Imperial College London

MEng Individual Project

Imperial College London

Department of Computing

Kungfu: A Novel Distributed Training System for TensorFlow using Flexible Synchronisation

Author:
Andrei-Octavian Brabete

Supervisor:
Dr. Peter Pietzuch

Second Marker:
Dr. Daphné Tuncer

June 17, 2019

Abstract

Modern times are distinguished as the Renaissance of Deep Learning, with remarkable advances across a wide range of domains and abundant research constantly pushing its boundaries. As models and datasets increase in size, people move to distributed clusters, which pose new challenges to training. Distributed training systems usually use the Parallel Stochastic Gradient Descent algorithm to scale out training. This algorithm creates large network traffic and an intensive need for synchronising the entire system. The fundamental limitation is that they inevitably explode the batch size and force the user to use large batches for training, in order to reduce communication traffic and the intensive demands for synchronising the entire system. Existing systems tend to adopt high-end specialised hardware and network such as InfiniBand, which eventually fail because of large network traffic and large clusters reaching hundreds of nodes. The hardware solutions are very expensive, do not scale and fundamentally, the system will suffer from large batch training, so the user may not be able to converge training when scaling out. In this project, I aim to design a system for Deep Learning that enables flexible synchronisation to address communication and large batch training solutions. The system brings three new designs: it has an abstraction that allows the user to declare flexible types of synchronisation, much more complex than Parallel SGD, it has a high-performance communication system where workers exchange gradients and models to collaborate for training and it enables monitoring for network and training statistics, providing support for dynamic adaptation of synchronisation strategies. I develop two advanced synchronisation algorithms based on this new system. One algorithm can exchange partial gradients if the network experiences bottlenecks, while the other enables each worker to selectively synchronise with other peers, reducing the global synchronisation overhead. These algorithms can further benefit from monitoring and adaptation to improve convergence and performance. I run large-scale test-bed experiments in commodity cloud to evaluate the effectiveness and performance of the proposed systems and algorithms. Experimental results show that the new system can converge to state-of-the-art accuracies while improving training performance by up to 22.5 times.

Acknowledgements

My work was possible, firstly, because of my parents who gave me full support in every decision I made along my way. They helped me build a strong foundation of human value, which will always be reflected in my pursuits.

Secondly, I would like to thank my supervisor, Dr. Peter Pietzuch for the valuable meetings which inspired me to develop a more clear vision of the system and also to grow as an engineer and improve my critical thinking. Also, I would like to show my appreciation for the high-end infrastructure which I was able to use for my experiments.

Special thanks go to my second marker, Dr. Daphné Tuncer, for the valuable feedback at the beginning of my work and for the interesting discussion on Networking.

Special thanks go to Luo Mai and Guo Li for collaboration on the larger Kungfu project and for the effective supervision during my project. Our discussions were unique learning experiences which helped me improve in all aspects of my practice and shaped my vision for developing Deep Learning systems.

Special thanks go to Alexandros Koliousis, for the valuable discussions about training algorithms, models and synchronisation and for inspiring me to become better at running Deep Learning experiments.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Current times are characterized as the Renaissance of Deep Learning (DL) [7]. With the recent advent of DL Techniques powering most recent breakthroughs in Intelligent Games, Visual Perception, Self-driving cars, Speech and Language, Art and Creativity [7], the field has experienced a rise in popularity proportional to the scale of achievements. Biologically inspired [8] to model human perception, DL can parallel or even exceed human performance on complex tasks. The impressive achievement of winning the ImageNet competition [9] [10] in object recognition using Deep Neural Networks in 2012 has later been superseeded by the success of the ResNet [11] architecture (2015). The revolution of Generative Advesarial Networks (GANs, 2014) [12] opens new research frontiers in Visual Perception, such as image-to-image translation [13] and face generation [14]. DeepMind's AlphaGo further proves how a computer program can defeat a world champion at the game of Go [15]. These achievements in DL constitute the beginning of an era of novelty and achievement. As computational complexity of models increases and data sets become large-scale collections. Big industry players such as Google, Uber, Baidu and Facebook are focusing their efforts [5, 16, 17] on improving user experience by using Deep Learning at scale.

A neural network is constituted as a collection of layers capable to represent relations between dataset inputs through a set of parameters called weights. The weights quantify non-linearities between specific features of the data and cause neurons to fire or activate when such features are recognised on newly fed data. After the pioneering of the perception [18] as the basic unit capable of learning, theorems have been developed for combining these units in order to achieve better learning capacity. One such theorem is the Universal Approximation Theorem [19], which states that stacking perceptions can approximate any continuous function. However, width would make it difficult to represent most complex relations between inputs in modern datasets. Therefore, the preferred organisation of neurons considers depth of architecture. To reach good performance on tasks, the eventual solution is to increase depth of neural networks for effective learning on big datasets.

Deep neural networks (DNNs) are often trained using supervised learning through the use of *stochastic gradient descent* (SGD) [20]. Users provide a training and testing dataset that consists of labelled data samples. Samples are fed iteratively into a neural network which makes a prediction. The prediction is compared against the label and then the loss is produced (i.e., the distance between the prediction and label). The aim of the training is to minimise the loss. Based on the loss, the training algorithm produces gradients to correct the weights of a neural network layer-by-layer using back-propagation [21]. Today's datasets often contain millions (ImageNet [22]) or even billions of data samples (Click Dataset [23]), to amortise per-sample training cost, people often adopt *mini-batch SGD* [24]. Data samples are grouped into batches and fed into the neural networks in an iterative manner. Gradients are produced and applied on a per-batch basis.

More recently, practitioners have paid substantial attention to accelerate training through distributed training [25, 5]. One such motivation is constituted by modern AI-backed systems with continuous ingestion of data, for which periodic training is critical for up to date inference capabilities (e.g., TensorFlow Federated, Smart Mobile Keyboards Training [26]). The problem demands leveraging the resources of multiple machines and the fast interconnects that are usually present in cluster setups for solving distributed optimization, which is a prerequisite mathematical problem

for large-scale learning tasks [16]. Distributed training concerns multiple replicas which train models in parallel on distinct shards of the entire dataset. A key operation in distributed training is to *synchronise* the replicas after each iteration (i.e., batch) of training. This enables replicas to incorporate more meaningful weight updates learnt by other replicas on their distinct shard, and thus ultimately accelerate convergence [27]. In these days, people usually use the *parallel SGD* (P-SGD) algorithm to synchronise replicas. This algorithm adopts parallel training workers to distribute the cost of computing gradients. At the end of an iteration, all workers compute gradients locally, compute a global average model and then move to the next iteration with the synchronous average model. P-SGD has been implemented as the de-facto synchronisation approach by most, if not all, DL platforms such as TensorFlow [28], PyTorch [29], Caffee [30], CNTK [31] and Keras [32].

1.2 Problem statement

Scaling-out training, however, often incurs a non-trivial communication bottleneck [33, 16, 34]. This is often due to the high communication requirement incurred by supporting all workers to frequently synchronise large DL models in a commodity infrastructure. There are two major issues resulting in this bottleneck:

- *Large network bandwidth consumption.* DL systems often run over data centre networks where network bandwidth is restricted. Providing full bi-section bandwidth (e.g., 40Gbps) in data centres is prohibitively expensive [4]. As a result, data centre providers often over-subscribe networks, i.e., a typical over-subscription ratio is between 1:4 and 1:16 [4]. Synchronising DL models in over-subscribed networks is particularly challenging. The state-of-the-art DL model like ResNet-152 [11] contains 60,344,232 parameters [35], summing up to hundreds of mega-bytes (e.g., 244 mega-bytes [35]) of data required by each synchronisation operation. The bandwidth challenge becomes more prominent as the performance of a GPU is quickly evolving. For example, the contemporary V100 GPU [36] can complete several and tens of mini-batches of training for a ResNet-50 model per second [37]. This boosts synchronisation frequency, and thus raises the bandwidth requirement substantially.
- *Expensive synchronisation barriers.* It is also expensive to maintain system-wide synchronisation barriers for a DL training job. Every iteration, coordinating all training workers to exchanging gradients not only creates challenging all-to-one and one-to-all communication patterns [16, 38, 33], making the system vulnerable to bandwidth shortage and TCP in-cast issues [34]. It also makes the system suffer from stragglers [39], failures [38], and competing tenants [40] which are common in commodity infrastructure (e.g., Huawei Cloud Public Shared P100 GPU Pool [41]). More importantly, the existence of this barrier makes the DL system difficult to leverage increasingly available cost-effective preemptive GPU resources [42, 43]. These resources offer 70% discounted price compared to reserved resources [44, 45]; however, they can be reclaimed at anytime with a short period of notice (e.g., 30 seconds). As the availability of resources becomes unpredictable, maintaining a strict global synchronisation barriers can bring the entire system to halt often. This is undesirable for time-consuming neural network training which can span days or even weeks.

1.3 Existing work

Existing systems attempt to address the above two issues by either improving networking performance or synchronisation efficiency. To improve networking performance, they resort to expensive specialised communication infrastructures such as InfiniBand [46] and NVlink [47]. These infrastructures provide low-latency and high-bandwidth; by default, they have limitations to run in large-scale cloud and Ethernet environments [48, 49] and they are predominantly deployed at small scale. To improve synchronisation efficiency, there are efforts like Horovod [5] and Baidu AllReduce [50] that leverage the high-performance collective communication systems such as Open-MPI [51], to speed up the averaging computation of gradients. There are also efforts that explore for asynchronously exchanging gradients [52], filtering out gradients [53] and compressing network traffic [54]. These efforts, however, only mitigate the issues of P-SGD. When the system scale becomes large, it eventually suffers from tremendous amount of synchronisation and from overheads of maintaining barriers [55].

More recently, there are new synchronisation algorithms being proposed to avoid communication bottlenecks while achieving the same training accuracy as P-SGD. At the high level, these algorithms explore three novel directions: (i) they relax synchronisation barriers by letting replicas to consolidate through a global average model in an *asynchronous* manner [56, 57]. This average model *corrects* local replicas instead of *replacing* replicas as in P-SGD. This allows training workers to evaluate more local minima through diverged model replicas, and thus achieve high training accuracy; (ii) they also let replica to synchronise with *selective* replicas, instead of all replicas as in P-SGD [58]. This reduces synchronisation traffic while preserving consolidation among replicas; and (iii) they let replicas to synchronise *partial* gradients with peers [33] when the network bandwidth is limited. Though promising, to the best of our knowledge, these emerging synchronisation algorithms are only available on proof-of-concept systems and not supported within any popular DL platforms such as TensorFlow, PyTorch and CNTK. These popular systems provide P-SGD as the only synchronisation option, often through dedicated systems like parameter servers [16] and all-reduce systems [5, 51, 50]. Implementing a new synchronisation algorithm requires largely modifying the existing systems, which makes it often infeasible for most DL system users.

1.4 My approaches and evaluation result

In this project, I explore new system designs that can enable flexible synchronisation algorithms for the de-facto deep learning platform: TensorFlow. My key idea is: *embedding synchronisation operators into the execution graph of a training worker to transparently intercept gradients and model variables and supporting user-defined policies to flexibly direct synchronisation among workers.*

Realising this idea, however, is not trivial due to three main limitations in TensorFlow: (i) TensorFlow requires developers to manually declare the dataflow including the operators to synchronise. To adopt different synchronisation algorithms, developers have to largely rewrite their training programs [59]; (ii) TensorFlow adopts dedicated, monolithic components such as parameter servers [16] to implement synchronisation operations. This not only limits the synchronisation policies user can specify (e.g., parameter servers only allow users to synchronise gradients not model variables), but also incurs cross-system communication overheads; (iii) TensorFlow provides very little, if any, support for helping developers understand the effectiveness of the underlying synchronisation algorithm, making it hard for developers to choose synchronisation tailored for their requirements.

To address above limitations, I propose and implement three novel designs for TensorFlow:

- *A high-level yet flexible synchronisation abstraction.* To enable flexible interception of gradient and model variable traffic, I extended the TensorFlow Optimizer [60] interface and develop a collection of *distributed optimisers*. These optimisers can be easily added to existing TensorFlow training programs (i.e., simply wrapping the existing training optimiser). Internally, users can easily configure the places to capture gradients and model variables at different timings of training. We show that such an abstraction is sufficient to support all the synchronisation algorithms we are aware of by far; while incurring negligible extra development overhead.

- *A high-performance communication system that provides diverse synchronisation primitives for parallel training workers.* I implemented a high-performance communication system that allow training workers to flexibly exchange the intercepted gradients or model variables. This communication system provides collective and point-to-point synchronisation primitives such as all-reduce and model request. The communication operations can also be easily configured to execute synchronously or asynchronously. This allows developers to explore different synchronisation configurations leading to various training accuracy and performance. Last but not least, this communication system has a peer-to-peer decentralised architecture, moving away from the conventional master-slave architecture adopted by most TensorFlow synchronisation systems. The communication system can thus support a large number of training devices, and even adopt the challenging preemptive GPU resource.

- *Novel monitoring and adaptation mechanisms for evaluating and controlling synchronisation algorithms online.* To help developers understand the effectiveness of synchronisation, I also implement rich monitoring and adaption support for synchronisation algorithms. In

particular, I provide online monitoring operators for *network statistics* and *training statistics*. These operators can provide sufficient insights to evaluate the *hardware* and *statistical* efficiency [57] of a synchronisation algorithm. Based on these monitoring data, I also implement an adaptation mechanism that helps developers to adjust the configuration of a synchronisation algorithm online.

I demonstrate the usage of the above novel components through two synchronisation algorithms: *partial gradient exchange* and *selective peer model averaging*. The former algorithm allows TensorFlow users to dynamically select the important gradients to synchronise every iteration when the network bandwidth is limited. The latter allows TensorFlow users to evaluate and choose which replicas to synchronise, which can significantly break down the synchronisation barrier and reduce synchronisation traffic. I evaluate the performance of these two algorithms on a Cloud test-bed. Experimental results show that they can improve the training time of state-of-the-art DL models, e.g., ResNet, by up to 40% in a high-performance DL cluster where the network is provisioned with InfiniBand [46]. In a more realistic Cloud network setting where network bandwidth is oversubscribed, they can improve the training throughput by up to 22.5x. Further, we enable the partial gradient exchange algorithm to leverage the online monitoring and adaptation primitives to improve training accuracy. By making informed online decisions to switch between P-SGD and partial gradient exchanging, the training time of a ResNet-50 job can be reduced by 2.9 hours while still converging to the state-of-the-art accuracy.

1.5 Contributions

This thesis has four key contributions:

- Designed a high-level yet flexible synchronisation abstraction to help TensorFlow users declare diverse synchronisation policies with minimal changes to their training programs.
- Implemented a high-performance communication system that can be naturally embedded within existing TensorFlow worker implementation; while providing novel collective and peer-to-peer synchronisation primitives that are key to realise emerging synchronisation algorithms.
- Proposed novel monitoring and adaptation mechanisms to help evaluate and control synchronisation algorithms online.
- Demonstrated the usage of the proposed synchronisation systems using two practical synchronisation algorithm and run large-scale test-bed experiments to verify the performance and effectiveness.

Chapter 2

Background and Motivation

This chapter presents the background and motivation of my thesis. It first provides the theoretical framework that enables training and explains the training components in TensorFlow. Subsequently, the transition to a distributed set-up is made through presenting the de-facto algorithms people use nowadays for distributed training. From these, we identify key synchronisation challenges and describe what it means to achieve flexible synchronisation. The perspective is then focused on existing systems where we identify a gap to be covered by a new system design for supporting flexible synchronisation.

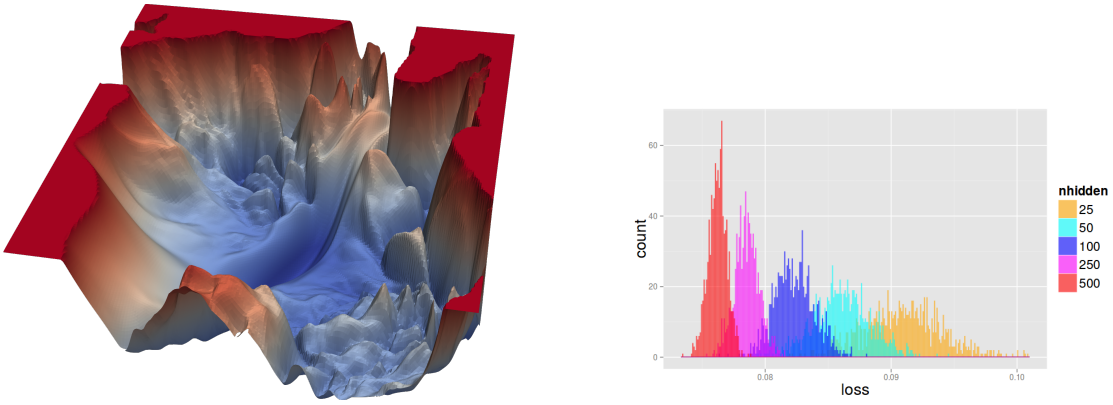
2.1 Deep learning

This section provides an insight into Deep Neural Networks complemented by a presentation of the mathematical background essential for understanding how training works.

2.1.1 Deep neural networks

Since the breakthrough of the multi-layer perceptron and the back-propagation algorithm in 1986 [21], the class of problems which can be solved using neural networks becomes larger and the problem specifications more complex. A Neural Network Model is a collection of layers and the types of layers and their order in the Neural Network forms the architecture of the model. Each layer consists of weights, also denoted as parameters Thus, it becomes imperative to adopt new architectures that improve accuracy for these problems. This motivates more complex neural networks with more than two layers called deep neural networks, which are able to represent and learn nonlinear data features. But why is depth preferred to width in neural networks? The Universal Approximation Theorem [19] states that stacking perceptrons [18] in a single layer of the neural network can arbitrarily approximate any continuous function arbitrarily accurately. However, this has two drawbacks in practice: (1) the difficulty of working with large non-sparse matrices which leads impossibility to split the computation such that the resources are efficiently used and (2) the complexity of modern datasets requires the neural network to make more subtle correlations between complex features. As such, the preferred approach is to use Deep Neural Networks for the majority of learning tasks.

There are three basic types of neural networks: feed-forward, convolutional and recurrent networks. These are often interwoven to achieve better performance on various learning tasks. These are composed of layers with different functions. Fully-connected layers capture non-linearities in data and constitute a building block for Deep Neural Networks. Individual fully-connected layers are often the heaviest in terms of number of parameters or weights. Recurrent neural networks rely on retaining and forgetting information fed sequentially. Our focus is on Deep Convolutional Neural Networks for supervised learning, which are biologically inspired [8] to model human perception. This is realized through convolutions, which are designed to fire for the most prominent features of their inputs. Convolutions are common in Computer Vision tasks, where multiple layers create hierarchical activation maps that emphasize most relevant features of an image (edges, corners, shapes). Because of the wide applicability there are algorithms [61] that allow efficient computation of convolutions using simple matrix operations, benefiting from considerable hardware support for parallel computation (GPUs).



(a) (b)

Figure 2.1: (a) Loss Landscape of Neural Networks [1], (b) Empirical Loss in Deep Neural Networks [2]

In these days, one of the state-of-the-art Deep Neural Networks is ResNet [11] which has been successfully used for Computer Vision Tasks such as recognizing objects in Images. A notable achievement superseeding existing Deep Neural Network architectures is made by ResNet-152, winner of the 2015 ImageNet [10] competition involving recognizing 22,000 classes of objects in 14 million images. The building block of a ResNet model is the residual block , which constitutes a grouping of convolutional layers. The novelty brought by this family of models is constituted by the skip connections between residual blocks, which allows deeper layers to receive inputs from earlier layers. The models are parameterized by the number of residual blocks, such that the training difficulty and the complexity of the learning tasks are in direct correspondence with this number. For example, ResNet-32 is trained on CIFAR-10 [62] dataset consisting of 50000 training images and 10000 validation images and the more complex ResNet-50 is trained on the ImageNet [22] dataset, consisting of 1,281,167 training images and 50000 validation images. The latter models represent benchmark Convolutional Neural Networks worth exploring when designing a new system due to their practical applicability, robustness and the heavy computation incurred by their training.

