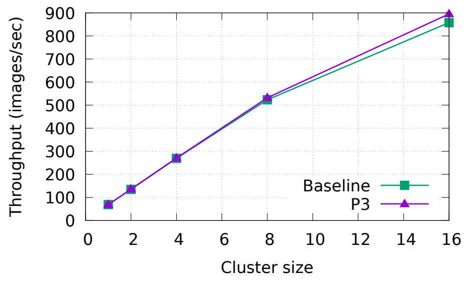
(c) Sockeye at 4Gbps

Figure 5.3: Network utilization of *P3*

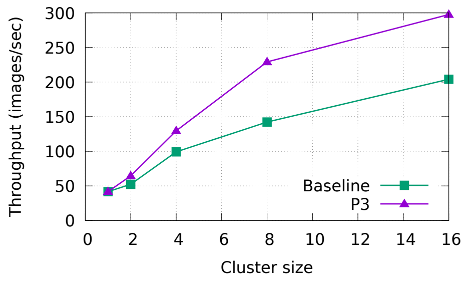
tem. We conduct this experiment for ResNet-50, VGG-19, and Sockeye and measure the traffic generated from and received by one of the four worker machines. The network utilization is measured at the interface level using Linux's *bwm-ng* tool [38] with 10 millisecond precision. Figure 5.2 shows the network utilization of the baseline system. The baseline implementation has bursty network traffic generated with regular peaks and crests across all models. This pattern is observed in other frameworks like TensorFlow [1] and Poseidon [41] as well (see Appendix A). For Sockeye and VGG-19, the network idle time is extremely dominant because of the presence of heavy layers. Moreover, the inbound and outbound traffics are not overlapped as the baseline fails to fully utilize bidirectional bandwidth.

In contrast, Figure 5.3 shows the network utilization graph with *P3*. We observe that *P3* significantly improves the network utilization compared to the baseline. In Figure 5.3a and 5.3b, the network idle time is seen to be reduced with *P3*. Especially for Sockeye in Figure 5.3c, *P3* utilizes bidirectional bandwidth more effectively than baseline system. This is one of the key explanations for the speedup observed in Sockeye despite having heavy initial layers.

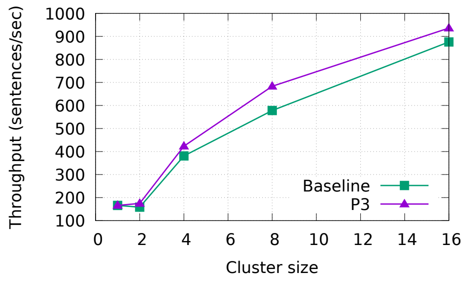
5.5 Scalability



(a) ResNet-50



(b) VGG-19



(c) Sockeye

Figure 5.4: Throughput scaling with different number of machines

We perform scalability analysis on ResNet-50, VGG-19, and Sockeye in order to show how well *P3* can perform on large clusters compared to the baseline system. We conduct this experiment by distributing models on AWS g3.4xlarge clusters of different sizes (2, 4, 8 and 16 machines) over a 10Gbps network.

Figure 5.4a shows that on ResNet-50 both the baseline and the *P3* achieve similar performance. As shown in Section 5.3, a 10Gbps network is more than enough to linearly scale ResNet-50. The throughput of VGG-19 has also been considerably improved with *P3* ; by as much as 61% on an eight machine cluster (Figure 5.4b).

Figure 5.4c shows the scalability of Sockeye. LSTM-based models like Sockeye are hard to scale over multiple machines, because of the heavy initial layers and difference in iteration time in worker machines due to the variable sequence length of input data. Nevertheless, with *P3*, we improve throughput of Sockeye by as much as 18% on an 8-node cluster.

5.6 Training accuracy

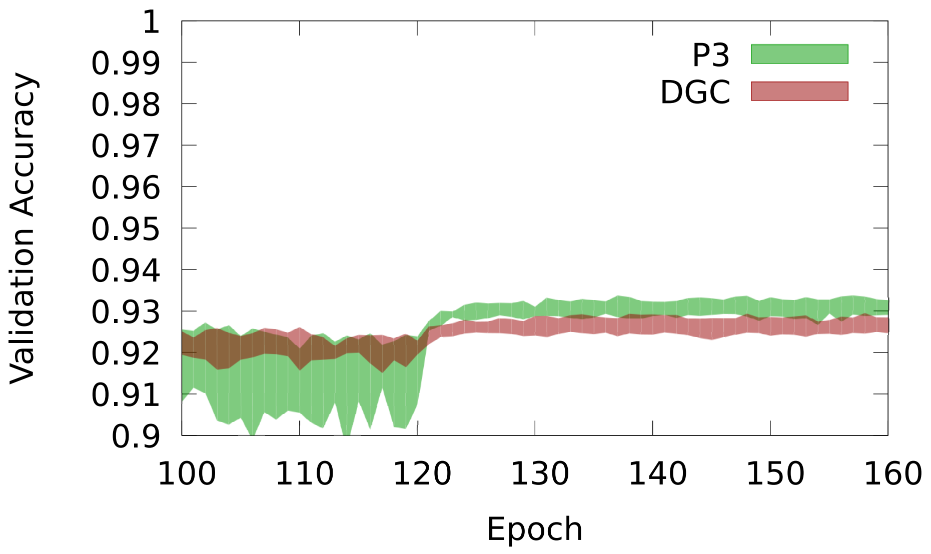


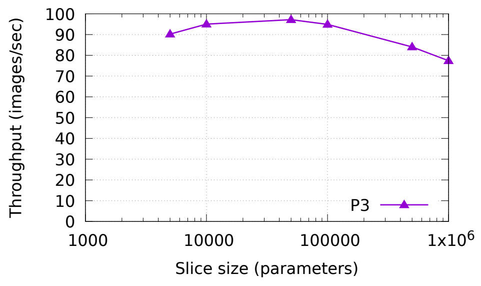
Figure 5.5: *P3* v.s. DGC

As we noted in Section 1, there are many compression techniques proposed for improving data parallel training performance. These methods can provide higher performance gains compared to *P3*, however, at the cost of reduction in the model quality. In this section, we compare convergence accuracy of *P3* with the state-of-the-art compression technique called DGC [25]. DGC reduces the amount of data transferred by taking advantage of the sparsity in the gradient updates. The key idea is to synchronize only those parameters with top-k gradients and accumulate the rest locally. In this experiment, we use a sparsity threshold of 99.9% per layer based on the configurations used in the original paper [25].

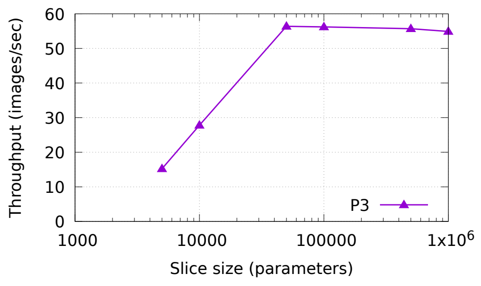
We trained ResNet-110 on the CIFAR-10 dataset for 160 epochs over a four-machine cluster with both *P3* and DGC using five different hyper-parameter settings. Figure 5.5 shows the validation accuracy range of *P3* and DGC from these experiments. The dark bands represent the gap between the worst and best accuracy on the five hyper-parameter setting. We observe that the final accuracy obtained with *P3* is always better than DGC. We calculate an average accuracy drop of 0.4% with DGC.

Unlike compression based mechanisms (like DGC), *P3* always communicates the full gradients with other worker machines and does not make any modification to the original SGD algorithm. As a result, the performance benefits from *P3* comes without any penalty on model accuracy.