2.1.2 Learning through gradient descent

The key mathematical concept in training is optimization for supervised learning, where inputs have known labels. Notable supervised learning applications are image classification, image segmentation, scene recognition, pose recognition and speech recognition [7]. The fundamental optimization problem is solved by an algorithm called Gradient Descent [20].

Stochastic gradient descent Deep Learning algorithms involve optimization of the generalization error characterized by a function called loss function, error function or cost function [63], terms that can be used interchangeably. The aim is to minimize the loss function with respect to the inputs, current set of parameters, hyper-parameters and target labels. There are two types of optimization, *convex* and *non-convex*, depending on the surface of the objective function. Deep Learning deals with non-convex optimization where functions may have complex surfaces similar to the illustration in Figure 2.1a using gradient-based approaches. In convex optimization, the function surface does not have negative curvature and the global minimum can always be found by descent algorithms. Non-convex function optimization does not guarantee global optimality, hence the minimum found is most often local [7].

Stochastic optimization for Deep Learning does *not always lead* to a unique solution and instead tries to find *good approximations* of the global solution. Choromanska et al., 2015 [2] have shown empirically (Figure 2.1b) that convergence to a minimum that is close to the global function minimum in the complex surface of the loss function commonly shown in Deep Neural Networks becomes less important as the number of layers increases. When the network becomes deeper, the distribution of the loss values becomes more concentrated. In most of the experiments, the optimization yields roughly the same minimum loss, but a different minimizer [2, 7]. This means that there is no unique solution in stochastic optimization for Deep Learning. However, finding

a good solution is hard and new techniques are constantly developed to improve the foundational algorithms presented in this section [64] [65] [66].

Gradient descent [67] is the technique used to minimize the loss function. This is an iterative algorithm used to seek an extreme point of a function. The aim of the problem is to find the minimum of a function given a current point in its range. Assuming a multi-dimensional space, the first step in minimization is computation of the derivative of the function with respect to its parameters, which gives the gradients. Gradients have two key properties: magnitude and direction. Gradient descent works by making steps of fixed size in the direction of the steepest descent, which is the direction of opposite sign to the computed gradients. This adjustment gives the update rule by which parameters of the function are adjusted. When a local minimum is reached the value of the parameters at that point give the optimal solution to the optimization problem. The formalisation of the algorithm is presented next.

Let $L_D : \Theta \rightarrow R^+$ be the loss function defined on the whole data set, where Θ is the set of parameters of the model. L is assumed to belong to the class of twice differentiable functions. The expression for L_D for training data $(x_i, y_i) \in X \times Y$, $i \in \{1 \dots N\}$, some parametric model from the hypothesis space H, $f_\theta \in H, f_\theta : X \rightarrow Y'$, and sample loss function $l : Y' \times Y \rightarrow R^+$, is:

$$\hat{L}_D(\theta) = \frac{1}{N} \sum_{i=1}^{N} l(f_\theta(x_i), y_i)[7]$$

The objective is to minimize $\hat{L}_D(\theta)$ with respect to the current set of parameters, that is to make a *change* d in θ which produces the biggest decrease in the value of the loss function $\hat{L}_D(\theta)$ [7]:

$$\begin{aligned} \mathbf{d} &= \arg\min_d \hat{L}(\theta + d) - \hat{L}(\theta) \; such \; that \; \|d\| = 1 \\ &\approx \arg\min_d \nabla \hat{L}(\theta)^T d \; such \; that \; \|d\| = 1 \end{aligned}$$

Coosing the l_2 metric ball yields:

$$\begin{aligned} \mathbf{d} &= \arg\min_d \nabla \hat{L}(\theta)^T d \; such \; that \; \|d\|_2 = 1 \\ &= -\nabla \hat{L}(\theta) \end{aligned}$$

After a pass through all data points in D, the following weight update rule is applied: $w_{t+1} \leftarrow w_t - \eta \nabla \hat{L}_D(\theta)$, where η represents the learning rate and $\nabla \hat{L}_D(\theta)$ is the gradient of the loss function with respect to the parameters of the model after a full pass through the dataset. The algorithm continues until convergence or for a pre-defined number of iterations. In Machine Learning, this is considered a **batch** optimization method, where the term batch refers to *the whole data set*.

Mini-batch stochastic gradient descent Iterating through D is expensive for modern Deep Learning algorithms given the scale of data sets. A more efficient approach is to use **stochastic** optimization techniques. At the opposite end of the spectrum lies the Stochastic Gradient Descent [20] which is an online method that approximates the true gradient by drawing one random sample from the data set [63]. Convergence is achieved by multiple passes over the data set. In Deep Learning, the compromise method is the mini-batch Stochastic Gradient Descent [24], which draws a random sample set (with replacement) from the independent and identically distributed data points and computes an unbiased estimator for the true gradient at each iteration [63]. The mini-batch size is usually a power of two. Complete processing of all mini-batches constitutes an epoch.

Mini-batch Stochastic Gradient Descent can be formalised using the following expression for a setting where the mini-batch size is b and the set of inputs in a mini-batch is denoted by B:

$$w_t \leftarrow w_{t-1} - \frac{\eta}{b} \sum_{i=1}^{b} \nabla_{x_i} l_B(f_\theta(x_i), y_i)$$

where η is the learning rate and $l_B(f_\theta(x_i), y_i)$ is the individual loss computed on one input in a mini-batch.

Key hyper-parameters One notable aspect is that the true loss function is unknown to the optimization problem. Instead, training approximates the value of the loss function given the data. This is one of the main reasons why training is a difficult optimization problem, but it can be conquered by using a large data set and careful hyper-parameter tuning.

Back-propagation In supervised learning, training data are fed to the network and their known labels used to quantify the loss of the current model, also known as forward pass [7]. These are updated during the training process using computed gradients. The aim is to minimize the loss function with respect to the parameters of the model using the backpropagation algorithm [21], which computes a gradient correction for all model parameters, also known as backward pass [7]. This involves repeated application of the chain rule in differentiation in order to compute the gradients. The correction intuitively represents new experience to be incorporated such that the model improves on the supervised learning task.

2.2 Deep learning systems

DNN training requires specialised systems that meet the computational needs of training and provide appropriate hardware support for efficient computation. The fundamental element in neural network training is a *tensor*, which represents a multi-dimensional array of numbers. Training relies on tensor operations for the forward and backward pass, which are the most computationally intensive operations. All layer functions reduce to basic matrix operations [61]. DL systems are necessary because they provide mathematical optimization through *vectorization* for layer computations and necessary compilation and hardware support (Intel Avx [68], XLA (Accelerated Linear Algebra) [69]). More specialised, heterogeneous hardware has been developed for heavy-weight computation required to train Deep Learning models, such as GPUs [36, 70] and TPUs [28] (Tensor Processing Units). GPUs (Graphics Processing Units) enable stream computations [71] i.e., where the same operation is applied for sequential independent data inputs efficiently and in parallel. GPUs provide high-bandwidth dedicated memory and support for floating-point computation [71]. These heterogeneous devices provide computations which are orders of magnitude faster than CPUs [72]. Conventional machine learning systems such as Spark MLlib [73], however, are designed for using CPUs. Hence, practitioners must design new DL systems that can exploit heterogeneous devices for efficient DNN training.

TensorFlow [74] is one of the most popular systems for large-scale machine learning. It was designed to serve research purposes and as production-level infrastructure for training and inference at Google and to bring a novel perspective on the dataflow approach. This high-level programming paradigm has been previously adopted by other systems such as MapReduce [75] and Theano [76]. Fundamentally, such a system consists of a high-level API where users can define the computations in a *driver program* and a runtime execution graph where the specified computations get executed on input data. However, TensorFlow proposes a new design approach [74] where: (1) the nodes in the execution graph called *operators* hold mutable data called *tensors* which can be transmitted from one node to another for sequential mutability using custom transmission media, (2) the system provides low-level mathematical primitives which enable granular definition of models and customisable training rules and (3) support for heterogeneous devices such as TPUs (TensorFlow Processing Units), GPUs, CPUs and other lighter-weight mobile devices.

Next, we introduce some key concepts which are essential for understanding the TensorFlow paradigm. The runtime [3] presents a layered architecture (Figure 2.2) composed of: a *front-end* - high-level training and inference libraries built upon Python and C++ clients and a *back-end* - a master-slave system where the master is responsible for dispatching dataflow graph execution tasks taken over by follower services and realised on devices. The interesting feature of TensorFlow is device placement of computation which can be determined by the TensorFlow runtime or specified by the user. The library already provides a comprehensive collection of operators which are sufficient for model specification, prediction and inference, but to be able to develop a system overlayed on the existing infrastructure, the user can define operators with custom behaviour, which can be run on CPUs, GPUs or even TPUs. The user can thus leverage the performance of hardware acceleration.

The focus is on the piece of computation executed on a device. This is called *kernel* and includes the logic of a TensorFlow operation. Execution takes place in a session, after the graph is

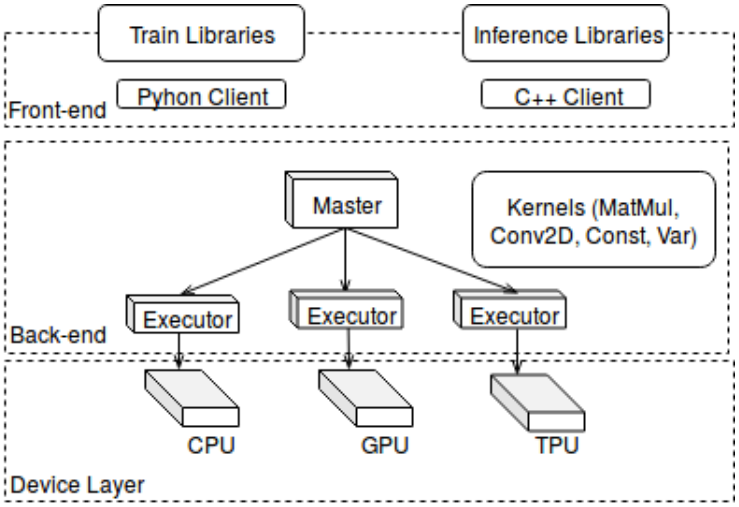


Figure 2.2: TensorFlow architecture overview [3]

constructed and scheduling decisions are automatically made by the back-end, but the system also allows the user to specify order constraints on specific operators using *dependency control blocks* (e.g., Listing 2.1, line 12). The separation between dataflow construction and the execution entry point must be explicitly made in code. A tensor is a multi-dimensional array and is the basic unit of computation in TensorFlow. Operations use tensors as inputs and may return tensors as outputs. Variables are mutable tensors.

```
1  import tensorflow as tf
2
3  X = tf.constant(tf.ones([3, 3], dtype=tf.int64))
4  Y = tf.constant(tf.ones([3, 3], dtype=tf.int64))
5
6  Z = tf.Variable(tf.zeros([3, 3], dtype=tf.int64))
7
8  step           = tf.Variable(tf.zeros([], tf.int32))
9  increment_step = tf.assign_add(global_step, 1)
10
11 def create_assign_op_with_dependency():
12     with tf.control_dependencies([increment_step]]):
13          return tf.assign(Z, tf.matmul(X, Y))
14
15 assign_op = create_assign_op_with_dependency()
16
17 with tf.Session() as sess:
18      sess.run(tf.global_variables_initializer())
19
20      for i in range(10):
21         sess.run(assign_op)
```

Listing 2.1: Simple TensorFlow Driver Program

Figure 2.3 is the entry point in iterative computation using TensorFlow and represents conceptual translation of the driver program in Listing 2.1 into a computation graph, consisting of tensors (lines 3, 4, 6 and 8) - elementary units of computation and operators - responsible for tensor transformations or runtime checks. The program is designed to multiply matrices X and Y (line 13) and to place the result in Z ten times (line 20). Z is a variable (line 6) and X and Y are constants (line 3 - 4), which means that Z's tensor values can change. The user-land is aware of the number of steps required by this computation (10), but in order to keep track at runtime new state-keeping operators are necessary, such as the global step increment are (line 9). The dependency control block (line 12) ensures that the step is always incremented before the matrix computation is executed. The dependency are best visualised in the dataflow graph presented in Figure 2.3. This presents two confluent branches of computation that intersect in the *control_ dependency* block. All branches have as leaves tensor variables or tensor constants. All variables should be explicitly and globally initialized at the beginning of the program (line 18) and come with a default *init* operator responsible for initilisation (see dataflow graph, *init* operator belonging to variables Z and *step*). When the results of both branches are ready after possibly asynchronous multi-device computation, other operators beyond the dependency control will be

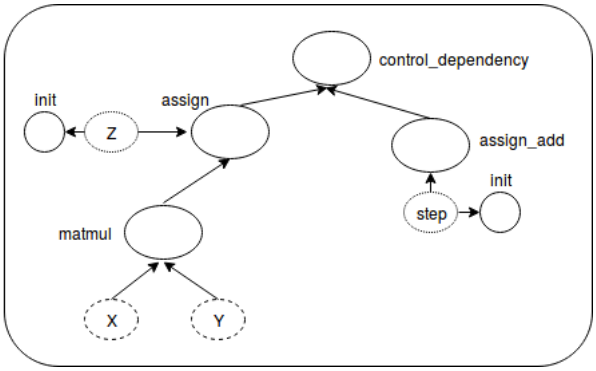


Figure 2.3: From dataflow specification to computation Graph

further executed.

Given that training neural networks is an iterative process, we further need to clarify how a neural network is represented. Layers of neural networks are TensorFlow variables representing multi-dimensional tensors. These are mutable during training according to the variable update rule based on gradients, which are tensors as well. Optimizers are responsible for computing gradients and for applying them to the model. The global step or iteration number indicates the number of batches processed. The stopping condition for training can be a specific accuracy which measures the generalisation power of the model or an epoch number in the case of well-known models, which can be derived from the global step given that the mini-batch and dataset sizes are known.

2.3 Distributed deep learning

Deep Neural Networks are computationally intensive and the volume of data is large. Consider size of datasets such as ImageNet [22], which consists of 14 million images and SQuAD [77], which consists of over 100,000 answerable questions for NLP applications. Models such as ResNet-152 [11] can reach over 60 million parameters [35]. Given such large volume of data and the large number of trainable parameters, it could often take months to train a contemporary DL model (e.g., the state-of-the-art model Google BERT [78], on SQuAD dataset [77], could take *40 to 70 days to train even with 8 GPUs* [79]). Distributed training systems are built to leverage multiple node resources. Scaling out training comes as a natural solution when the resources available on one independent worker cannot deliver a model ready for inference within acceptable time limits. Instead, modern Cloud infrastructure provides clusters of nodes that range from commodity machines with modest interconnects to high-performance multi-GPU and network accelerated machines. Distributed systems for training offer flexible ways to support fast training on different types of clusters and enable the models to do meaningful updates which incorporate gradient or model information aggregated from other workers. These systems adopt different architectures for which multiple synchronisation schemes are employed.

Data parallelism To decrease training time, DL systems often scale out training through data parallelism. The data set is sharded across all workers and each worker runs the same computation on its assigned data partition. Workers or replicas are processes responsible for training their local model while synchronising with other workers to send or receive updates. After each training iteration completes, gradients are averaged and used by each model for a local update, which incorporates new information aggregated from all workers, which constitutes the synchronization stage. One alternative to data parallelism is model parallelism, which involves splitting the computation across workers, each being responsible for a subset of the task. The results at each worker enter an aggregation phase consisting of the exchange of updated parameter values such that a complete view of the model is available at each worker. For the purpose of a distributed Deep Learning system, which deals with large data sets, complex computation graphs and a potentially large number of worker machines, a data parallel approach is most suitable.

Algorithm 1: Stochastic Gradient Descent (SGD)

input : l_i training examples;
D dataset size;
T number of iterations;
η learning rate
w_0 initial model weights
output: w, set of updated weights updated after T iterations

for $i \in 1 \dots T$ **do**
  Draw $j \in \{1 \dots D\}$ uniformly at random;
  $w_t \leftarrow w_{t-1} - \eta \nabla_w l_j(x_j, w_{t-1})$;
end

Algorithm 2: Parallel Stochastic Gradient Descent (Parallel SGD)

input : l_i training examples;
D dataset size;
T number of iterations;
η learning rate;
w_0 initial model weights;
k number of workers
output: w, set of model weights updated using aggregated gradients after T iterations

for $i \in 1 \dots T$ **do**
  for $i \in 1 \dots k$ ***in parallel*** **do**
    $g_i \leftarrow SGD(\{l_1, \dots, l_D\}, 1, \eta)$
    Aggregate gradients from all machines $G = \frac{1}{k} \sum_{i=1}^{k} g_i$;
    $w_t \leftarrow w_{t-1} - \eta G$;
  end
end

Parallel mini-batch stochastic gradient descent A popular algorithm to achieve data parallel training is parallel stochastic gradient descent [38]. This algorithm is MapReduce-friendly [38] and relies on the estimation assumed by Stochastic Gradient Descent. Given a fixed number of iterations, the algorithm makes a random choice from the entire dataset at every iteration, makes a forward pass in the model followed by gradient computation (backward pass) and performs the weight update rule. The algorithm is presented formally [38] in Listing 1. When there are multiple workers in the system each iteration incorporates an *aggregated* update from all workers, computed using SGD for 1 iteration. The aggregation is the sum of all gradients computed locally by each worker. The formal representation in Listing 2 shows the local view at each worker. This means that theoretically, the algorithm involves all-to-all communication. The pseudo-code snippets adapted from [38] show how this algorithm works.

The whole data set can be naively replicated on all machines. Each machine sees a partition of the data from the k shards assigned to each machine. This constitutes a synchronous data parallel algorithm denoted as S-SGD or Parallel SGD and can be more generally represented by the following equation [6] for a setting where the local batch size is b:

$$w_t = w_{t-1} - \frac{\eta}{kb} \sum_{j=1}^{k} \sum_{x \in B_{n,j}} \nabla_w l(x, w_{t-1})$$

where k is the number of workers, b is the local batch size, $B_{n,j}$ is the local batch of worker with index n corresponding to worker j, x is a training example chosen from the current batch.

One notable achievement of the algorithm [38] is the decoupling between data set size and training time by exploiting approximations of the true gradient through averaging, producing an equivalent statistical estimation as Mini-Batch Stochastic Gradient Descent.

This concludes the section on fundamental principles of Distributed Training Systems. The motivation for scaling out training due to increasing size demands from models and datasets lead to new directions for parallelism and implicit adaptation of the baseline optimization algorithm

called Mini-batch Stochastic Gradient Descent to the multi-worker setting.

2.4 Synchronisation challenges

Running distributed DL training in data centres is, however, challenging. The main reason is that synchronisation among parallel training nodes produces a non-trivial communication requirement which is hard to be fulfilled by commodity hardware adopted by data centres. In the following, I present the popular communication infrastructure used by today's DL systems, and conduct a case study to analyse the impact of the communication bottleneck.

2.4.1 Communication infrastructure of deep learning systems

Data centre providers rely on a large pool of hardware resources which they lease in a diverse range of compute services [80, 81, 42, 43] for public usage. As the providers cannot use resources for a single task, they employ virtualisation techniques [82] - hardware virtualisation (VMs) and OS virtualisation (containers) - in order to safely isolate multi-tenant processes. Of interest to Distributed Deep Learning systems deployed in the Cloud is the underlying virtualisation technique and the quality of the network links, to quantify possible performance penalties incurred during communication.

Even with virtualisation enabled, machines need to communicate with each other to accomplish distributed computation via a network [4, 83, 34]. This is realized usually through Ethernet links. Network packets represent the smallest *semantically loaded* unit of communication. Inter-machine communication is realized through switches, often arranged in a tree topology [84], responsible for processing and redirecting packets from input ports to output ports, such that they reach the destination.

There are mainly four kinds of communication infrastructure available for a DL system. Virtual networks and Docker networks [85] are probably the most popular in the Cloud. They offer ease of deployability and support the DL system to run at large scales (i.e., hundreds of nodes) while charging modest prices. More recently, specialised communication infrastructures, such as InfiniBand [46] and NVLink [47], also has become available in public clouds. These infrastructures provide ultra low-latency and high-throughput, making it suitable to support synchronisation in DL systems. They however adopt non-Ethernet-friendly techniques, e.g., Quantised Congestion Notification (QCN) [86], for data transmission, making it unsuitable for large-scale Cloud deployment. Most importantly, they significantly increase the cost of training, making them only available to large industry players. In the following, I will present more details about these four kinds of networks.