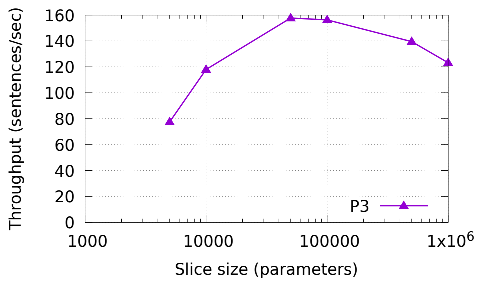
5.7 Parameter slice size selection



(a) ResNet-50



(b) VGG-19



(c) Sockeye

Figure 5.6: Granularity v.s. Throughput

As we show in Section 4, a small gradient packet size can improve the network utilization and, in turn, can improve overall training throughput. In this section, we show how the size of the parameter slice affects training performance. Figure 5.6 shows the throughput obtained for ResNet-50 and VGG-19 with *P3* on different parameter slice sizes.

Initially, the throughput increases as size decreases, and reaches its peak at $50,000$ parameters when it starts to drop. This happens because if the size is too small, the overhead of synchronizing packets at small granularity gets too high, overshadowing the benefits of parameter slicing. In all our experiments, we use a maximum granularity of $50,000$ parameters per slice as we found this to be optimal empirically.

Chapter 6

Related Work

In this thesis, we describe the key limitations in the data parallel deep learning distribution techniques used in popular ML frameworks (e.g., TensorFlow and MXNet), and propose solutions to mitigate these limitations by taking advantage of domain specific characteristics of deep learning models. To the best of our knowledge, this is the first work to summarize and address these limitations.

One notable prior work which proposes domain specific optimizations for data parallel deep learning workloads is Poseidon [41]. This work introduced the idea of Wait-Free-Back-Propagation (WFBP) which hides the communication overhead behind backpropagation by independently synchronizing individual layers in the neural network. We build upon this idea, and show that we can overlap computation with both forward and backward propagation. We further improve this idea by using parameter slicing that utilizes network bandwidth better.

Most of the recent papers in this area try to reduce communication overhead by sending fewer gradients. One popular method to reduce data transmission is gradient quantization (representing the gradient values using fewer bits). For example, 1-bit SGD [32] represents a 32-bit floating point gradient value in a single bit. Additionally, an error feedback is added to the SGD algorithm in order to account for the information loss that comes with the value approximation. 1-bit SGD can provide up to $10\times$ speed-up for traditional speech recognition applications. Another work called QSGD [5] proposes a family of compression schemes which balance the trade-off between the accuracy and gradient precision to provide good performance. Similar to QSGD, TernGrad [39] uses 3-level numerical compression to reduce data transfer in data parallel training. Both QSGD and TernGrad provide mathematical guarantees on the bounds of final model convergence accuracy. In contrast, *P3* always sends the full 32-bit parameter values.

Another approach is the sparse parameter synchronization. The idea is to synchronize only a few parameters on every iteration instead of the whole model. Gradient dropping method [2] only synchronizes parameters which have gradient values larger than a selected threshold. The threshold is calculated based on a fixed compression ratio. AdaComp [11] automatically tunes the compression ratio depending on the local gradient activity and achieves up to 200$\times$ compression.

All the above techniques make the trade-off between training performance and model accuracy because of the information loss introduced by value approximation or stale parameter updates [23]. *P3* , on the other hand, does not introduce any information loss since it always sends the full gradient matrix on every iteration.

On the other hand, a more recent work called DGC [25] offers up to 600$\times$ compression and around 5$\times$ speedup in low bandwidth networks while maintaining the same baseline accuracy on several DNN models. DGC use local gradient accumulation and momentum correction techniques to maintain the same accuracy. Even though the authors report no accuracy loss with DGC, there is no formal proof on the convergence guarantees cited in the paper. And as shown in Section 5.6, we find it difficult to reproduce their results despite substantial effort.[1] In our experiments, *P3* always gives better accuracy than the DGC. Based on these results, we conclude that our mechanism is a safer approach, as *P3* does not introduce information loss in the training algorithm and therefore there is no potential risk of accuracy loss. Moreover, our proposal is an orthogonal approach to the compression techniques and can be used on top of compression mechanisms to further improve performance.

A similar work to *P3* has been done and published independently by Hashemi et. al. [17]. Similar to *P3*, their proposed system called TicTac is also trying to reduce the communication overhead in data parallel DNN training by scheduling the parameter synchronization based on the predetermined priorities of layers. That

[1]This includes personal communication with the authors in order to get all their experiments correctly.

being said, there are a few fundamental differences in the proposed solutions. Tic-Tac is implemented on Tensorflow and they perform parameter synchronization at a coarse-grained level, specifically at a granularity of send-recv operations, whereas *P3* does the same at much finer granularity with parameter slicing optimization. Parameter slicing optimization helps *P3* to achieve better control over communication scheduling and better bandwidth utilization compared to TicTac. Secondly, TicTac uses static analysis on the DNN computation graph in order to prepare the optimal scheduling of the send/recv operations. This scheduling strategy is prepared offline and is used throughout the training. *P3* on the other hand performs the communication scheduling at runtime with the help of the producer-consumer mechanism communicating through a priority queue. We believe this immensely simplifies our design and also helps to deal with jitters in the network.

Chapter 7

Conclusion and Future Work

In this thesis, we analyze the data parallel distributed training methods used in current ML frameworks and observe that they fail to fully utilize available network bandwidth and induces high penalty on training performance under network bandwidth limitations. Based on this observation, we propose a new parameter synchronization method called *P3*, which improves the training performance by better utilizing the available network bandwidth. We implement *P3* on top of the state-of-the-art ML framework MXNet and demonstrate it to have higher resiliency towards bandwidth constraints and better scalability than the baseline MXNet implementation. With *P3*, we improve training throughput of ResNet-50 by as much as 25%, Sockeye by 38% and VGG-19 by 66%. We also have made the source code of *P3* publicly available.[1]

Currently, *P3* only supports distributed training on parameter servers. However, we believe that the optimizations proposed in this work are general enough to be applied to any gradient aggregation methods. Parameter synchronization using MPI operations are also used in practice by several ML frameworks (e.g. PyTorch [30]). We plan to extend *P3* to support *all_reduce* operations in the future.

We also plan to further improve *P3* in several ways. Currently *P3* uses predetermined priorities for the parameter slices by statically analyzing the DNN computation graph. This works fine for most of the DNNs as many ML frameworks represent DNNs as static computation graphs. We believe extending *P3* to per-

[1]https://github.com/anandj91/p3

form efficient communication scheduling for a generic data flow graph would be an interesting future work. Computation graphs with dynamic control flows might require more sophisticated communication prioritization strategy.

Another potential area of improvement would be on the estimation of the optimal parameter slice. Currently *P3* uses a common parameter slice size threshold for all the models which was obtained after an empirical analysis conducted on ResNet-50, Sockeye and VGG-19 described in Section 5.7. It would be worth investigating the generalizablity of such a common fixed threshold on a wide range of DNNs.

Bibliography

[1] 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, Savannah, GA, 2016. USENIX Association. ISBN 978-1-931971-33-1. URL https://www.usenix.org/conference/osdi16/technical-sessions/presentation/abadi. → pages 2, 7, 23

[2] A. F. Aji and K. Heafield. Sparse communication for distributed gradient descent. *CoRR*, abs/1704.05021, 2017. URL http://arxiv.org/abs/1704.05021. → pages 2, 29

[3] M. Alan, A. Panda, D. Bottini, L. Jian, P. Kumar, and S. Shenker. Network evolution for dnns. *SysML*, doc/182, 2018. URL http://www.sysml.cc/doc/182.pdf. → page 2

[4] Alexey N. Kuznetsov. Linux Traffic Control, 1999. URL https://linux.die.net/man/8/tc. → page 20

[5] D. Alistarh, D. Grubic, J. Li, R. Tomioka, and M. Vojnovic. Qsgd: Communication-efficient sgd via gradient quantization and encoding. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems 30*, pages 1709–1720. Curran Associates, Inc., 2017. → pages 2, 28