Virtual networks When VM clusters are provisioned their networking stack is often controlled by software. VMs are often grouped in logical network partitions called Virtual Local Area Networks (VLANs) [87], which are software-defined. The overhead in virtual networks comes from: (1) the virtualization technology used for network I/O, which determines how packets are processed on the path VM-hypervisor-hardware and (2) software-based switching realised through virtual switches within hypervisors [87, 88]. This delegates the responsibility of communication to hypervisors, adding a new level of indirection. A common network virtualisation technique is SR-IOV, through which the NIC (Network Interface Card) is virtualised. This gives the illusion to each VM that it has full control over the NIC and communication is supported by direct hardware interaction, with no hypervisor involvement for packet processing (Figure 2.4a).

Docker networks With the wide adoption of Docker [89] containers as a light-weight way of OS virtualisation, it is important to consider their effect on performance. In some situations, communication between containers can be realised through InfiniBand [82], but this requires additional effort [90]. In most cases, containers communicate over an overlay network [85] which adds a new level of indirection to communication. Docker Engines handle the overlay routing and packet processing, inducing new performance costs (Figure 2.4b).

InfiniBand InfiniBand (IB) [46] is a high-performance networking fabric realised through RDMA-enabled [91] Network Interface Cards which allow bypass of OS kernel packet processing for in-

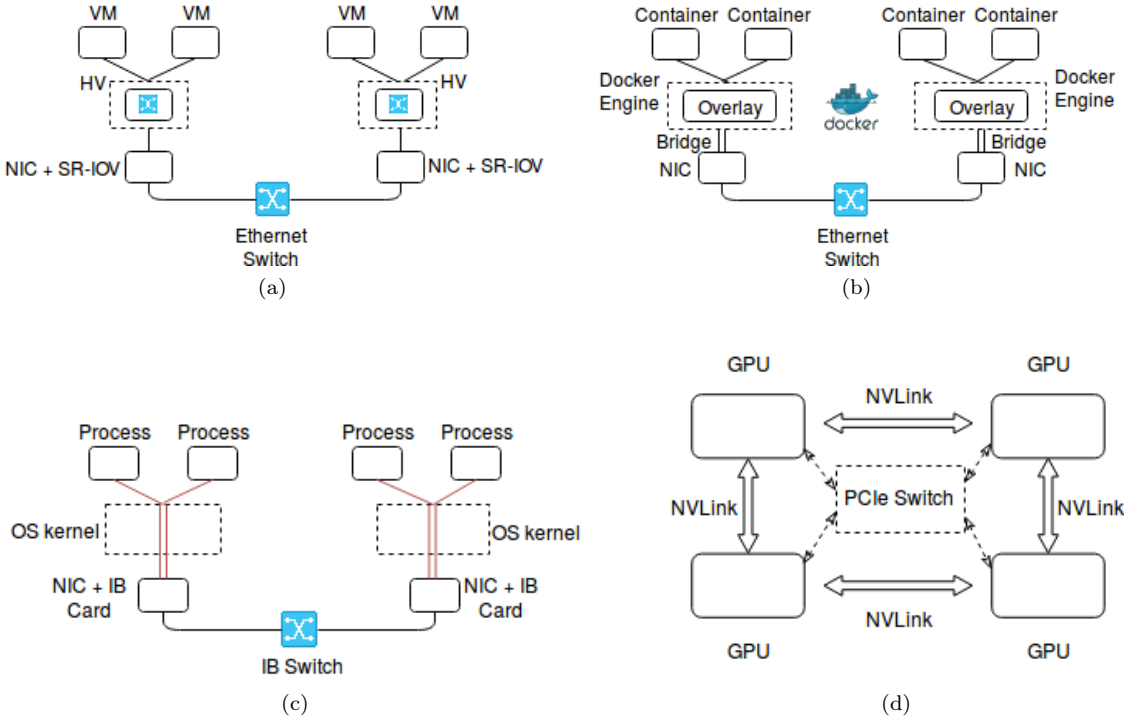


Figure 2.4: Four types of networks present in the Cloud: (a) Virtual network (b) Docker network (c) InifiniBand Network (d) NVLink GPU device communication

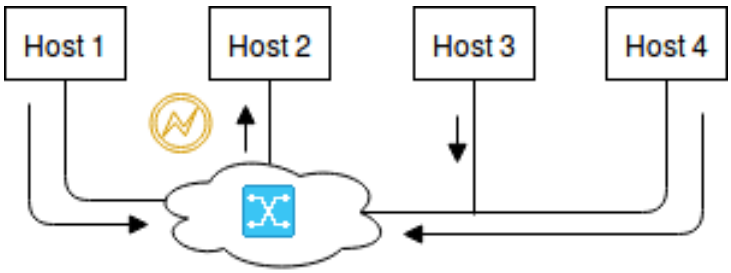


Figure 2.5: Over-subscription problem [4]

creased performance. The technology provides state-of-the-art communication capabilities in modern data centers with speeds up to 40 Gbps [92]. Speed comes at a very high cost both for customers and Cloud providers, especially when provisioning is done at scale. According to [93], the standard does not scale well in environments with heterogeneous clusters or hot-spots, because it relies on static routing which oftentimes requires manual configuration in software. Moreover, compatibility with virtualisation techniques [82] or types of Ethernet is limited as IB requires dedicated switches and Network Interface Cards (NICs) [94] Figure 2.4c.

NVLinks NVLinks [47] represent a high-performance communication fabric specific to GPUs. This can leverage both GPU-to-GPU communication and GPU-to-CPU communication (bidirectional), with speeds up to 25 Gbps. This networking technology is benefic to distributed training synchronisation, especially for *collective all-reduce* gradient aggregation, which can leverage dedicated collective communication libraries such as NCCL [95, 96]. The drawbacks of NVLink technology are: (1) **locality**, as NVLinks can only be leveraged on one server and (2) **cost**, because it requires extra hardware and the Cloud provider need to invest extra money to enable this feature (Figure 2.4d).

(a) (b)

Figure 2.6: (a) View of system barrier (b) Straggler issue

2.4.2 Scaling challenges

Large communication traffic Deep Neural Networks can reach parameter counts in the order of tens of million of parameters with sizes of a few hundred mega-bytes. For example, ResNet-152 [11] comprises 60,344,232 parameters [35], summing up to 244 mega-bytes [35]. We have previously seen that after each iteration, workers perform synchronously a global aggregation operation. The gradient count is the same as the parameter count, which means that the network has to support synchronisation for hundreds of mega-bytes for each worker. However, this is a problem in modern data centers where networks are over-subscribed. Hosts may not dispose of the necessary bandwidth to send the gradient data efficiently due to high over-subscription ratios of 1:4 or 1:16 [4]. This means that the link between a host and its switch may benefit from a small fraction of available bandwidth, as other hosts are sharing the inward switch link. When all other hosts send large amounts of data over the affected link (Figure 2.5), this becomes a significant communication bottleneck which incurs large delays unacceptable for high-performance training systems. The bandwidth usage becomes even a bigger issue as the performance of GPUs increase at a fast pace. For example, the state-of-the-art V100 GPU [36] can process tens of training mini-batches per second [37]. In such a case, it needs to synchronise several giga-bytes of data per second, aggravating the communication problem in Cloud environments.

Expensive system-wide execution barrier Even when the Cloud infrastructure provisions clusters with ultra-fast InfiniBand [46] links and efficient GPU training, there is still a naturally occurring system barrier due to synchronisation. Due to heterogeneity in task execution, workers are likely to complete their task at different times. Figure 2.6a shows four workers represented as arrows. Short arrows are tasks that finish *early*. The barrier is the explicit wait imposed on all workers of the system until all tasks are completed. The execution resumes in the next iteration, possibly with similar task completion behaviour. When all tasks finish early and have to wait for a single task to complete in order to move forward to the next iteration, the system suffers from the *straggler issue*. This is undesirable in training systems, but the occurrence of such behaviour is highly likely. Theoretically, Parallel SGD leverages the resources of multiple nodes and can benefit if each worker uses training acceleration hardware (GPUs, TPUs). The algorithm in its pure form does not employ any fault tolerance scheme [38] and suffers from the *straggler* issue, as it creates an explicit synchronisation barrier enforcing accumulation of gradients from all workers.The strawman solution of all-to-all communication does not scale [33], but this is also the case in more efficient approaches such as all-reduce. Hardware diversity in training systems is difficult to account for and systems that do not combat such issues by good design tend to experience significant slow-downs when explicit barriers are enforced. Oftentimes, training jobs run in the Cloud may use public resource pools, with links and machines not fully dedicated to the user. Dependence on the pool's resource utilisation makes training vulnerable to stragglers.

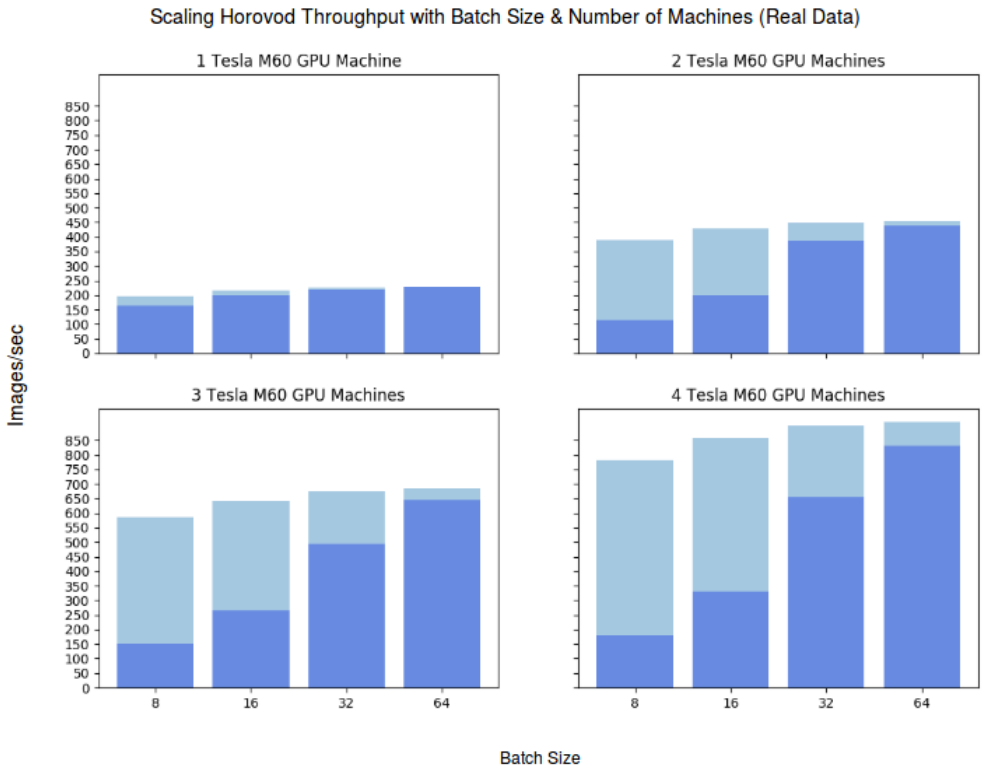


Figure 2.7: Scaling Horovod: Trade-off Number of Nodes-Batch Size, as compared with ideal throughput

2.4.3 Case study: synchronisation performance in cloud networks

In order to understand the impact of communication bottlenecks, we run distributed DL training jobs using a state-of-the-art synchronisation system Horovod in a public cloud: Microsoft Azure Cloud Computing Platform [97].

Experiment setup The experimental setup consists of a cluster of four virtual machines, each dedicated with four NVIDIA Tesla M60 GPU Machines which are connected by commodity network links and communicate over a virtual network. When VM clusters are provisioned their networking stack is often controlled by software. VMs are often grouped in logical network partitions called Virtual Local Area Networks (VLANs) [87], which are software-defined.

Horovod [5] implements the parallel SGD to achieve synchronisation through a high-performance OpenMPI network stack. To test the performance of Horovod, we measure the speed of training, also called throughput, which represents the rate of image ingestion per second. The experiments proceed as an investigation of how changes in parameters (number of nodes, batch size) affect training time. Distributed parallel mini-batch Stochastic Gradient Descent is a widely-used strategy which synchronises the gradients of all workers using all-reduce.

The reference Deep Neural Network used for experiments is ResNet-50 [11] which is trained on a subset of the ImageNet dataset [22], using the benchmarking code provided by the Horovod project [98]. In the following, a worker in the system denotes the training process run one one virtual machine using one GPU.

Experimental result To evaluate the scalability of Horovod, we increase the number of nodes and measure training throughput. The ideal throughput is computed by multiplying the throughput for a training run of an independent worker (1 GPU) by the total number of workers (GPUs) in the system. Figure 2.7 shows that the system is further from achieving the ideal throughput when the number of machines grows. When moving from one machine to two machines, the throughput increases by 50% for a small batch size of 8. The transition from two machines to three machines

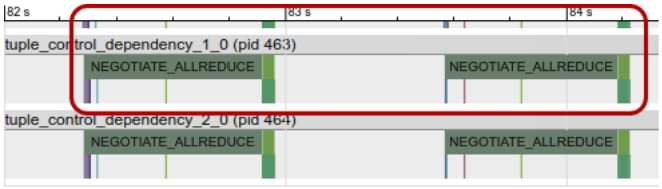


Figure 2.8: Horovod timeline showing frequent all-reduce barriers

shows an increase by 40%. When the scaling problem, becomes more difficult i.e., from three machines to four machines, the throughput only increases by 10%. This is a downward trend which becomes more and more nuanced with smaller batch sizes. For larger, batch sizes (64), when the number of machines is increased from one to two, there is a 1.25 throughput increase. From two to three machines, it increases by 44.4%. For more difficult scaling (from two to three machines), the throughput increase is only 30%. There exists a downward trend for larger batch sizes as well, but the system scales better. There is a trade-off between batch size and number of machines. Training with larger batch sizes provides higher throughput in the system, but this is *artificially* created. In addition to the number of machines, batch size is another key parameter that determines the interval of launching synchronisation. The smaller the batch size, the higher the frequency of synchronisation. We measure the training throughput of Horovod using batch sizes as 8, 16, 32, and 64 in a 16-GPU cluster (4 machines). When the batch size is 16, the throughput is 59.49% lower than ideal, when the batch size is 32, thoruhgput is 26.14% lower, while with a larger batch size (64) the throughput is only 13.87% further than ideal.

We also record the important synchronisation events in Horovod and try to identify the performance bottlenecks using the Horovod timeline. The trace study shows that training is blocked by frequent synchronisation barriers which reduce the GPU utilisation and increase training time. The effect can be seen in the sample timeline from Figure 2.8, presenting a snapshot of two synchronisation barriers. The all-reduce phase ($NEGOTIATE\_ALLREDUCE$) can last by up to half a second and the synchronisation barriers are much larger than training time (light green on Horovod timeline). This is due to the centralisation employed by Horovod, where one worker plays the role of master - all gradients are aggregated by this worker, which then broadcasts all-reduced gradients (Appendix A.1).

Takeaway The analysis of Horovod in public Clouds has two key takeaway messages: (i) Even though Azure networking has provided 25 Gbps connection among the 4-GPU servers, the network still appears to be a major bottleneck during scaling. This bottleneck would become even more serious when using contemporary GPUs like P100 [36] and V100 [70] as these GPUs can complete each mini-batch of training faster than the M-60 GPU; (ii) Horovod uses a master node to coordinate the execution barriers of all workers. Detailed trace study show that this master becomes a main scaling bottleneck even at a relatively small scale: 4 nodes (16 GPUs). For a detailed description of the protocol used by Horovod, Appendix A.1 can be consulted. The performance penalty of a system-wide barrier has become a dominating factor for communication bottleneck today as the use of hundreds of parallel GPUs is increasingly common in recent DL studies [27].

2.5 Limitations of existing synchronisation systems

Given the importance of synchronisation, we have observed a recent thrive of synchronisation systems proposed for DL systems such as TensorFlow. In principle, all these systems adopt a parameter server architecture [99, 16] or an all-reduce high-performance implementation [51, 27], and have full, dedicated support for parallel SGD.

2.5.1 Parameter servers

Figure 2.9 shows the typical communication patterns of a parameter-server-based synchronisation system. The system can run different applications concurrently, each handled by a separate worker. Workers are coordinated by master processes, which keep track of the global model. Each worker

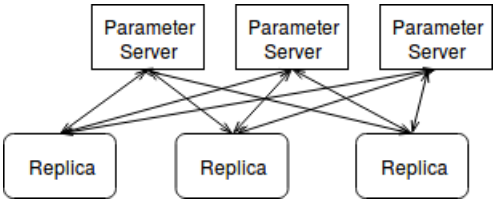


Figure 2.9: Parameter Server communication overview [5]

has access to a partition of the data used to compute gradient updates. Workers do not communicate with each other, but only with the parameter servers. They send gradient updates to their assigned master, which aggregates and applies them to the global model in a synchronous manner. The asynchronous parameter server scheme is preferred for large scale Deep Learning problems, because the number of updates is high [39] and the training throughput is significantly better than in the synchronous case. The authors of [39] show that staleness can significantly impact the test accuracy, thus Deep Learning scientists opting for such models must impose tight bounds to avoid model degradation.

The parameter server architecture has three key limitations in addressing the emerging synchronisation challenges:

- *Extra overhead of managing external synchronisation infrastructure.* To use parameter servers, DL users must manage external synchronisation infrastructure in addition to the training infrastructure. This requires extra mechanisms to manage failures and scale resource if need.

- *Non-trivial partitioning of synchronisation and training resource.* Given the sizes of DL models being trained and the various hyper-parameters, i.e., batch size, being used, the optimal partitioning of hardware resource between synchronisation (i.e., parameter servers) and training (i.e., training workers) can largely vary [100]. For example, the use of small batch sizes for training could significantly increase synchronisation frequency, and thus increase the demand for the number of parameter servers. However, pre-configuring such a number is non-trivial as DL users often try various settings of system parameters when tuning training accuracy, resulting in frequent reconfiguration of the parameter servers.

- *High performance overheads.* When using parameter servers for synchronisation, training workers need to copy gradients out of GPUs and and then move the average model back to GPU memory. If parameter servers are unfortunately placed at a different node, this gradient copy even have to go through network stacks, which can significantly increase synchronisation latency.

2.5.2 High-performance all-reduce-based systems

Motivated by the limitations of parameter servers, practitioners have recently developed new synchronisation systems like NCCL [96, 95], Horovod and Baidu AllReduce [5, 50]. These systems let workers exchange gradients directly instead of relying on external synchronisation components, thus removing the need for maintaining external infrastructure for synchronisation and partitioning resources. To efficiently perform the exchange, they often adopt high-performance implementation of the all-reduce operation which can efficiently compute the sum of all local gradients and broadcast the sum back to all workers. The worker themselves compute the average gradients and apply to their local model replicas. To ensure that each worker can start all-reduce at the correct timing, the all-reduce systems often come with a master node that coordinates the all-reduce operation (Figure 2.10).

Figure 2.10 also shows an efficient implementation called ring all-reduce [101, 5, 50]. The algorithm rationale relies on each node sending its gradients to the direct neighbour in its local spanning ring. The logical spanning ring is constructed such that values can be aggregated by successive send operations in the ring. The aggregated value eventually reaches the start node again via the unique inward edge. At this point, the node has efficiently aggregated all values from other nodes. To better understand the performance of ring all-reduce, we also analyse its bandwidth usage and the detailed analysis result is presented in Appendix A.2.