[6] Amazon. Amazon Web Services, 2019. URL https://aws.amazon.com/. → page 20

[7] D. Amodei, R. Anubhai, E. Battenberg, C. Case, J. Casper, B. Catanzaro, J. Chen, M. Chrzanowski, A. Coates, G. Diamos, E. Elsen, J. Engel, L. Fan,

C. Fougner, T. Han, A. Y. Hannun, B. Jun, P. LeGresley, L. Lin, S. Narang, A. Y. Ng, S. Ozair, R. Prenger, J. Raiman, S. Satheesh, D. Seetapun, S. Sengupta, Y. Wang, Z. Wang, C. Wang, B. Xiao, D. Yogatama, J. Zhan, and Z. Zhu. Deep speech 2: End-to-end speech recognition in english and mandarin. *CoRR*, abs/1512.02595, 2015. URL http://arxiv.org/abs/1512.02595. → page 1

[8] A. A. Awan, C. Chu, H. Subramoni, and D. K. Panda. Optimized broadcast for deep learning workloads on dense-gpu infiniband clusters: MPI or nccl? *CoRR*, abs/1707.09414, 2017. URL http://arxiv.org/abs/1707.09414. → page 9

[9] L. Bottou. Large-scale machine learning with stochastic gradient descent. In *in COMPSTAT*, 2010. → pages 2, 7

[10] M. Burtscher and P. Ratanaworabhan. Fpc: A high-speed compressor for double-precision floating-point data. *IEEE Trans. Comput.*, 58(1):18–31, Jan. 2009. ISSN 0018-9340. doi:10.1109/TC.2008.131. URL http://dx.doi.org/10.1109/TC.2008.131. → page 2

[11] C. Chen, J. Choi, D. Brand, A. Agrawal, W. Zhang, and K. Gopalakrishnan. Adacomp : Adaptive residual gradient compression for data-parallel distributed training. *CoRR*, abs/1712.02679, 2017. URL http://arxiv.org/abs/1712.02679. → page 29

[12] J. Chen, R. Monga, S. Bengio, and R. Józefowicz. Revisiting distributed synchronous SGD. *CoRR*, abs/1604.00981, 2016. URL http://arxiv.org/abs/1604.00981. → pages 1, 7

[13] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *CoRR*, abs/1512.01274, 2015. URL http://arxiv.org/abs/1512.01274. → pages 2, 4, 7

[14] D. Cunningham, B. Lane, and W. Lane. *Gigabit Ethernet Networking*. Macmillan Publishing Co., Inc., Indianapolis, IN, USA, 1999. ISBN 1578700620. → page 2

[15] J. Daily, A. Vishnu, C. Siegel, T. Warfel, and V. Amatya. Gossipgrad: Scalable deep learning using gossip communication based asynchronous gradient descent. *CoRR*, abs/1803.05880, 2018. URL http://arxiv.org/abs/1803.05880. → page 9

[16] ec2instances.info. Easy Amazon EC2 instance comparison, 2019. URL https://www.ec2instances.info/?region=us-west-2. → page 20

[17] S. H. Hashemi, S. A. Jyothi, and R. H. Campbell. Communication scheduling as a first-class citizen in distributed machine learning systems. *CoRR*, abs/1803.03288, 2018. URL http://arxiv.org/abs/1803.03288. → page 29

[18] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015. URL http://arxiv.org/abs/1512.03385. → pages 1, 5, 19

[19] F. Hieber, T. Domhan, M. Denkowski, D. Vilar, A. Sokolov, A. Clifton, and M. Post. Sockeye: A toolkit for neural machine translation. *CoRR*, abs/1712.05690, 2017. URL http://arxiv.org/abs/1712.05690. → pages 3, 5, 19

[20] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22Nd ACM International Conference on Multimedia*, MM '14, pages 675–678, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-3063-3. doi:10.1145/2647868.2654889. URL http://doi.acm.org/10.1145/2647868.2654889. → pages 2, 7

[21] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P.-l. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, pages 1–12, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-4892-8. doi:10.1145/3079856.3080246. URL http://doi.acm.org/10.1145/3079856.3080246. → page 1

[22] J. Keuper and F.-J. Preundt. Distributed training of deep neural networks: Theoretical and practical limits of parallel scalability. In *Proceedings of the Workshop on Machine Learning in High Performance Computing Environments*, MLHPC '16, pages 19–26, Piscataway, NJ, USA, 2016. IEEE Press. ISBN 978-1-5090-3882-4. doi:10.1109/MLHPC.2016.6. URL https://doi.org/10.1109/MLHPC.2016.6. → page 7

[23] S. Khoram and J. Li. DNN model compression under accuracy constraints, 2018. URL https://openreview.net/forum?id=By0ANxbRW. → pages 2, 29

[24] 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 *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 583–598, Berkeley, CA, USA, 2014. USENIX Association. ISBN 978-1-931971-16-4. URL http://dl.acm.org/citation.cfm?id=2685048.2685095. → pages 7, 16

[25] Y. Lin, S. Han, H. Mao, Y. Wang, and B. Dally. Deep gradient compression: Reducing the communication bandwidth for distributed training. In *International Conference on Learning Representations*, 2018. → pages 2, 4, 20, 25, 29

[26] L. Luo, J. Nelson, L. Ceze, A. Phanishayee, and A. Krishnamurthy. Parameter hub: a rack-scale parameter server for distributed deep neural network training. *CoRR*, abs/1805.07891, 2018. URL http://arxiv.org/abs/1805.07891. → page 2

[27] Mellanox. InfiniBand Cards, 2018. URL http://www.mellanox.com/page/infiniband_cards_overview. → page 20

[28] Nvidia. Nvidia Quadro P4000, 2017. URL https://www.pny.com/nvidia-quadro-p4000. → pages 1, 20

[29] Nvidia. cuda c programming guide, 2018. URL https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html. → pages 1, 2

[30] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer. Automatic differentiation in pytorch. 2017. → page 31

[31] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning representations by back-propagating errors. *Nature*, 323:533–, Oct. 1986. URL http://dx.doi.org/10.1038/323533a0. → page 7

[32] 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. → pages 2, 28

[33] T. Shanley. *Infiniband*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002. ISBN 0321117654. → page 2

[34] N. Shazeer, A. Mirhoseini, K. Maziarz, A. Davis, Q. V. Le, G. E. Hinton, and J. Dean. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. *CoRR*, abs/1701.06538, 2017. URL http://arxiv.org/abs/1701.06538. → page 2

[35] S. Shi and X. Chu. Performance modeling and evaluation of distributed deep learning frameworks on gpus. *CoRR*, abs/1711.05979, 2017. URL http://arxiv.org/abs/1711.05979. → page 8

[36] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. *CoRR*, abs/1409.1556, 2014. URL http://arxiv.org/abs/1409.1556. → pages 3, 5, 19

[37] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna. Rethinking the inception architecture for computer vision. *CoRR*, abs/1512.00567, 2015. URL http://arxiv.org/abs/1512.00567. → page 19

[38] Volker Gropp. Bandwidth Monitor NG (Next Generation), 2019. URL https://linux.die.net/man/1/bwm-ng. → page 23

[39] W. Wen, C. Xu, F. Yan, C. Wu, Y. Wang, Y. Chen, and H. Li. Terngrad: Ternary gradients to reduce communication in distributed deep learning. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems 30*, pages 1509–1519. Curran Associates, Inc., 2017. → pages 2, 29

[40] 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, ukasz 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. URL http://arxiv.org/abs/1609.08144. → page 1

[41] 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 *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 181–193, Santa Clara, CA, 2017. USENIX Association. ISBN 978-1-931971-38-6. → pages 4, 23, 28