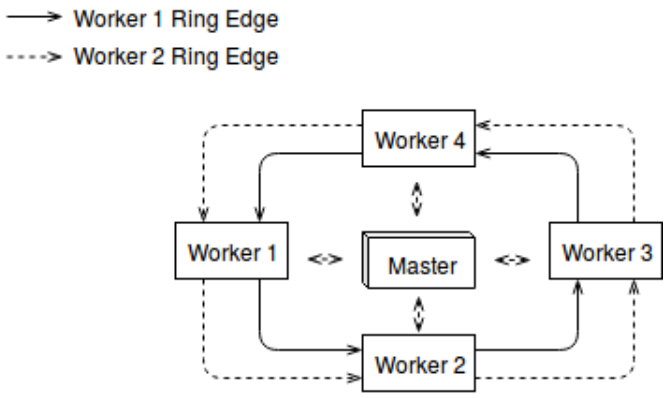


Figure 2.10: Typical all-reduce system architecture where a master coordinates parallel ring all-reduces

Though offering improved performance, all-reduce-based synchronisation systems eventually fail to resolve the emerging synchronisation challenges as well. These systems also have three key issues:

- *Insufficient scalability.* The all-reduce algorithm assumes the network to provide full-bisection bandwidth and thus it can achieve optimal bandwidth usage when exchanging the gradients. This assumption is, however, rarely valid in commodity data centres where the bandwidth in the network core is often over-subscribed. As a result, all-reduce systems can achieve the promised performance only with specialised communication hardware like InfiniBand (this is also verified by our previous case study for Horovod), making them an expensive solution that can be afforded only by large DL stack-holders.

- *High coordination overheads.* What further compromises the system scalability is the use of a master-slave architecture for coordinating all-reduce operations. The growing size of a DL system can significantly increase the coordination overheads, making the system vulnerable to single-node bottleneck.

- *Dedicated support for parallel SGD.* A fundamental limitation of all-reduce-based systems is their dedicated support for parallel SGD. The all-reduce operation has a restricted semantics and it is only suitable for computing aggregated metrics all training workers as in parallel SGD. However, parallel SGD suffers from its inevitable changes to batch size [27], thus enforcing users to use large batch sizes during training when using multiple GPUs. As a result, all-reduce-based synchronisation systems poorly support the DL models that are restricted to small batch training [102], making these systems a non-universal solution for synchronisation.

2.6 The need for supporting flexible synchronisation

In this project, we argue that: to fundamentally resolve the emerging synchronisation challenges, it is necessary to design a new distributed training system that can effectively support flexible synchronisation. These new synchronisation algorithms should achieve the same training accuracy as in parallel SGD while avoiding (i) exchanging a large amount of data for synchronisation and (ii) maintaining expensive system execution barrier. More importantly, the new training system shall tackle the large-batch training issue. This can be achieved by allowing DL users to implement new synchronisation algorithms, e.g., synchronous model averaging [57], that can keep batch size constant during scaling. Recent studies for DL system synchronisation have shown promising results. These studies can be broadly classified into two classes: studies that reduce synchronisation traffic by performing model averaging selectively [58] and studies that aim to relax the synchronisation barriers by allowing each worker to send subset of its gradients for aggregation [33]. These create a new synchronisation techniques where gradient or model information are partially exchanged with workers.

The key intuition for flexible synchronisation can be best understood through a simple example. Figure 2.11 presents a snapshot of the state of four workers at the end of a training iteration. They compute the gradients on the local model and they need to synchronise in order to compute the average gradient which has value 0.5. However, it is sufficient that only two replicas synchronise gradients to obtain a good estimation of the global average gradient. For example Workers 0 and 3 can exchange gradients, such that their aggregated gradients will become 0.5. Workers 1 and 2 will perform the weight update rule using the local gradients, 0.6 and 0.4 respectively. This approach naturally reduces the network traffic and relaxes the global synchronisation barrier.

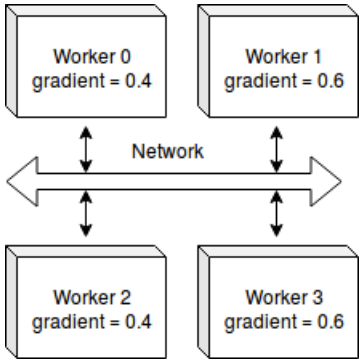


Figure 2.11: Partial Exchange Key Intuition

2.6.1 Synchronising partial gradients

One notable contribution is Ako [33], a novel decentralized training algorithm that relies on partial gradient exchange between workers that train local models. Homogeneity of worker roles eliminates the task-dependent resource allocation problem previously encountered in parameter server approaches. Ako achieves scalable decentralized synchronisation through partial gradient exchange and decoupled CPU & network use through independent network tasks that send accumulated gradients asynchronously. The theoretical guarantees of the Ako algorithm are: high hardware and statistical efficiency, network link staturation and a robust fault-tolerance scheme [33].

Partial gradient exchange is the technique by which workers can share a partition of their local gradients with other peers. Each worker partitions the full gradient update into p disjoint partitions, which are scheduled for sending in round robin fashion. After an iteration completes, one partition is broadcast to all other peers. All other $p-1$ partitions are saved and accumulated to the new gradient updates generated in subsequent rounds. Once other peers receive the partition, they update the corresponding weight parameters. It takes p synchronisation rounds to transmit the whole gradient at time t [33]. Ako is a novel algorithm that relaxes synchronisation barriers, proposing a scalable approach to Distributed Deep Learning.

Enabling flexible synchronisation strategies based on partial gradient exchange would attenuate the major limitations of parallel training. First, by exchanging a subset of the gradients, less network traffic is created, which can enable efficient training in multiple types of networks present in Cloud environments. The algorithm could leverage poorer network links found in virtual machine clusters or overlay networks in Docker containers at lower cost. The concern of large network traffic would be thus diminished. Reducing the network traffic naturally reduces the synchronisation barrier, which eliminates partially the training barrier bottleneck and any concerns of low hardware utilization, for example when the batch size is small and strong GPUs complete iterations at very fast rates.

2.6.2 Synchronising models among selective peers

One popular alternative synchronisation approach used in communication constrained environments, such as online learning in self-driving cars, is model averaging. It works by training models separately and exchanging parameters for synchronisation, rather than gradient updates. The improvement does not come from the amount of information transmitted, but rather from the

choice of communication frequency. This approach works for any model, as it treats the underlying algorithm as a black box. [103].

We have previously stated that Deep Learning problems aim to optimize non-convex functions. However, model averaging works only with convex functions, as any finite number of descents is guaranteed to find the global minimum, thus the average loss is expected to be less than the individual loss. In the non-convex case, the average model may have a worse performance than any composing model [103]. The solution for this problem can be found by starting from periodic model averaging, where model parameters are exchanged at fixed time intervals. Redundant communication can be eliminated by considering utility of communication. Kamp et al. [104] devised working strategies for dynamic model averaging conditional on the communication gain between workers: when most of the losses are large, the system should synchronise, expecting to minimize the overall loss on the average model, but when only a small fraction of workers have large losses, no synchronisation occurs. [103]. A similar approach is taken by the Crossbow system [57], which uses a modified approach of model averaging called Synchronous Model Averaging, designed to run on GPUs. The principle is to use independent tasks which train the model and calculate corrections to be applied in batches to a Central Average Model. The expectation is that replicas reach consensus on the optimality of the average model and eventually reach the same minima [57]. An important aspect that ensures convergence is that the gradient-based optimization takes into consideration gradient direction consistency with past updates.

Deep Learning scientists have moved away from asynchronous synchronisation strategies because they affect convergence. A disruptive algorithm proposed by Zhang, et al. [58] brings together the advantages of synchronous and asynchronous Parallel SGD, creating an algorithm which is robust in heterogeneous environments, does not affect convergence and benefits training speed using asynchronous communication. This algorithm creates a visible shift to decentralized training in which workers directly communicate with each other in a peer-to-peer fashion and where there is no global synchronisation barrier.

2.7 Summary

In this chapter, we have presented how DL training works and we have set-up the grounds for TensorFlow program development. Then we introduced distributed DL through data-parallelism as a way to tackle dataset size increase and model size increase and emphasized the de-facto algorithmic approach to distributed training, which is Parallel SGD or S-SGD. We have identified that de-facto synchronisation algorithms such as S-SGD incur naturally occurring (theoretical) penalties at the level of *large network traffic* and *large synchronisation barriers*. Then we shifted focus to real cluster set-ups and described how they are realised in modern Cloud environments (interconnects, virtualisation) and provided a case study of the performance of Horovod [5] for a commodity VM cluster provisioned by Microsoft Azure, in order to identify the bottlencecks, concluding that running Horovod's version of all-reduce (based on MPI communication stack [105, 51]) in such environment does not scale. An in-depth look at how systems realise synchronisation (Parameter Servers, MPI-based approaches) and confirm that they have not reached a good maturity level for flexible synchronisation. Developing partial synchronisation strategies within the TensorFlow system is particularly challenging because it does not provide sufficient system abstraction for distributed clusters to determine which peer to communicate with for partial model synchronisation and does not support flexible choice of gradients for partial synchronisation.

Chapter 3

Kungfu: Enabling Flexible Synchronisation in TensorFlow

By far, we have motivated the need for supporting flexible synchronisation in distributed training systems. In this chapter, we describe a new distributed training system called Kungfu that can effectively enable flexible synchronisation in a state-of-the-art DL platform: TensorFlow. Our system proposes new abstraction (Section 3.2 and 3.3) that can easily piggyback into existing TensorFlow API while allowing easy declaration of various synchronisation policies. Also, it proposes a novel implementation (Section 3.4) for training workers so that workers can flexibly exchange gradients and model variables with other in a scalable fashion. We implemented two novel synchronisation algorithms on Kungfu and end this section with comprehensive test-bed experiments.

3.1 System requirements

The design process starts from defining the requirements of the system. This focuses to improve Deep Learning scientist's experience of specifying training strategies (user level API) and to seamlessly allow distribution to scale in both high-performance and commodity clusters which suffer from network bandwidth shortage (over-subscription, poor link quality of service). Moreover, a system should combat explicit barriers leading to delays caused by stragglers (heterogeneous GPU clusters, resource sharing in multi-tenant Cloud environments). To achieve this it is necessary to provide: a high-level, yet flexible abstraction that can implement flexible synchronisation strategies and a high-performance communication stack that allows workers to efficiently exchange gradients and models.

(1) A high-level yet flexible synchronisation abstraction that can be integrated within existing DL platforms Deep Learning pipelines can often become difficult to manage, especially at scale. Thus, it is necessary to provide high-level (easy-to-use) and flexible abstractions that enable multiple synchronisation strategies with ease.

We have previously identified that TensorFlow does not provide sufficient granularity of its internal components of interest for synchronizatin. Essential building blocks of the dataflow graph must be leveraged (gradient and variable tensors) and the existing cluster abstraction must be transparently extended such that peer-to-peer communication support can be underlyingly enabled. The system should provide an API which enables the user to leverage these building blocks with minimal changes to the driver program. Most common APIs, such as the ones provided by TensorFlow [28] and Keras [32] often provide cumbersome abstractions for distributed training, which require manual specification of cluster allocation. More recently, systems such as Horovod [5] implement new paradigms of distributed training abstraction which enable easy specification of distributed tasks.

(2) A high-performance training worker implementation that allows efficient and flexible communication of gradients and model variables. Enhancing a system must not affect its performance, which requires efficient implementation that ensures distributed training happens with low overhead. Thus, the schemes employed for partial synchronisation must not involve

heavyweight computation at every iteration and dynamic decisions during training must rely on efficient metric approximations. Most common performance issues arise to the imbalance between communication and computation. Communication should be realised with minimal blocking of the critical training path by using directly intercepted gradients and variables, which are efficiently communicated to other peers.

(3) A scalable system architecture that can efficiently run on a large number of commodity training devices. The aim of distributed system design is scalability, which refers to minimisation of performance penalties incurred when increasing the number of processes in the system. While a user has at disposal clusters of tens or hundreds of machines, it may be difficult to benefit from the large number of resources due to communication bottleneck. The system must best leverage the resources to achieve high hardware utilization subject to constraints such as communication bandwidth. The choice of synchronisation strategy must be decoupled from the training process: every instant spent synchronising is time taken away from computation-intensive training.

Oftentimes, workers that benefit from hardware acceleration (GPUs, TPUs) do not benefit from similar performance enhancements at the network level. For example, state-of-the-art V100 GPU [36] can process tens of training mini-batches per second [37], but performance may be limited by large synchronisation barriers over commodity links or by lack of bandwidth due to overprovisioning [34] in modern data-centers. Cloud environments often represent a challenge for scaling distributed training system due to the communication limitations caused by the underlying system (.e.g, Docker [89] overlay networks or virtual switches add new levels of complexity in communication).

3.2 Synchronisation abstraction

TensorFlow and Keras have a high-level distributed training API [28, 32]. Though easy to use, it completely masks the cluster from the users and let users only able to use parallel SGD to synchronise replicas. More importantly, to adopt their distributed training API, developers must largely modify their single-GPU training programs to incorporate the functionality [5, 106]. More recently, the Horovod [5] library for distributed training is growing popular quickly. The advantage of Horovod is usability, as users can minimally change their training program to support distribution [5]. The disadvantages of Horovod are that it only provides support for parallel SGD, similar to Keras [32] and has insufficient scalability because it makes use of a rigid MPI-based [105] all-reduce library and a master-slave architecture.

Understanding TensorFlow abstraction The proposed non-invasive API for distributed training is motivated by tremendous simplification of the driver program, such that users can focus on defining the model. The responsibility for distribution is completely shifted towards the system, which transparently builds the dataflow graph, assigns tasks to hardware resources and creates an overlay for communication. This design is founded on usability and paves the path for accessible distributed strategies. Users create a driver program according to the high-level TensorFlow Python API [28]. This breaks down into four main components: dataset preparation, neural network layer specification, inference loss computation and optimization specification. Optimizers are classes which implement algorithms that compute gradients and apply them to model parameters. TensorFlow provides implementations of many stochastic optimization techniques widely used in practice (Gradient Descent [20] [18], Momentum [66], Adam [65], AdaGrad [64]). Essentially, these classes have the responsibility of training, as they provide functionality for the weight update rule, by which gradients are applied to the model. Thus, optimizers are the naturally occurring choice where the exchange seam can be inserted.

Design solution In order to achieve this in a manner that allows minimal changes to the driver program, new classes derived from the TensorFlow's Optimizer class [60] are created. The new wrappers override two methods of the Optimizer base class, which are shown in Listing 3.1 and in Listing 3.2, together with the hooks necessary for the corresponding synchronisation strategy realized in a non-invasive manner unexposed to the user. The listings highlight the interception point for gradients (Listing 3.1) and variables (Listing 3.2):

```
def compute_gradients(self, gradients, variables):
    """Intercept local gradients and perform worker negotiation."""
    negotiated_gradients = distributed_negotiation(gradients)
    return self.base_optimizer.apply_gradients(negotiated_gradients, variables)
```

Listing 3.1: Gradient interception before negotiation

```
def apply_gradients(self, gradients, variables):
    """Local model averaging and gradient update."""
    avg_variables = peer_model_averaging(variables)
    apply_gradients = self.base_optimizer.apply_gradients(gradients, avg_variables)
    with tf.control_dependencies([compute_average_model_op]):
        return apply_gradients
```

Listing 3.2: Seamless variable interception before peer model request

This approach provides a simple abstraction layer as a kick-off point for flexible synchronization strategies presented next. The **PartialGradientExchangeOptimizer** is responsible for wrapping the functionality of any TensorFlow optimizer in order to exercise its responsibility for intercepting the list of locally computed gradients, partitioning the gradient set and enabling the distribution strategy on the deeper level of abstraction as a TensorFlow dataflow graph.

```
__init__(
    optimizer,
    budget_fraction=0.1,
    device='gpu'
)
```

Listing 3.3: Class initializer for PartialGradientExchangeOptimizer

Args:

- `optimizer`: an instance of any `tf.train.Optimizer` TensorFlow class defining the learning algorithm by which gradients are computed.
- `budget_fraction`: fraction from the total size of gradients representing the capacity of a partition
- `device`: device for executing the synchronisation logic. Can be one of 'cpu' or 'gpu'. If 'gpu' is specified, synchronisation uses NCCL, otherwise all-reduce is done in parallel for each tensor using the underlying networking stack.

The **PeerModelAveragingOptimizer** wraps TensorFlow optimizers in order and intercepts all model variables during every iteration. Before executing the parameter update, the dataflow logic retrieves a model from one other peer and performs model averaging.

```
__init__(
    optimizer,
    peer_selection_strategy='roundrobin',
    request_mode='sync',
    model_averaging_device='gpu'
)
```

Listing 3.4: Class initializer for PeerModelAveragingOptimizer

Args:

- `optimizer`: an instance of any `tf.train.Optimizer` TensorFlow class defining the learning algorithm by which gradients are computed.
- `peer_selection_strategy`: peer selection algorithm for point-to-point model requests. Can be one of 'roundrobin' or 'random'.
- `request_mode`: device where model averaging is computed. This paramater is sensitive for training performance. Can be one of 'gpu' or 'cpu'. 'cpu' means that averaging is executed within the request CPU operator. 'gpu' means that TensorFlow operators are used to execute averaging on GPU.

Algorithm 3: Partial Gradient Exchange (Logical View)

input : $G = \{g_0 \dots g_{N-1}\}$ set of gradient tensors;
$GlobalStep$ global step;
output: $PNG = \{g_0 \dots g_i \dots g_{i+k} \dots g_{N-1}\}$ set of gradient tensors where only a subset of gradients $\{g_i \dots g_{i+k}\}$ are negotiated via all-reduce

1 $P \leftarrow partitionEqualSubsets(G)$ /* $P = \{p_1 \dots p_k\} : \quad \forall i, j \quad size(p_i) - size(p_j)$ is minimized */
2 $localGradients \leftarrow \{\}$;
3 $aggregatedGradients \leftarrow \{\}$;
4 **for** $p_i \in P$ **do**
  /* $p_i = \{g_j \dots g_{j+|P|}\}$ */
5 **if** $GlobalStep \mod |P| = i$ **then**
6 **for** $g_j \in p_i$ **do**
7 $gAggregated \leftarrow aggregateFromAllWorkers(g_j)$
  $aggregatedGradients \leftarrow aggregatedGradients \cup \{gAggregated\}$
8 **end**
9 **else**
10 $localGradients = localGradients \cup \{g_j \dots g_{j+|P|}\}$
11 **end**
12 $PNG = localGradients \cup aggregatedGradients$;
13 $PNG = restoreInitialOrder()$;
14 **end**

3.3 Synchronisation operators

This subsection provides an insight into the logic behind the dataflow operators used to build the training system and explains how distributed strategies are achieved by creating new operators which are directly plugged in the TensorFlow execution graph. Design decisions follow-up on enhancing the functionality of newly defined optimizers and reusing existing methods called on the training dataflow execution path. Building new operators is necessary to access the underlying network stack which encapsulates cluster abstraction and communication primitives. A new enhancement in the networking stack allows support for direct peer-to-peer communication for further exploration of decentralised learning.

Algorithm: Partial Gradient Exchange Before diving into concrete TensorFlow dataflow specification, it is important to understand the logical view behind Partial Gradient Exchange. Algorithm 3 is a formalisation of the main steps required to partially exchange gradients between peers. Initially, partitions are created (line 1) to encapsulate approximately equal sized disjoint subsets of the total gradient set, where all gradients belonging to one partition are negotiated at once (lines 6-8). When the global step indicates that the current partition should not be negotiated, all local gradients belonging to the partition are used for the variable update (line 10). The algorithm returns a set containing both aggregated gradients and local gradients (line 12), taking care to restore initial gradient order for consensus on variable update order (line 13).

New operator roles Concrete dataflow implementations of the functions highlighted in Listing 3.1 and Listing 3.2, for gradient exchange (`distributed_negotiation(gradients)`) and for peer-to-peer unidirectional variable exchange and averaging (`peer_model_averaging(variables)`) are: a *partial gradient exchange negotiator* - responsible to select partitions to be negotiated via all-reduce in round robin fashion using global step and to keep track of partition index where gradient is placed, *synchronous and asynchronous model request and averaging* operators - responsible to engage in unidirectional model exchange with one other peer, followed by averaging and *a model store* - responsible for keeping up to date copies of model variables.