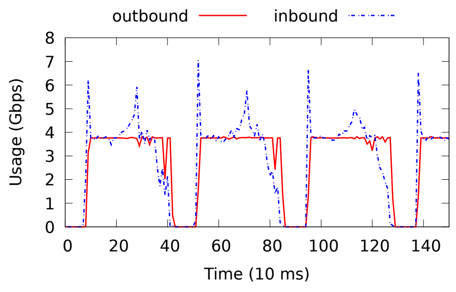
[42] H. Zhu, M. Akrout, B. Zheng, A. Pelegris, A. Phanishayee, B. Schroeder, and G. Pekhimenko. TBD: benchmarking and analyzing deep neural network training. *CoRR*, abs/1803.06905, 2018. URL http://arxiv.org/abs/1803.06905. → pages 2, 8

Appendix A

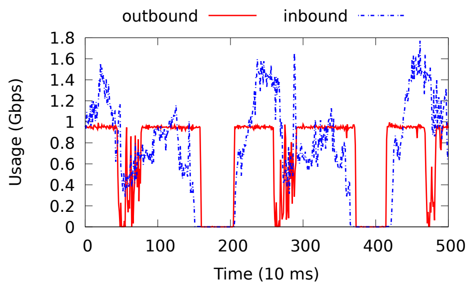
Additional Results

A.1 Comparison with other frameworks

In this section, we show that the limitations described in Section 3 exist in other frameworks as well.



(a) ResNet-50 on TensorFlow at 4Gbps



(b) InceptionV3 on Poseidon at 1Gbps

Figure A.1: Network utilization

Figures A.1a and A.1b show the network utilization measurements of TensorFlow and Poseidon taken while training ResNet-50 and InceptionV3 respectively on a 4-node cluster. Similar to MXNet, these frameworks also utilize network poorly even under bandwidth constraints.

A.2 P3 vs. Asynchronous SGD

ASGD algorithm does not perform synchronous update at the parameter server which means each worker machine is only blocked by its on parameter updates in an iteration as opposed to waiting for all the participating workers to finish. ASGD algorithm runs faster than synchronized SGD, however, at the cost of reduced convergence rate because of the stale parameter updates.

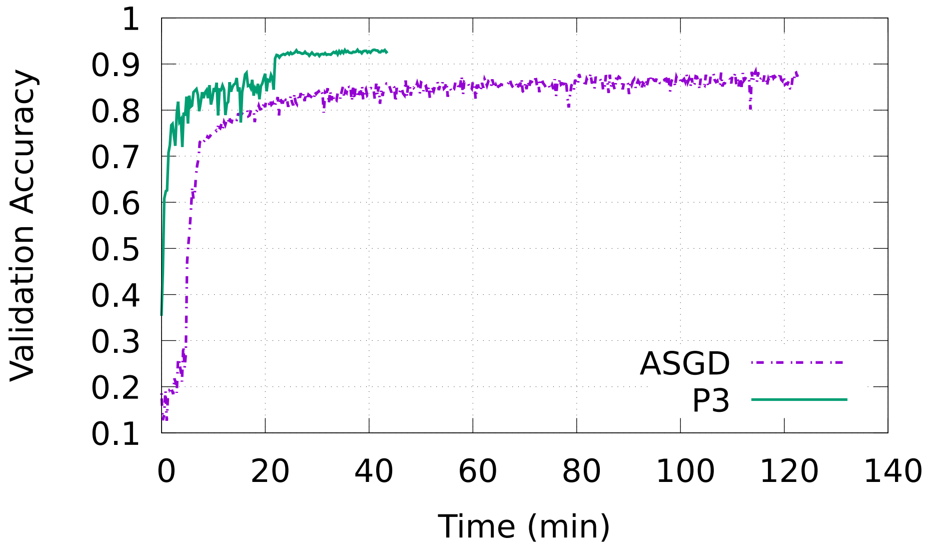


Figure A.2: ASGD v.s. P3

We measured the accuracy of ResNet-110 on CIFAR-10 on a 4-machine cluster and 1Gbps network with both *P3* and ASGD. *P3* reaches a final top-1 accuracy of 93% whereas for ASGD, it is only 88%. Additionally, *P3* is able to achieve 80% accuracy roughly 6× faster than ASGD.