Design trade-offs: conquering TensorFlow abstraction The initial design stages of the system focus on analysis of benefits and drawbacks of the possible ways of implementing the logic to achieve partial gradient exchange and peer-to-peer model averaging. The choices revolve around

two options: using the TensorFlow dataflow API or using a more flexible implementation by implementing custom TensorFlow operators within the C++ runtime system, which would be compatible with the user-level API. The advantages of the former approach refer to code simplicity achieved by an implementation in the high-level TensorFlow API. However, the drawbacks are that lack of flexibility, as developer can not define desired custom behaviour, for example a stateful operator with complex state-tracking data structures or an adaptive behaviour which dynamically changes hyperparameters of the system. The latter approach may lead to possible design complications, because they delegate the responsibility of specific operator registration for device placement to the user (CPU operators and GPU operators). This would naturally split the concerns of operator logic, which needs to be explicitly handled by the developer according to the device.

```
1  def partial_exchange_with_gpu_all_reduce(global_step, gradient_tensors):
2      """Partial exchange with GPU all-reduce."""
3      partitions = partition_gradient_set(gradient_tensors)
4      negotiated_gradient_tensors = []
5      for i, partition in enumerate(partitions):
6          negotiated_partition = tf.cond(
7              tf.equal(tf.mod(gs, num_partitions), i),
8              lambda partition=partition:  gpu_group_all_reduce(partition) ,
9              lambda partition=partition: partition)
10         fill_with(negotiated_partition, negotiated_gradient_tensors)
11     return negotiated_gradient_tensors
```

Listing 3.5: Distributed negotiation: dataflow simplicity for partial exchange

In order to conquer the abstraction behind distributed negotiation, which has been presented in the previous section, Listing 3.5 illustrates a simple way to use the high-level Python API provided by TensorFlow to implement a static partial exchange scheme based on gradient partitioning. This constitutes a simple bridge between the dataflow and the network stack which enables collective communication.

Algorithm: Peer Model Averaging Before diving into the concrete TensorFlow specifics required for Peer Model Averaging, it is important to understand the logical view. Algorithm 4 formalises the steps required for direct peer interaction for model exchange. Function *peerModelAveragingOptimizer* represents the *request sender* side and presents the steps for model request, averaging and variable update. Initially, each worker enters an initialisation phase, where model variables are identically initialised across all workers using broadcast (line 2). A first model checkpoint is made in the model store (line 2), such that workers can reply asynchronously to requests received from other peers before any training step has completed. Using the peer selection strategy, a destination peer is selected (line 4) and a request is issued to this peer (line 5). In the synchronous case, execution blocks until the requested model is delivered. A model average is performed element-wise on the two variable lists of requester variables and destination peer's variables (line 8), followed by a an update of the *average model* with the locally computed gradients (line 9 - 11). After the training iteration completes, the updated model is checkpointed in the model store (line 12). Function *handleRequest* is executed on the destination peer's side by: concurrent access to the model store (lines 15-17) for model retrieval and direct reply to the requesting peer.

The design of peer-to-peer model averaging becomes problematic, as TensorFlow does not provide sufficient cluster abstraction and the API does not include lightweight request - reply operators which can be used to directly communicate with a worker while training. At this point, the proposed API becomes disruptive. It introduces a model request operator which has two modes of operation. One is synchronous and blocks training until a reply containing a serialized model is delivered, while the second one is asynchronous and leverages concurrency features enabled by the flexible network stack implementation. The request dataflow hooks are now able to delegate a request while being integrated into the execution graph. Subsequent logic leverages the TensorFlow API and provides three benefits: (1) when the graph is placed on the GPU, execution leverages a high-degree of parallelism and a basic operation such as tensor averaging becomes inexpensive through GPU optimizations provided by the TensorFlow library, (2) simplicity of dataflow specification and dependency control on operation executions using the special `control_dependencies` TensorFlow block, which ensures that gradient updates are applied after the average is computed and (3) tensors are easy to manipulate, which also enables users to define other model mutation techniques, for example the momentum-based model averaging defined in the Crossbow system [57]. Another component of the peer-to-peer model averaging is model serving. In order to ensure fast replies, an in-memory model store is designed. Workers save a serialized copy of their model

Algorithm 4: Peer Model Averaging (Logical View)

function `peerModelAveragingOptimizer(`*V, G, η, strategy, requestMode*`)`
  input : $V = \{v_0 \dots v_{N-1}\}$ set of variable tensors (the model);
  $G = \{g_0 \dots g_{N-1}\}$ set of gradient tensors;
  η learning rate;
  strategy peer selection strategy;
  requestMode request mode;
  output: $V^\star = \{v_0^\star \dots v_{N-1}^\star\}$ updated model after average with one selected peer

  `/* Executed once, at the beginning of training */`
  $initializeModelVariables(V)$;

  `/* First checkpoint, initialize Model Store (executed once) */`
  $saveModel(V)$;

  $p \leftarrow selectPeer(strategy)$;

  `/*` $PV = \{pv_0 \dots pv_{N-1}\}$`, peer` p`'s model */`
  $PV \leftarrow requestModel(p)$;

  if *requestMode* ***is*** *synchronous* **then**
    $blockUntilReplyDelivered()$;

  $[v_0^\star \dots v_{N-1}^\star] \leftarrow [\frac{1}{2}(v_0 + pv_0) \dots \frac{1}{2}(pv_{N-1} + pv_{N-1})]$;

  `/* Update average model */`
  for $i \in \{0 \dots N-1\}$ **do**
    $v_i^* \leftarrow v_i^* - \eta \cdot g_i$;
  end

  $saveModel(V^\star)$;
  return $V^\star$;

function `handleRequest(`*srcPeer, connection, modelStore*`)`
  input : *srcPeer* request source;
  connection communication channel (TCP connection);
  modelStore referene to local Model Store;
  `/* Access Model Store */`
  $lock(modelStore)$;
  $V' \leftarrow retrieveLocalModel(modelStore)$;
  $unlock(modelStore)$;

  `/* Reply */`
  $replyWithModel(connection, srcPeer, V')$;

first after weight initialization and subsequently at the end of each iteration. As this is a custom addition compatible only with the base networking stack of Kungfu, a new operator was created to bridge this interaction.

3.4 Worker implementation

Next, we describe why the proposed design achieves *homogeneous* workers able to communicate in a *decentralised* manner. First, we highlight new contributions to the worker modules. We detail the implementation and motivate design choices, followed by concrete examples of worker interactions.

3.4.1 Overview

The new modules of the system support partial gradient synchronisation (*Partial Gradient Exchange (PGX)*) through collective communication and model synchronisation through direct peer-to-peer communication (*Synchronous and asynchronous Peer Model Averaging (SPMA and APMA)*). These components comprise new dataflow logic capable of interacting with the underlying communication stack while being executed by the TensorFlow runtime. The approach most effectively

tackles this challenge by using existing operators and when needed, defining new TensorFlow operators which can be plugged in the dataflow graph. We present how these components integrate by isolating the main stages of distributed training using our system (support Figure 3.1) are described next:

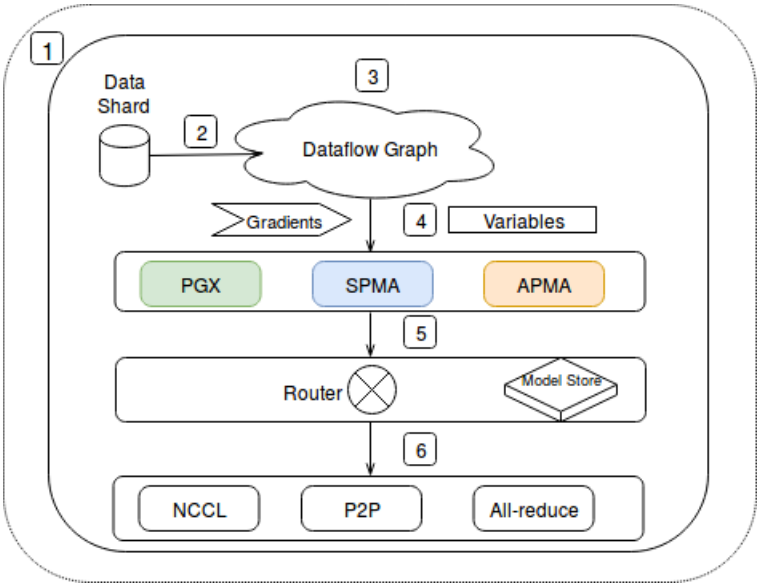


Figure 3.1: Worker architecture: PGX is the Partial Gradient Exchange strategy, SPMA is the Synchronous Peer Model Averaging strategy and APMA is the Asynchronous Peer Model Averaging Strategy

1. Workers load a data shard into memory and create an iterator to efficiently process batches. This is done by all workers on *different shards* of the entire dataset.
2. User-specified driver program is interpreted and the runtime execution graph is created. This stage is realised by the TensorFlow runtime system.
3. The model iteratively trains on batches of data, computing a new gradient set at each iteration. This is the model training stage, realised locally by each worker.
4. Variables and gradients are delegated to the **synchronisation strategy**, which saves them in the Model Store or passes them directly to the Router for negotiation. This is depends on the user's choice of synchronisation strategy and represents new functionaliy enabled by the system.
5. Communication between workers is realised through a **light-weight overlay networking stack** or through NCCL. The networking stack is deployed as a wrapper of the training job and is initialised on deployment on each worker. The entire cluster abstraction is accessible from the communication stack provided by the wrapper process.

The implementation is focused on encapsulation and homogeneity, creating a decentralized training system in which a worker can flexibly enable specific components subject to cluster constraints. The principal modules allow workers to play distinct roles during training. All new operators that augment the TensorFlow operator set are built using C++ as classes derived from `SyncOpKernel` and `AsyncOpKernel` [107]. A kernel defines the runtime computation of an operator and allows it to maintain state. As described in Section 2.2, kernels can be registered for a specific device type to leverage training acceleration hardware. An important mention is the difference between the two types of kernel. While the synchronous version blocks the critical path of computation, the asynchronous version allows free execution of other kernels until the output of asynchronous kernels has been computed. This insight reflects implementation decisions for partial gradient exchange and for peer-to-peer model averaging.

The logic within the networking stack for supporting peer-to-peer interactions is implemented in Go [108]. This programming language provides useful concurrency features needed to support

asynchronous I/O required by the synchronisation communication stack through goroutines, which represent lightweight green threads managed by the runtime system and through Go channels, which are thread-safe data types which enable message passing between goroutines. These features meet the requirements of the network stack: (1) handle all complex collective communication components (broadcast and gather graph, gradient receiving, aggregation and sending) and (2) handle asynchronous I/O for flexible TCP connection handling.

3.4.2 Partial Gradient Exchange

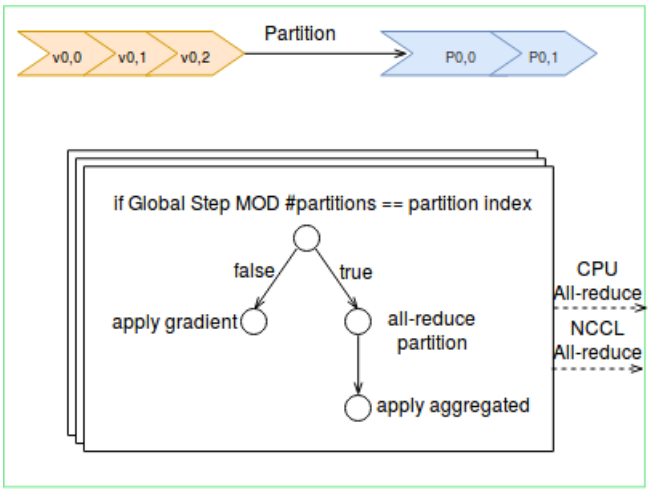


Figure 3.2: Partial gradient exchange dataflow

This synchronisation strategy uses existing TensorFlow operators for defining the logic of partial gradient synchronisation through round robin scheduling of partitions. During training, gradients are first intercepted and partitioned into approximately equal-sized buckets. These are scheduled at runtime using the global step as a partition selector. When a partition is selected, all its gradients are negotiated using all-reduce. The design supports easily both NCCL all-reduce and CPU-based all-reduce. This collective communication is handled by the communication stack of Kungfu.

Partitioning The partial gradient exchange component intercepts the set of gradients computed for each trainable variable of the model. In order to exchange less gradients during synchronisation, the set is partitioned. An important decision at this stage is the choice of partitioning algorithm: one approach inferred in the Ako algorithm [33] is that of partitioning the entire serialised gradient set (a byte buffer) into a number of chunks equal to the number of workers, while the other approach is to partition the gradient set directly by the number of bytes of the gradient tensor. As the level of granularity employed by the former approach is likely to affect convergence because it assumes an asynchronous setting with gradient accumulation, the latter approach is chosen. First, an approximation of the K-partitioning algorithm - implementation based on [109]) - is explored, which places variables in K bins constrained to have approximately equal sizes (where possible). However, this approach proves to be model dependent and bins turn out to be disproportionate across different models.

Partitioning design choice Although dependence on model architecture is not completely eliminated, a new solution, which is more intuitive for the user relies on the bin-packing problem [110] - algorithm implementation adapted from [111]. The algorithm finds the minimum number of bins of fixed capacity required to fit all variables according to variable size. This algorithm does not impose any constraints on variable placement in bins such as placement by variable type which can be convolution or fully connected layer tensors. The **motivating choice** for this type of offline partitioning is to obtain a set of equally sized bins conditioned on the user-provided budget, which is important for balanced network usage.

Dataflow implementation The partitioning step is run before the dataflow is created. Each gradient tensor belongs to a unique partition. Each partition is negotiated in a round robin fashion, depending on the global step - this logic is run in the TensorFlow execution graph. As described in Section 2.2, the global step TensorFlow variable is used to keep track internally of the training step. The dataflow graph in Figure 3.2 is replicated for all gradient partitions, which is a subtle detail of the dataflow paradigm. Using a TensorFlow conditional operator, the runtime behahviour branches: if the current partition must not be negotiated, all gradients within which represent local worker-computed gradients are applied to the model, otherwise all grdients enter an all-reduce phase which can leverage NCCL GPU acceleration or run on the CPU, using the networking stack for communication. One special requirement needs to be met in the case of NCCL all-reduce: all GPU devices must agree globally on the order of execution, which is realised by defining an *order group* such that all-reduce can be done identically for all partitions across devices. This is supported by the Kungfu implementation of NCCL collective communication. Partition scheduling is best shown in Figure 3.3 where the step i deterministically selects the gradient partitions $P_{\cdot,i \mod numberOfPartitions}$ for synchronisation, where $numberOfPartitions = 3$.

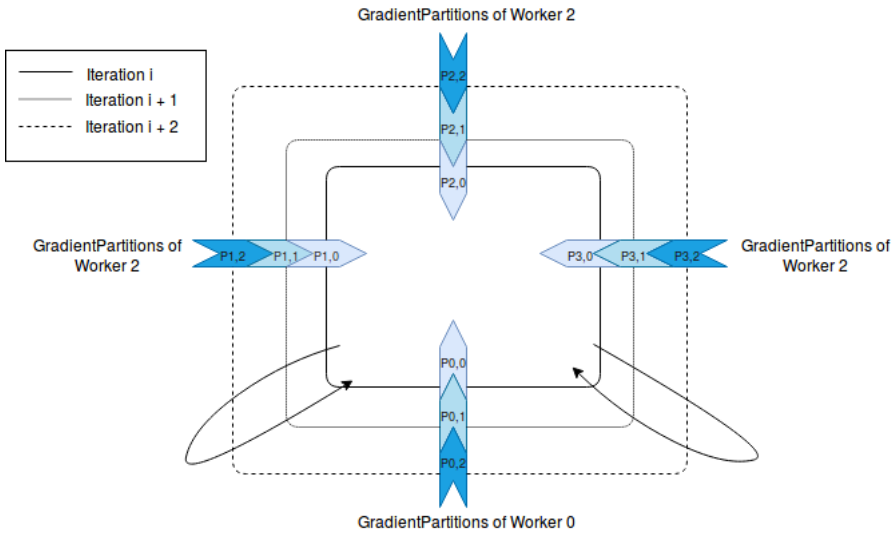


Figure 3.3: Round-robin partition synchronisation

Worker interaction Figure 3.4 represents a conceptual interaction with topology abstracted away. Workers take part in the all-reduce algorithm as explained in Appendix A.2 with less gradients. This algorithm is efficient because it considers only multicast interactions with direct neighbours of a worker in the spanning tree of the topology graph. Our focus is to improve communication system-wide and this is best shown in the diagram. There are three gradient variables in the system summing up to a total size of 8. Assuming the user desires partitions with half the capacity of the total model size, there will be two gradient partitions who will halve traffic created in the network compared to a full exchange system.

3.4.3 Peer Model Averaging

There are two components for peer-to-peer model averaging which rely on a common mechanism: after each iteration workers issue a request to another peer selected according to a user-specified strategy, the peer prepares a response from its local model store containing a serialised model and the requester updates its local model by plain model averaging. The distinguishing feature between the synchronous and asynchronous implementation is the latter's leverage of concurrency features of the underlying networking stack.

The final implementation is a result of decisions on: (1) design of the communication protocol, (2) integration of the communication protocol in the TensorFlow dataflow graph:

1. Defining how workers communicate is a requirement for a predictable protocol. An asynchronous communication protocol in which there exists no flow control becomes problematic

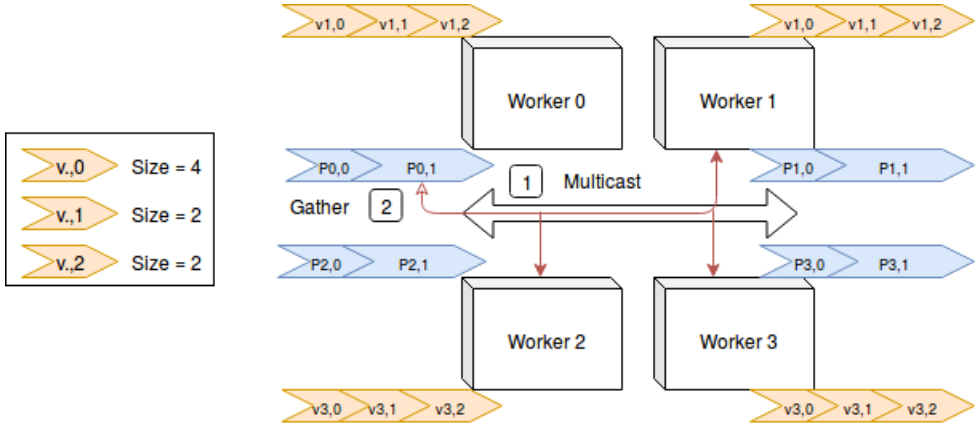


Figure 3.4: System-wide communication reduction through partial exchange

in cases where the system is sensitive to data staleness. As, such the protocol implemented relies on TCP connection reuse for requests and replies. This way, the model receiver knows precisely that delivered model is not stale and can pass it to higher-level layers in the system.

2. The implementation proceeds from the interception point of the model variables represented as multi-dimensional tensors. But how can these variables be averaged with another peer's variables? This is achieved through a new TensorFlow operator delegated with the responsibility for the request. The request can be made for each individual variable or for the entire model. The first approach is likely to lead to suboptimal network usage or to high contention on the model store, which leads to the simple solution of coarse granularity for model representation.

Model Buffer `ModelBuffer` is a C++ utility class which is responsible for serialisation and deserialisation of the list of variables which constitute a model. The class contains the data bytes of all tensor variables of the model and a reconstruction vector which keeps track of byte offsets. Each offset is the starting point where data bytes of a variable are found. This object is unique for operators responsible for model averaging and model saving and is doubled by a *prefetch* buffer for the asynchronous operator. The implementation is based on the global ordering of model variables across workers, such that correct reconstruction of variables can be achieved using offsets and variable sizes.

Peer Selector Given that a training iteration has completed, how does a worker know which peer to request a model from? It is hard given only local knowledge of the model to make an informed decision. We employ two strategies that make use of the cluster abstraction: (a) *random selection*, achieved using the global cluster knowledge registered in the communication stack, represented by peer IDs also known as ranks and (b) *round robin peer selection*, using the same cluster knowledge and the training step. These strategies represent a good starting point to discriminate which peer selection strategy is better: one that is *unpredictable* and is likely to achieve a *better approximate trajectory close to the global average model* or one that is ordered and may show benefits of periodic synchronisation with peers. Peer selector is implemented as an abstract class extended by two strategies: random peer selector and round robin peer selector. The random selector uses a uniform integer distribution utility class to select uniformly at random an element from the list of peer ranks. The round robin selector keeps track of the global step using an integer variable incremented with every call for selection and uses this variable to index into the vector of peer ranks.

Model Request There is a unique model averaging operator per worker. Both synchronous and asynchronous versions are implemented as TensorFlow `OpKernel` derived classes. On construction,

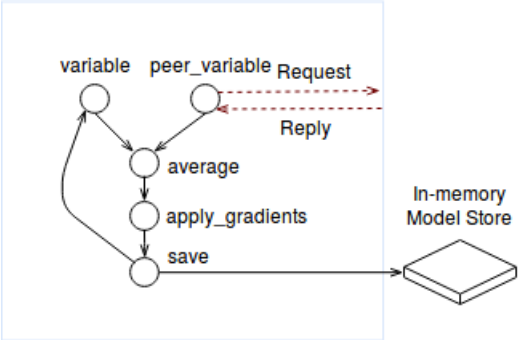


Figure 3.5: Synchronous peer-to-peer model averaging dataflow

the class initializes variable meta-data (sizes, byte offsets) for recovery of model variables from the model store. The logic of the operator is placed in the overriden `Compute` function and called at runtime during every iteration when the dataflow graph is executed. The synchronous version illustrated in Figure 3.5 blocks training for a duration equal to the request latency. When another peer's model is received, the model buffer is populated and the variable tensors are reconstructed (See figure: *peer_variable*) to be further processed by other operators in the dataflow graph for model averaging (Figure pointer: avg operator).

This approach is optimized through a second prefetch buffer which acts as a cache. This architecture decouples the training task from the model serving task and the blocking request task for less overhead during training, according to Figure 3.6. Although the model request operator is an `OpKernel`, background requests are achieved through registering C++ callbacks delegated to the networking stack through cross compatibility enabled by CGo. When such a callback is provided go launches a goroutine for the process and calls the provided C++ callback after the request completes. Goroutines are light-weight green threads which enable flexible concurrency in the Go programming language. Because of the large number of iterations required to train neural networks, launching a background request every time overloads the Go runtime, which supports a maximum of 10000 goroutines. This demands a mechanism for request rate limiting to ensure that no redundant background tasks are spawned. This is achieved through an atomic boolean flag which is reset after an existing background task completes. This mechanism also ensures that the cache is always up to date.

Model serving is realized through TCP connection reuse. This means that when a requester establishes a TCP connection with a destination peer, this connection is kept alive and the message from the receiver is delivered using the same connection. The underlying networking stack provides flexibility for connection handling through a connection pool conceptualized through a Go channel, which can receive messages of from multiple connection types according to the type of interaction which can be one of peer-to-peer or collective. A model is served from the local model store, implemented as a thread-safe byte buffer. This is done when the peer-to-peer connection is handled, always acquiring a lock on the model store. The benefit obtained from placing the model store deep in the networking stack is to minimise the latency incurred by constructing a reply message. One advantage of the serialised model store is reduced contention when peers are flooded with requests and are updating the store frequently at every training iteration.

Model Saving Workers need to periodically update the model store such that other peers are not served stale models. This is done during every iteration through a similar callback-based mechanism with background task rate limiting as for model requests. Model saving is a simple buffer copy from the C++ buffer allocated by the `ModelBuffer` object to the Go-land model store buffer.

Decentralised interaction Figure 3.7 illustrates a snapshot of the decentralised interaction in which Worker 3 randomly selects Worker 0 to engage in a unidirectional model exchange, while all other workers continue their training process. This approach reduces network traffic and allows training to progress while synchronisation takes place in parallel. This interaction is based on the algorithm developed by Zhang, et al. in the paper *Asynchronous Decentralized Parallel SGD* [58],

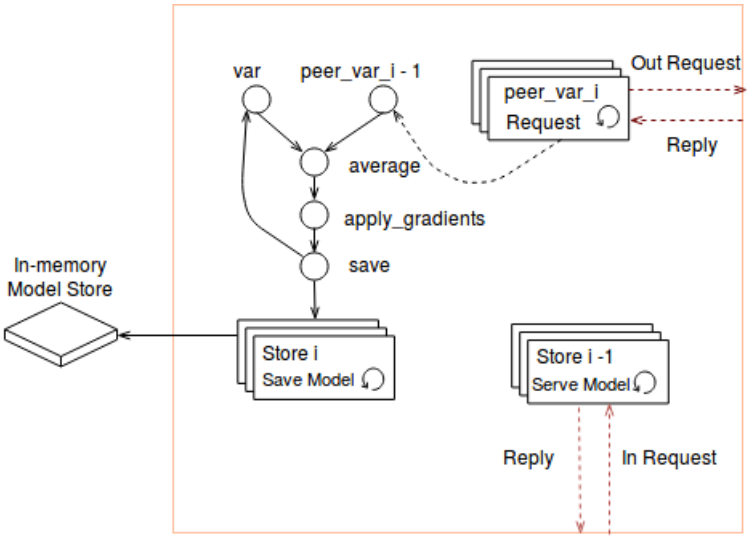


Figure 3.6: Asynchronous peer-to-peer model averaging dataflow

2018, however they present a variant in which there exists a bidirectional model exchange between peers.

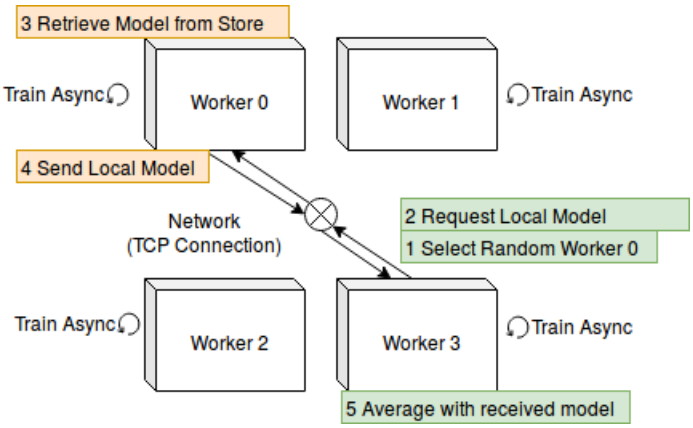


Figure 3.7: Unidirectional Peer Model Averaging

3.4.4 Worker Architecture

The natural question is *how can synchronisation be effective when it is not strict and there is no single entity responsible for strictly collecting gradient information from all workers?* Deep Neural Network training does not yield a unique solution due to its stochastic nature, which is an avenue worth exploring for decentralised training, where barriers are relaxed with the confidence of maintaining convergence properties. In a decentralised setting, all workers are capable of the same behaviour and periodically bring contributions to the system. To achieve this behaviour in our system, the design is focused around encapsulating the same functionality in all workers.

Homogeneous Deploying Deep Learning jobs in the cloud can often be a cumbersome process, sometimes requiring users to manually map the cluster resources to specific roles in the training process (e.g., resource partitioning in parameter servers). As [33] describes, finding the optimal resource partitioning is not an easy task and it tightly depends on a number of factors which users are unable to control or may not be aware of. The paper [33] also shows empirically that resource partitioning focused on improving specific metrics such as iteration completion time (*hardware efficiency*) or model improvement per iteration (*statistical efficiency*), degrades the quality of a

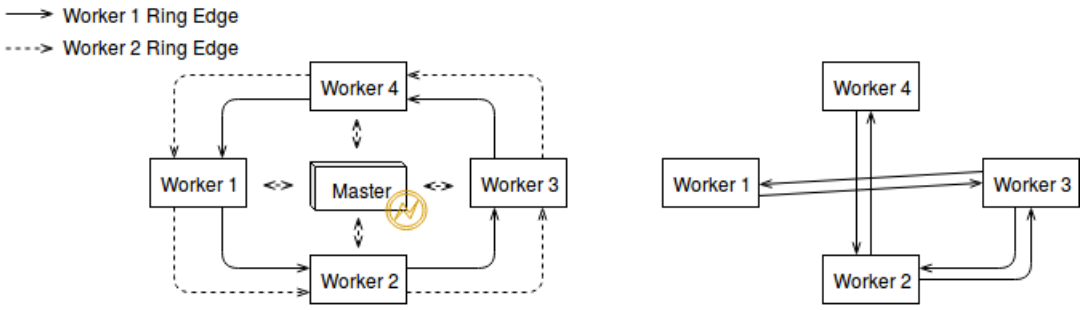


Figure 3.8: Shift to a decentralised architecture

model [33]. Users may not always benefit from ready-made cluster configurations for their model and even if they do, they may not dispose of the necessary hardware resources.

Our aim is to design a homogeneous system for ease of deployment. Kungfu's architecture is scalable and easy to deploy. Workers are built as independent entities with homogeneous functionality, so there is no resource partitioning as in parameter servers. This eliminates the need for manual training configuration. When starting a Kungfu process, users need to specify only the host IPs and how many workers they wish to deploy on a host (e.g., for an 8 GPU machine, a user can specify any number of workers between 1 and 8, each training on one GPU). This is convenient, as the cloud usually provisions different types of resources and people want to easily deploy distributed training jobs without the burden of resource partitioning.

Decentralised The system fits in the paradigm of decentralized architectures. Previous systems such as Horovod rely on a master node to determine when synchronisation is complete, which makes training vulnerable to failure with no fault tolerance mechanism in place. When DL clusters increase it is often cumbersome to have master nodes in the system. Systems such as Horovod exhibit scaling difficulties even on a small number of machines mainly due to the master node.

Figure 3.8 shows two types of decentralised interactions realised by the Kungfu system, where the master role is completely eliminated. The new type of interaction is realised by collective communication and direct peer interaction. All-reduce is often realised through bandwidth-optimal [101, 50] ring all-redue algorithms, where the overlay topology graph is a set of rings for each peer - every peer in the ring aggregates gradients from all peers by successive send operations. All-reduce can alternatively generalise (Appendix A.2) to other overlay topologies such as tree, star and clique. The user choice can depend on specific cluster knowledge, but by defult the system does not require such changes. This approach provides significant communication adaptation benefits (dynamic cluster overlays, cluster cliques) that could benefit flexible synchronisation. In the proposed system, control is delegated uniformly to workers. For direct peer interactions, which requires reliability, TCP sockets are used assuming overlay neighbors are known. These enable the system to become resilient to worker failures and to easily employ fault tolerance schemes such as TCP connection timeouts (for peer interactions) or back-up overlay graphs for collective all-reduce.

The proposed worker architecture shows that partial gradient exchange and unidirectional model averaging can be achieved in a fully asynchronous, decentralised way. They do not require a coordinating process and can instead *autonomously* perform all operations required for flexible synchronisation - gradient aggregation through all-reduce and direct interactions with peers.

3.5 Evaluation

In this section, we evaluate the system performance. The first question to answer is: does flexible synchronisation hurt convergence of a DL model compared to parallel SGD? To answer this question, we perform convergence experiements that compare the model averaging techniques and partial gradient exchange with parallel SGD, and show that *some* flexible synchronisation strategies can maintain the accuracy bounds tight. Once resolving the convergence question, we turn our focus to evaluate the effectiveness of resolving communication bottlenecks by using a flexible synchronisation system. The evaluation is organised as follows: we start by presenting the ex-

perimental set-up consisting of Cluster hardware, models and their chosen hyperparameters, then describe the datasets.

Goals of the experimental evaluation are: (i) to explore convergence properties of the system when using proposed flexible synchronisation strategies on state-of-the art DNNs, (ii) to quantize the benefit obtained from flexible synchronisation compared to baseline systems, (iii) to explore scalability in high-performance Cloud environments dedicated for DNN training and in multi-machine GPU clusters with commodity network links, (iv) to explore the impact of new hyperparameters introduced for synchronisation on convergence and scalability (v) to explore convergence under different conditions (multiple machines, varying batch size).

3.5.1 Cluster Hardware

To demonstrate the effectiveness of flexible synchronisation, three cluster set-ups are used. The first cluster setup uses latest-generation GPUs and network interconnects and is used for convergence and scalability tests. The second cluster setup is for multi-machine scalability tests, over virtual, commodity network. The third setup involves a server machine where smaller models such as ResNet-32 are run.

Cluster choice motivation The cluster choice emphasizes wide benefits users can obtain through flexible synchronisation in different settings. Firstly, the choice of a GPU cluster with InfiniBand [46] interconnect provides high bandwidth and ultra-low latency, so the user might be *less* motivated to use flexible synchronisation, so if we show existence of a benefit, the system becomes more likely to be used in practice. This was indeed achieved by showing training time reduction in the DGX-1 cluster. Secondly, the 16-machine cluster is a more realistic set-up, providing best-effort network services over commodity links. This fits in the types of clusters where switches are over-provisioned, likely creating bandwidth shortage for the hosts involved. This cluster is used in order to quantify how much benefit we obtain in an unpredictable environment with lower quality links. This is useful when analysing flexible synchronisation strategies, as its builds a complete view of the performance benefits motivating its practical use.

Hardware description The technical specification of the cluster setup is summarised below:

1. Cluster of 2 NVIDIA DGX-1 [112] machines, each dedicated with 8 Tesla V100 GPUs, with InfiniBand [46] interconnect, Dual 20-Core Intel Xeon E5-2698 v4 2.2 GHz and 256 GB system RAM.

2. Cluster of 16 machines, each dedicated with one NVIDIA P100 GPU, used for scalability tests. The cluster belongs to a public resource pool, which means that *there is not Quality of Service*. The network service is *best-effort*.

3. Server machine dedicated with 4 GeForce GTX TITAN X (Pascal) GPUs and 20 Intel(R) Xeon(R) CPU E5-2640 v4 @ 2.40GHz

Cluster set-ups 1 and 2 provisioned by Huawei Cloud's [41] cluster deployments rely on the Kubernetes [113] container orchestrator to manage and schedule Docker [89] containers encapsulating training jobs on multiple machines.

3.5.2 Models and Datasets

Models and datasets choice motivation The experiments aim to show that flexible synchronisation strategies can benefit training of a wide range of reference DNNs which have been well studied. These are chosen to create sufficient computational load on the system in order to enable a more accurate performance evaluation and to provide a sufficiently difficult training algorithm which is sensitive to any problem introduced by the system. Showing that the implemented strategies work with a wider range of models and datasets reinforces the motivation for practical use of the Kungfu system.

Kungfu all-reduce (CPU)	**Kungfu all-reduce (GPU)**
all-reduce in parallel for each tensor	all-reduce operators do not overlap, NCCL benefit
Horovod (MPI)	**TensorFlow Replicated**
centralised all-reduce with no parallel execution, blocking tensor fusion	centralised all-redue with no parallel execution

Table 3.1: Summary of baseline systems

Models and datasets details Experiments are run using an adapted version of the *TensorFlow benchmarks* project [59], which includes models for Deep Convolutional Neural Networks. The benchmarks version is compatible with TensorFlow version 1.12.0, which uses CUDA 9.0.

The main models used to run experiments are from the ResNet family [11]. Two models are used for evaluation. ResNet-32, a deep and low-dimensional network [57], which is trained on the CIFAR-10 [62] dataset (50,000 training images and 10000 validation images) and ResNet-50, a deep and low-dimensional large network [57], which is trained on a subset of the ImageNet dataset [22] containing $1,281,167$ images for training and 50,000 images for validation. All ResNet experiments share the same hyperparameter settings: weight decay 0.0001, momentum 0.9. Each training run is preceded by a warm-up stage of 20 batches, which is not timed. ResNet-32 is trained for 140 epochs and ResNet-50 is trained for 90 epochs in all experiments. In order to test the accuracy of the model, a checkpoint is created after every epoch - checkpointing time is subtracted from the training time. After training completes, all checkpoints are restored and used for testing on the validation set. The training instant (seconds from the beginning of training) where a checkpoint is made is used as a data point for physical time in the reported results.

It is important to understand a few common properties of ResNet models for a more clear view of the experimental results. First, they use a learning rate schedule [114], which encompasses pre-defined learning rate adjustments in order to overcome plateaus in validation accuracy. The effect of the learning rate schedule can be observed in the jumps in validation accuracy made at specific epochs during training. In ResNet-32, the learning rate is multiplied by 0.1 at epochs 80 and 120 [57] and in ResNet-50, the learning rate is multiplied by 0.1 at epochs 30, 60 and 80. Second, they use batch-normalisation [115] which is used to eliminate input similarities (covariate shift) in convolutional layers. They work by normalising the inputs of convolutions using batch statistics (mean and variance). Numerical instability leads to differences in model parameters across workers. This has a visible effect in the validation accuracy of parallel SGD, where there is variance in the validation accuracy computed by each worker.

3.5.3 Overview

All convergence results presented gather validation accuracies of all workers employed in the distributed training run. ResNet-50 convergence tests are run with 8 GPUs (workers). ResNet-32 experiments are run with 4 GPUs (workers). Each data point in a validation accuracy plot represents the average validation accuracy of all workers in the system. We also show the minimum and maximum validation accuracy achieved by a worker. Results presented in this manner provide a better understanding of how well the system behaves overall. Where the span of validation accuracies per worker is large, it means that there exists instability in the system.

Experiment design In designing the experiments, two factors are considered: (1) providing realistic convergence results comparable to the ones achieved by state of the art Deep Learning systems, (2) emphasizing the benefit of flexible synchronisation strategies by increasing the frequency of synchronisation in the system, while ensuring that hardware utilisation does not degrade significantly. To meet all these requirements, we choose small batch sizes, similar to [57]. All peer model averaging experiments are run with a batch size of 64 (ResNet-32 and ResNet-50). A small batch size ensures that workers perform model averaging more frequently, creating thus a better estimate for the *theoretical* global average model. For Partial Gradient Exchange experiments the batch size choices are: B = 64 for ResNet-50 and B = 32 for ResNet-32. The motivation for this is stress test the system when synchronisation could indeed become a bottleneck (e.g., in commodity networks such as the one underlying the P100 cluster).

Baselines One baseline is the MPI-based all-reduce implemented by Horovod, which is centralised, with the rank zero worker being the master. A version of Parallel SGD using collective

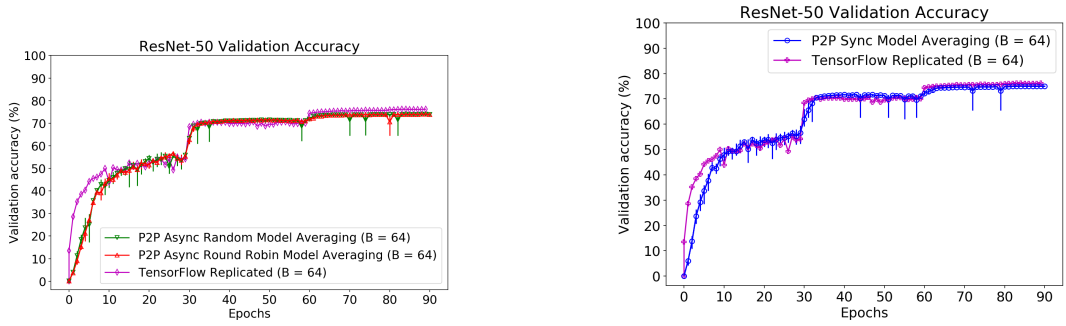


Figure 3.9: (a) Async Peer Model Averaging vs TensorFlow Replicated over epochs, (b) Sync Peer Model Averaging vs TensorFlow Replicated over epochs

all-reduce (Kungfu-provided baseline) is used for performance comparison, which coexists in the same system with Partial Gradient Exchange. Kungfu Parallel SGD uses collective all-reduce for CPU and NCCL synchronisation (Appendix A.2). This is performed in parallel for each gradient tensor, with the tensor name as global key. The key mechanism is the use of broadcast and gather graphs for each worker, where workers communicate with their direct neighbours in the spanning tree. One other baseline used is TensorFlow Replicated [116, 117] strategy with default all-reduce settings for the multi worker case (one machine, multiple GPUs): a centralized all-reduce algorithm is performed by cross-device aggregation of gradients at worker 0, followed by a broadcast. The default benchmark setting is `pscpu/pscpu` with an all-reduce shard size of 2. This means that there is a hierarchical all-reduce happening in parallel, each dealing with two shards of the dataset. The baselines are summarised in Table 3.5.3.

3.5.4 Peer Model Averaging

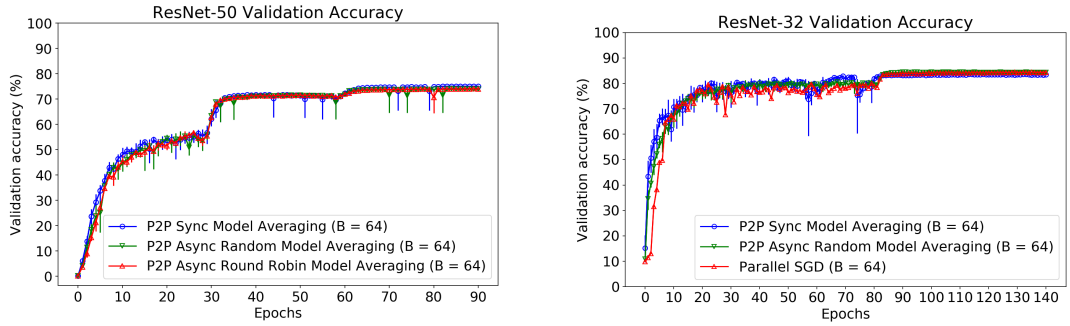
Evaluation shows that flexible synchronisation strategies do not affect convergence. The baseline is TensorFlow 1.12 Replicated strategy, which does a cross-device aggregation of gradients across GPUs. We use short names to denote the new peer-to-peer strategies for brevity of description: APMA denotes P2P Asynchronous Peer Model and SPMA denotes P2P Synchronous Model Averaging. We have decided to also showcase the peer selection strategy in the *worst case*, which can occur in the asynchronous version of model averaging, where employing distinct peer selection strategies could balance the workloads on the network and on background request and serving tasks. APMA uses a fixed stratgy for peer selection - random.

Convergence

The experiments show that the implemented flexible synchronisation strategy Peer Model Averaging converges similar to the baseline. Results are shown for ResNet-32, ResNet-50. All strategies used for training ResNet-50 are deployed on Cluster Setup 1, while for ResNet-32 Cluster Setup 3 is used.

Figure 3.9 aims to compare convergence of the two implemented strategies (synchronous and asynchronous version of Peer Model Averaging) with TensorFlow Replicated. Figure 3.9a shows that APMA with two different peer selection strategies has similar convergence properties as TensorFlow Replicated. A small decrease in accuracy (2%) at convergence can be observed for APMA selection, being explained by an expected small deviation from the global model trajectory due to asynchrony. Figure 3.9b shows that SPMA similarly has the same convergence properties as TensorFlow Replicated (76%), with only a 1% decrease. It is interesting to observe in these plots that the early stages of training are unstable (high worker variance $\pm 5\%$ in accuracy and lower accuracy than TensorFlow Replicated over the first 10 epochs) for SPMA and APMA. This is expected, as workers train on different data shards and their pairwise model stabilisation requires time.

Figure 3.10a aims to compare convergence of APMA and SPMA, where the APMA version has distinct peer selection strategies (round robin and random) and SPMA has round robin, as



(a) (b)

Figure 3.10: (a) Sync Peer Model Averaging vs Async Peer Model Averaging with peer selection strategies over epochs (ResNet-50), (b) Peer Model Averaging vs Parallel SGD (ResNet-32)

discussed previously. The conclusion is that both flexible synchronisation stragies converge to similar accuracies: Synchronous Model Averaging (75%) and Asynchronoyus Model Averaging (74%). For diversity of exploration, convergence is also tested on the smaller ResNet-32 model (Figure 3.10b), where the baseline is the Kungfu implementation of Parallel SGD and the result proves that SPMA and APMA are robust irrespective of peer selection strategy. All strategies converge to an accuracy of 83%. The convergence results on ResNet-50 and ResNet-32 show that the new strategies are slightly sensitive to model size (lower than baseline by 1-2% for ResNet-50 and same as baseline for ResNet-32).

Performance Benefit

Next, we show the *training speed benefit* obtained by the proposed strategies. For ResNet-50, it is shown that the decentralized learning approach can improve the time to convergence. The comparison also brings focus on the difference between the two variants of peer-to-peer model averaging. As the asynchronous version is an optimization of the synchronous one, it is expected to achieve better performance, which is confirmed by the experimental result. As the peer selections strategy has very low overhead, performance is not affected by this factor.

Figures 3.11 and 3.12 aim to provide a comprehensive study of the performance benefit brought by Peer Model Averaging. First, we compare SPMA and APMA with TensorFlow Replicated and observe clear improvement in training speed (2.7 hours faster for SPMA and 4.1 hours faster for APMA). We also want to compare the two new strategies and this deserves an isolated comparison - APMA is faster than SPMA by 1.4 hours. Figure 3.12 clearly shows that APMA performs better than SPMA in terms of time to convergence. Now that the benefits are quantised, we can begin analysing *why* the plots show this improvement. First, we need to restate the mechanism by which TensorFlow replicated performs the aggregatation: this is a 2-thread (*pscpu/pscpu*) inter-GPU transfer via RAM (no NCCL), using a centralised approach that waits for all devices to send their gradients. In comparison to our approch, where there are less messages in the system (model request and model reply) and workers are allowed to train with very small synchronisation strategies, the Replicated approach has a strict barrier that incurs penalties. Now, to motivate why the synchronous version is slower than the asynchronous version, we refer mainly to the blocking training due to the lack of background tasks (as present in the asynchronous version).

Convergence under different conditions We also investigate the convergence properties of Peer Model Averaging in a multi-machine setting, on two DGX-1 machines (Figure 3.13). As a baseline, we use a version of Parallel SGD based on the TensorFlow Replicated strategy. The strategies are run for 30 epochs on ResNet-50 (checkpoints are done every two epochs). We report the maximum validation accuracy among all workers in the system. The synchronous version of Peer Model Averaging shows a slight improvement of training time (approximately 10 minutes). The target accuracy of 56.4% is reached 10 minutes faster when using Peer Model Averaging. However, Parallel SGD (Replicated) eventually trains to 60.06% accuracy. Training time reduction is the notable benefit for this short run. The result provides confidence that the

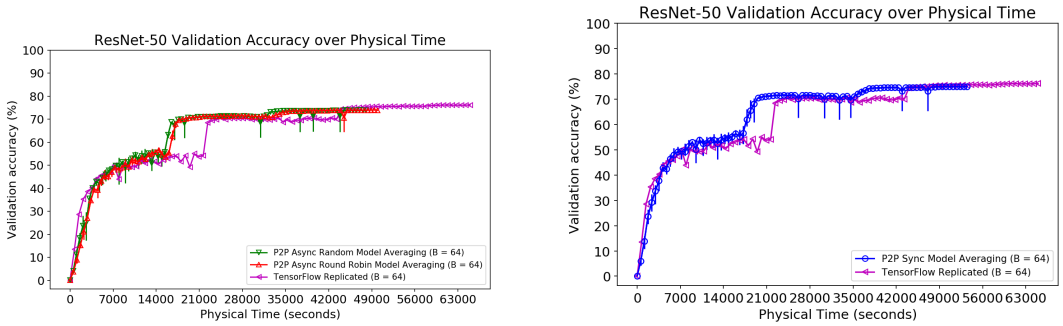


Figure 3.11: Peer Model Averaging strategies vs TensorFlow Replicated over physical time

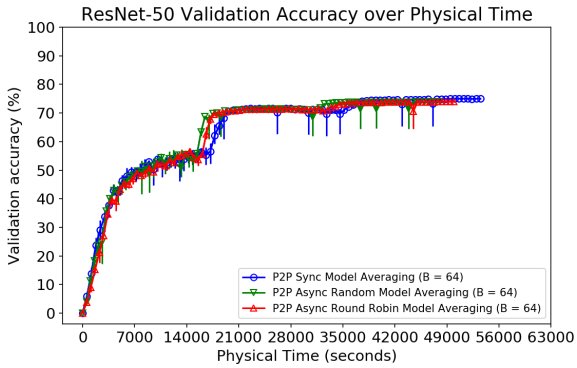
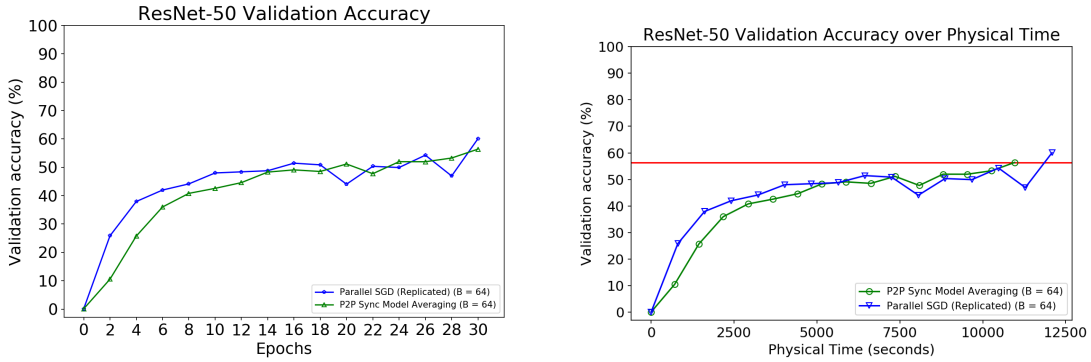


Figure 3.12: Sync Peer Model Averaging vs Async Peer Model Averaging with peer selection strategies over time



(a) (b)

Figure 3.13: ResNet-50 Validation accuracy on two DGX-1 machines (Peer Model Averaging vs Parallel SGD (Replicated))

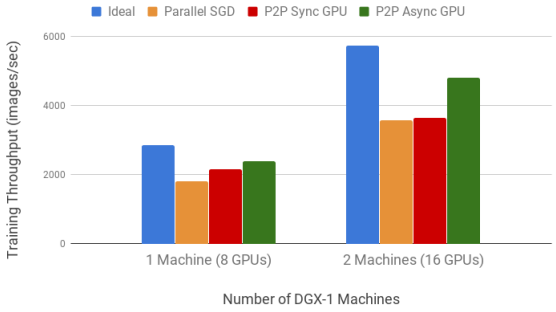


Figure 3.14: Peer Model Averaging scalability test on DGX-1 machines

system maintains convergence properties of the model even when synchronisation is done among 16 GPUs, maintaining the positive trend established by the single-machine run.

Scalability

It is important to understand the behaviour of the system in a setting with more workers. The bar graph in Figure 3.14 is a high-performance test of total (system-wide) training throughput for Synchronous and Asynchronous Peer Model Averaging on cluster set-up 1. The conclusion is that this approach leverages the hardware very well for training in the asynchronous case, which can be best quantified by the small difference (19% for both one and two machines) from the ideal throughput achieved by an independent learner.

The second bar graph in Figure 3.15 shows a similar behaviour at scale for the asynchronous version, when 16 machines are used. The system achieves linear scalability in the number of workers. The synchronous version of Peer Model Averaging shows *scaling difficulty* when the system is composed of a large number of workers. This difference is caused by the blocking training phase inherent to our design, aggravated by large network latency jitter. As measured in the 16-machine bare-metal cluster set-up 2 with Docker overlay network for inter-machine container communication, the request latency varies between 100 microseconds and one second. High variance indicates a heterogeneous environment where communication instability should be a decision factor when issuing requests. The penalties come from the tail of the latency distribution: even though frequency of large request latencies is low, this is significant enough to degrade performance of the system. The asynchronous version evades this problem through *non-blocking training* and *background tasks*.

Model averaging device placement

This subsection is a study of the effect of operator device placement for model averaging. Because flexible synchronisation strategies are introduced in a system where operator device placement is

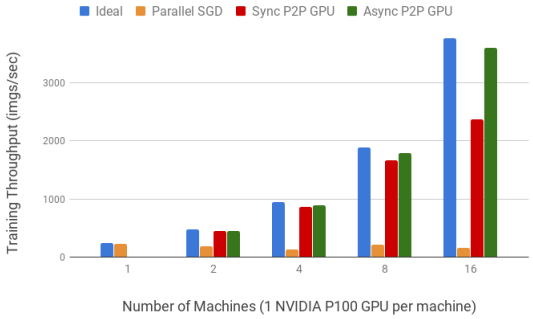


Figure 3.15: Peer Model Averaging multi-machine scalability test

a degree of freedom, it is important to understand how this can affect performance of the system. GPUs enable a high degree of parallelism in computation, but when flexible synchronisation strategies are enabled, the device placement for synchronisation becomes critical for training performance and is conditioned on factors such as the model size and hardware characteristics.

Sync P2P (CPU)	Parallel SGD (CPU)	Async P2P (CPU)	Ideal
2100 imgs/sec	2500 imgs/sec	3000 imgs/sec	3100 imgs/sec

Table 3.2: ResNet-32 training throughput

For example, training ResNet-32 with four workers (1 GPU per worker) on cluster setup 3 when CPU model averaging is enabled provides near-ideal throughput for asynchronous peer model averaging (Table 3.2). However, when training the larger ResNet-50 with more workers on more powerful GPUs (cluster setup 1), frequent data transfers between the GPU and the CPU create a bottleneck for training, decreasing training throughput per worker.

The result is also visible when training on 16 machines, each with one P100 GPU (cluster setup 2), where CPU execution of model averaging incurs performance penalties per worker (Table 3.3, first row compared to second row):

	Sync Model Averaging	Async Model Averaging	Parallel SGD	Ideal
CPU	87 ± 2 imgs/sec	136 ± 2 imgs/sec	10 ± 1 imgs/sec	235 imgs/sec
GPU	148 ± 32 imgs/sec	225 ± 2 imgs/sec	10 ± 1 imgs/sec	235 imgs/sec

Table 3.3: ResNet-50 training throughput per worker (16 machines, each with 1 V100 GPU)

The takeaway is that the TensorFlow approach of letting the user specify the placement of operators on GPU vs CPU can significantly affect training performance. As this degree of freedom is present in our system, the result provides a solid reason to allow the user to configure where performance-critical and frequent operations such as model averaging are executed.

3.5.5 Partial Gradient Exchange

Partial Gradient Exchange is one other component of flexible synchronisation that deserves exploration from the perspectives of performance and convergence. Evaluation shows that this strategy enables performance and degrades convergence of the model. However, the convergence gap through flexible techniques for enabling partial synchronisation can be covered through smarter synchronisation, as described in Chapter 4.

Convergence

The plot compares the convergence of ResNet-50 for Horovod, Parallel SGD and Partial Gradient Exchange. This plot is important for the algorithm evaluation in order to validate the key insight

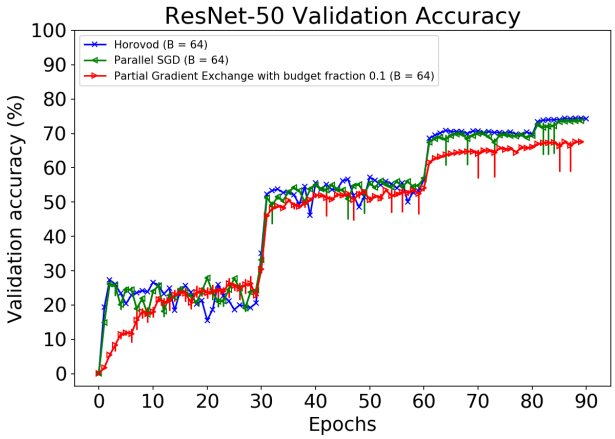
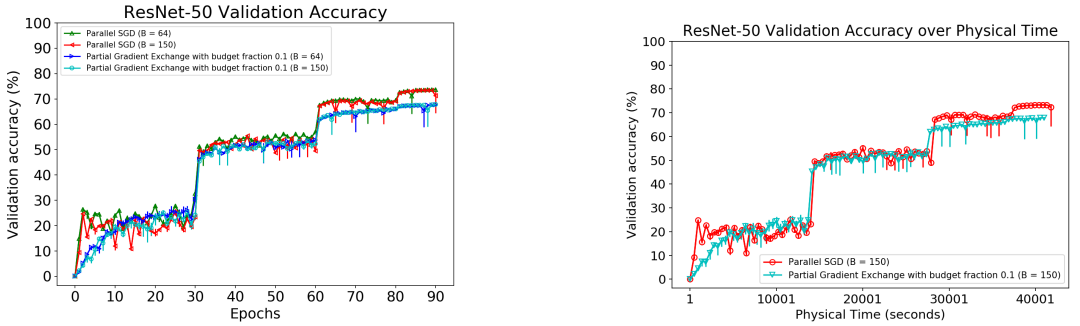


Figure 3.16: Partial Gradient Exchange convergence



(a) Validation accuracy over epochs

(b) Validation accuracy over physical time

Figure 3.17: Benefit of Partial Gradient Exchange in large batch conditions

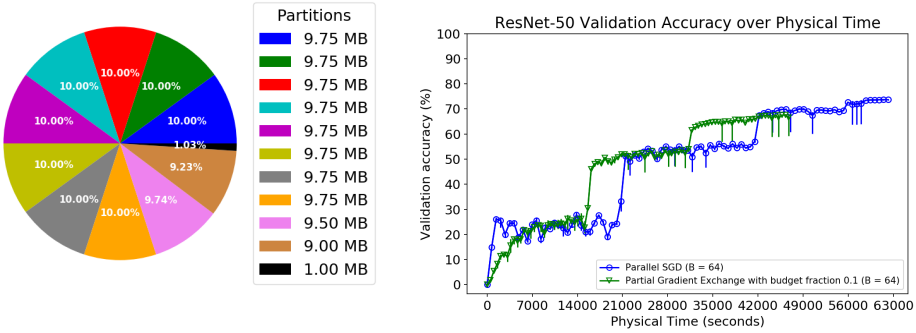
on which Partial Gradient Exchange is based: each peer has sufficient information locally to make progress and does not need to exchange its full gradient set. Experimental results (Figure 3.16) show that this insight overlooks some aspects important for training, as partial gradient exchange affects convergence by around 9% (Horovod and Parallel SGD 76% and Partial Gradient Exchange 67%). However, this gap can be closed by employing smarter synchronisation techniques to determine when partial synchronisation should kick-in, which is a new challenge for more flexible and effective partial gradient exchange.

3.5.6 Convergence under different conditions

Partial Gradient Exchange shows the same convergence properties when larger batch sizes are used. The time to accuracy, however, is much smaller in the large batch regime, where the *frequency of synchronisation decreases*, so the network is less likely to become a bottleneck. This is shown in Figure 3.17, where we first show that large batch regimes do not have an effect on convergence (Figure 3.17a), but they have a negative effect on training performance for Partial Gradient Exchange, where training reduction times are much lower (See small benefit in Figure 3.17b). It is fair to state that the breaking point of the Partial Gradient Exchange is large batch training and this type of synchronisation strategy should be preferably used when training with small batches.

Performance benefit

Partial gradient exchange provides a performance benefit when compared to Horovod and Parallel SGD. Compared to Horovod, the throughput increase is 5%, which is significant when considering lengthy training times spanning tens of hours. However, the comparison of interest is between Parallel SGD and Partial Gradient Exchange, which coexist in the same system called Kungfu. The aim is to show the effect produced by the change of algorithm in the system, while all other



(a) (b)

Figure 3.18: (a) ResNet-50 Partitions for partition budget fraction 0.1, (b) Accuracy improvement through coarse-grained Partial Gradient Exchange

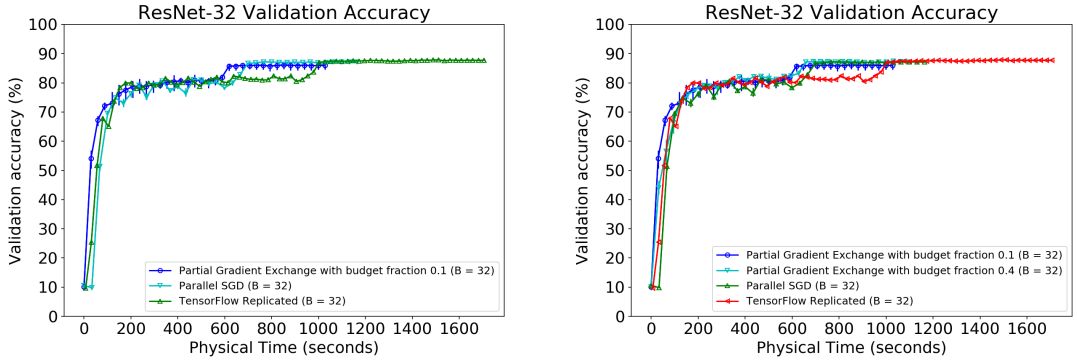
underlying components remain unmodified, the evaluation result compares the validation accuracy over physical time between Partial Gradient Exchange and Kungfu Parallel SGD. The 30% throughput increase (Figure 3.18b) is reflected in the plot of validation accuracy over physical time as an acceleration of training.

Figure 3.18a represents the distribution of model variables in the partitions negotiated in round robin manner every iteration. In a real setting, the user desires to achieve maximum performance and large generalisation power of the model. Introducing the budget fraction as a hyperparameter of the system can be difficult to manage by the user, as behaviour is model dependent. For example, the illustration presents the maximum number of partitions of the set of gradient tensors in ResNet-50. This consists of 11 partitions: 10 partitions are filled with convolutions and batch normalisation layers and one partition is filled with a large tensor variable corresponding to the fully connected layer. Exchanging these *smaller* partitions in round robin manner relaxes the synchronisation barrier, where only gradients belonging to one partition are exchanged and thus attenuates the large system barrier usually created by synchronisation.

In these experiments, training is done on 4 TITAN X (Pascal) GPUs from Cluster setup 3 and synchronisation operators use all available 20 CPU cores of the server machine. The findings in Section 3.5.4 about synchronisation operator device placement affecting performance and the difference in all-reduce mechanism explain the performance difference between the Parallel SGD and TensorFlow replicated baselines, where the latter uses direct sequential GPU aggregation. The effect of gradient partitioning is further explored for the smaller ResNet-32 model, which consists of the same blueprint architecture as ResNet-50, but contains only 32 residual blocks 2.1.1 which consist of tensor variables with smaller size. This makes it much easier to train for less complex tasks. Both plots show that employing partial synchronisation in smaller models does not affect convergence and provides a performance benefit compared to baseline systems. Using the same partitioning budget as in ResNet-50 yields the same gradient tensor distribution as in ResNet-32 3.20a. The distribution gives a training speed improvement of 10%, as the model is much smaller and the amount of network traffic is reduced.

To further explore the effect of Partial Gradient exchange in ResNet-32, a new experiment is run on Cluster setup 3 to test the hypothesis that less partitions capture the tendency of the algorithm to behave similarly to the Kungfu Parallel SGD algorithm, which belongs to the same underlying system. A larger partitioning budget of 0.4 ensures that three partitions are created. Although one of them is smaller, amounting for only 20% of the total size of tensors, it contains the large, fully-connected tensor variable, which makes its contribution significant for aggregation every three iterations. The plot (Figure 3.20b) proves that the hypotheses for ResNet-32 is tested: increasing the number of partitions in the system reduces training throughput and has a benefit on convergence. However, this approach is model-sensitive, as the hypothesis does not verify for larger models such as ResNet-50, where at scale, failing to incorporate big gradient sets can affect convergence.

The conclusion for Partial Gradient Exchange training runs on ResNet-32 and ResNet-50 is that although we obtain a performance benefit, skipping to negotiate gradients incurs accuracy penalties for large models. The new hyperparameter introduced is difficult to control without



(a) (b)

Figure 3.19: (a) Partial exchange with budget fraction 0.1, (b) Partial exchange with budget fraction 0.1

(a) (b)

Figure 3.20: (a) Partition sizes with budget fraction 0.1, (b) Partition sizes with budget fraction 0.4

sufficient knowledge of the Deep Neural Network architecture, but it opens up a new chapter for design of smart algorithms that can close the gap.

Scalability

This section presents the scalability results for multiple partitioning fractions when run in a two-machine GPU cluster (Cluster setup 1) with InfiniBand interconnect. In this case, creating network traffic is unlikely to reduce training performance, as the network has enough network bandwidth, the performance benefit is given by relaxed synchronisation barriers. The batch size used for training is 64 (unless otherwise specified in generalisability experiments), which is a small batch size enabling frequent synchronisation. The reason for choosing a small batch size is to analyse the behaviour when synchronisation is more frequent, in order to emphasize performance when synchronisation can indeed become a bottleneck for training.

One DGX-1 machine First, results are shown when running on one DGX-1 machine with 8 GPUs. According to NVIDIA, this type of machine provides the most advanced data center GPU built to date [36], which makes this test valuable to quantify performance of the system on the large ResNet-50 model. In Figure 3.21, the number local workers (GPUs) is varied and the scalability properties are determined based on gradient partitioning. TensorFlow Replicated and Parallel SGD show similar scalability properties. The trends from 1 to 8 peers can be quantified as follows: (a) TensorFlow Replicated shows increases in throughput by 136% from 1 to 2 peers, by 95% from 2 to 4 peers, by 25% from 4 to 8 peers and (b) Parallel SGD shows increases in throughput from 1 to 2 peers by 93%, from 2 to 4 peers by 92% and from 4 to 8 peers by 14.7%. There is a clear performance benefit given by Partial Gradient Exchange, quantified for 8 peers as a 47% increase

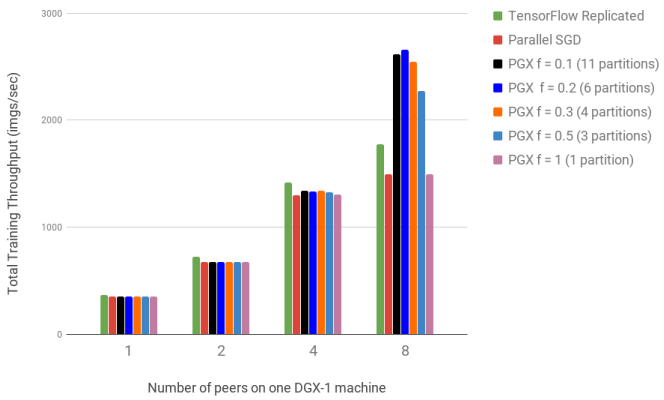


Figure 3.21: Partial Gradient Exchange scalability on one DGX-1 machine

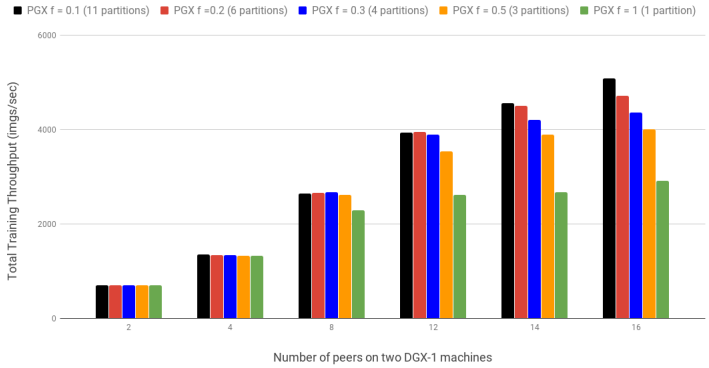


Figure 3.22: Partial Gradient Exchange scalability on two DGX-1 machines

when compared to TensorFlow replicated and by a 75% increase from the throughput achieved by Parallel SGD. One important observation is that Partial Gradient exchange with 1 partition behaves exactly the same as Parallel SGD.

Two DGX-1 machines On two machines, the number of workers is varied. There is an equal number of workers on each machine. Similarly, the system scales linearly in the number of workers, with better performance exhibited by more granular partitioning. We observe in Figure 3.22 that more granularity in gradient partitioning provides increased performance: (a) when moving from $f = 0.3$ (4 partitions) to $f = 0.1$ (11 partitions), the throughput gain is 16% (b) when transitioning from $f = 0.5$ (3 partitons) to $f = 0.1$ (11 partitions), the throughput gain is 27%. The conclusion of this experiment is that Partial Gradient Exchange performs well at with a large number of workers conditioned on the partitioning budget, so the adopted direction is worth exploring for achieving high-performance in multi-resource Cloud environments.

We further analyse the performance of Partial Gradient Exchange in a high performance cluster (Cluster set-up 1) and base the discussion on Figure 3.23. We can observe that high-performance GPUs and good network conditions enable near-linear scalability when NCCL is enabled. The environment is controlled and the system can very well leverage its compute power. When synchronisation is done on the CPU, however, small performance penalties are incurred by frequent GPU-to-CPU tensor transfers. Parallel SGD, which has strict synchronisation barriers has difficulties scaling even in this environment. On one machine, synchronisation using partial exchange provides a 36% throughput increase from Parallel SGD case when synchronisation is done on the CPU and a 45% increase when using NCCL. On two machines the increase are quantified as 45% (CPU) and 55% (NCCL), when compared to Parallel SGD.

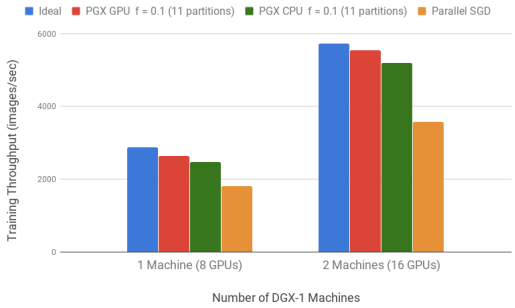


Figure 3.23: Fraction exploration: Partial Gradient Exchange scalability on DGX-1 Machines

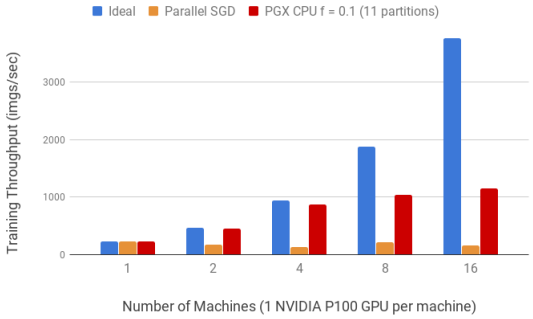


Figure 3.24: Partial Gradient Exchange CPU scalability 16 P100 Machines

16-Machine Cluster When the network conditions change and the performance of GPUs is lower (Cluster set-up 2), we can observe in Figure 3.24 that Partial Gradient exchange *does not* scale linearly in the number of workers. With the underlying Parallel SGD paradigm of collective communication, the barrier relaxation is not sufficient to enable the system to scale. This constitutes the *breaking point* of Partial Gradient Exchange, where the scaling properties are determined by the amount of gradients exchanged in an iteration. This is a largely model-dependent problem and represents a limitation for the designed system.

3.6 Summary

Flexible synchronisation strategies represent an enabler for systems to scale in cluster environments with a diverse range of hardware characteristics and network links. From individual multi-GPU machines and public resource pool commodity machines in the cloud to latest generation GPU and network accelerated clusters, flexible synchronisation strategies pave the way for scalability of Deep Learning training.

These benefits are conferred by a simple, yet robust design that provides the user with a rich set of options for flexible synchronisation enabled for one of the most popular Deep Learning systems of the present, TensorFlow. This was possible first by defining the system gap in TensorFlow, which provides rigid means of synchronisation such as parameter servers and MPI-based approaches. We identified the need to cover this gap by a high-performance and flexible system which can incorporate multiple synchronization strategies.

The intuition for designing such a system was that of partial synchronisation, which is based on a key insight of training: each peer has sufficient information locally to make progress. The advantage of designing such a system is that of determining smartly when specific synchronisation strategies need to be enabled. The main contributions are: (1) creating an abstraction for the user by creating new TensorFlow optimisers, shown to suffice simple specification of flexible synchronisation (with single partitioning scheme, single peer selection strategy) and (2) creating a high-performance communication framework with a lower level of abstraction that allows workers to exchange gradients and variables through partial collective communication and point-to-point communication.

First, the design proposes a transparent way of enabling flexible synchronisation by intercepting variables and gradients with *new wrapper* optimisers. This design enables exploration of decentralised training through: (1) gradient exchange and (2) model exchange between peers. For (1), we show how PGX we propose a formal algorithm for partial exchange and provide a concrete implementation as a TensorFlow dataflow graph with support for CPU and NCCL collective all-reduce. For (2), we propose an algorithm for synchronous and asynchronous *selective and unidirectional* peer model exchange and a concrete implementation in TensorFlow. The contribution here is three-fold: (a) providing configurable ways of realising these strategies (CPU, GPU), (b) detailing design optimizations that lead to asynchronous training, which can be generically applied by future systems and (c) building a new level of abstraction for point-to-point communication to leverage more granular a more granular cluster abstraction where peers can bidirectionally any data.

All strategies are evaluated in a high-end Cloud environment [41] which allowed analysis of convergence and scalability properties of PGX and SPMA. We have quantified performance benefits in comparison with industry-standard distributed training strategies such as Horovod [5] and a benchmark TensorFlow's version of P-SGD called Replicated [116, 117]. Experimental results show that by enabling the new strategies we manage to improve training performance to state-of-the-art accuracies and by enabling partial gradient exchange, the model generalization power decreases by 9%. We further incorporate active network and gradient noise monitoring (based on [118]) for online estimation of training quality and propose a direction to guide creation of dynamic synchronization policies such that the model inference gap can be covered.

Chapter 4

Supporting Users to Choose the Right Synchronisation

By far we have described a scalable implementation of a flexible synchronisation system and shown that this system can help DL models to accelerate training. However, there is a key question remained: given rich options of synchronisation, which synchronisation should be used? In this chapter, we try to provide comprehensive system support for helping users make the right choice. To achieve this, our key idea is to provide online system monitoring for both communication infrastructure and training statistics. Based on these two kinds of complementary monitoring metrics, we can further provide support to help users change the configuration of a synchronisation algorithm, and even switch to a new synchronisation. In the following, we discuss how we implement this idea and evaluate the proposed implementation.

4.1 Design principles

In this section, we introduce two key design principles that can lead to the anticipated system support for choosing synchronisation.

Monitoring of network and training statistics To evaluate a synchronisation algorithm, DL users often rely on network statistics and training statistics. The former statistics help describe the available network resource and the communication performance, implying the severity of network bottlenecks. The latter statistics is useful for determining the progress of a training job and thus help DL users, for example, adjust the synchronisation frequency. These two kinds of statistics are often complementary and must be provided at the same time. For example, when configuring the bin packing ratio of the partial gradient exchange algorithm, DL users not only need the network statistics to estimate the latency of exchanging a certain amount of gradients. They also require statistics that can reflect the convergence status of gradients so that the algorithm can correctly filter out gradients that have converged when bandwidth is limited.

Supporting the adaptation of synchronisation parameters. With monitoring data, we question how to leverage these data to improve the performance of synchronisation. Synchronisation algorithms often have hyper-parameters, e.g., the bin packing ratio of the partial gradient exchange algorithm and the number of peers to participate in model averaging in the EA-SGD algorithm, [56] that affect both hardware utilisation and learning behaviours. The optimality of these hyper-parameters are often determined by environment, model and dataset: resource allocation for parameter servers [33], `HOROVOD_FUSION_THRESHOLD` byte count for the capacity of the tensor set to be batch all-reduced in the Horovod Tensor Fusion Operation [5, 98], TensorFlow all-reduce specifications using a flexible BNF grammar [117] specifying how all-reduce should be executed depending on device support. The optimal settings of these hyper-parameters are often *adaptive*, similar to other hyper-parameters like learning rate and batch size which are usually dynamically adjusted during training to improve training performance [114, 118]. As a result, DL users expect the training system to support the adaptation of synchronisation parameters during training, and ideally, incurring negligible performance overheads.

4.2 Network statistics monitoring

The communication performance of a synchronisation algorithm can be determined by both the global execution barrier and the actual traffic that travel through network links. In order to precisely evaluate the performance, we aim to provide global and local monitoring for network usage.

Monitoring metrics Network monitoring is a field that has been extensively studied. From the perspectives of reflecting the network performance as a whole, availability metrics describe the robustness of a network and refer to the state of hosts, devices and links found in a network that can potentially affect the quality of service (QoS) [119]. From the perspective of evaluating link-wise quality, the maximum amount of data that can be sent through the network in a time unit is called *bandwidth* [119]. However, during transmission losses may occur (e.g., possibly due to routers dropping packets or switches dropping data frames), so the link is not fully utilised. The amount of data that can be sent through the network in a time unit is called *throughput*. It is also important to quantify segment losses by counting out of order TCP segments, as this gives a good measure of network conditions (e.g., congestion, parallel paths) [119]. Link quality can also be measured as the amount of time necessary for one packet to be transmitted between two fixed points in a network - this is called latency and largely depends on congestion status, delays due to packet processing along the path or even transmission media.

Existing monitoring options Fast advances in network monitoring create new state-of-the art techniques that increase the complexity of network monitoring. Of interest to us are data center networks, where operators have shifted to Software Defined Networking (SDN) for simplifying network management at data center scale. The new metrics provide a holistic view of the network through granular query interactions with the data plane to gather network metrics [120]. Adaptive querying techniques have been developed to improve how monitoring is done and to tackle granularity: *threshold-based* adaptation, *prediction-based* adaptation and newer self-adaptive techniques [120] which do not require manual tuning. Dedicated software products have been developed to handle monitoring infrastructure in systems [121] and these often involves hundreds of giga-bytes of monitoring data represented as a *time-series database* [121]. Modern operating systems also come with a wide range of network monitoring tools that rely on blocking packet captures. Some provide immediate accessibility to the user and provide high-level network usage information iftop [122] and others present network information at packet level Wireshark [123]).

Though easy, adopting *host-level* network monitoring tools out of box in Kungfu, however, is not effective. These tools often provide metrics that down to the packet level and must enable disruptive traffic inspection within the OS. This is realised by low-level network libraries which are part of the OS kernel and allow users to register hooks for packet processing in user-space [124]. These handlers can be blocking (Netfilter Queue [124]) or can instead pass packet copies to user-space (Netfilter Log [124]), but in any situation the overhead created is considerable. Adding a new level of indirection in user-space is not acceptable for DL applications which require high-performance.

Integrating monitoring with DL synchronisation The packet-level metric is often too granular for a synchronisation algorithm which often adopts collective communication and high-level point to point communication (i.e., requesting remote model). Its network performance can be only judged by taking application semantics into account. More importantly, using existing network monitoring tools often comes with heavy performance overheads for large clusters of hundreds of nodes each with hundreds of metrics (e.g., Prometheus Instrumentation Costs [121] or Netfilter Log/Queue user-space overhead [124]). These non-negligible overheads are particularly detrimental in the case of monitoring synchronisation algorithms. Synchronisation algorithms, such as asynchronous parallel SGD [39] and asynchronous peer model averaging, often adopt asynchronous designs to tolerate communication latency and enable model replicas to diverge in order to improve training accuracy. The extra non-negligible overhead on the data transmission path could significantly change the network performance under monitoring and non-deliberately alter the behaviours of asynchronous algorithms, making the resulting monitoring data less useful to guide the configuration of the algorithm when monitoring is disabled.

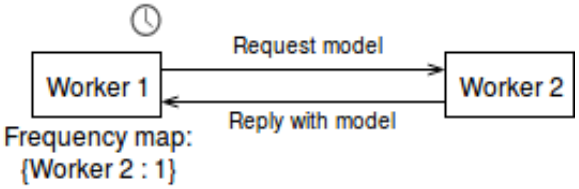


Figure 4.1: Overview of new network monitoring features for direct peer communication in Kungfu

Enhancing the Kungfu network stack As a result, in this project, we aim to implement *application-specific light-weight network monitoring* within the Kungfu worker implementation. In order to do this efficiently, we have decided to enhance the communication stack. This separates the application logic realised by C++ implementations of TensorFlow operators (kernels) from the metrics monitoring logic. The trade-off comes from the CGo interface which can enable specification of monitoring configuration using the internal API exposed to the C++ kernel classes.

The contribution of this work is building a new point-to-point communication primitive, which is complementary to the existing collective communication primitives for all-reduce provided already in the Kungfu project. The CGo API exported by the networking stack allows direct usage of functions in the C++ `tensorflow` namespace, where they can be directly used by the operators responsible for model requests. The exported API consists of one Go function:

```
GoKungfuRequest(peer int, model_buffer unsafe.Pointer, count int, dtype C.KungFu_Datatype, callback *C.callback_t)
```

Listing 4.1: CGo API for model requests

Design of the peer-to-peer communication protocol In order to explain how monitoring is enabled, we first explain the communication protocol. The design decisions that lead to the final API are mainly lead by designing a valid protocol for point-to-point communication. The protocol is valid if Peer i sends a message to Peer j and registers a listener such that it is then able to receive messages from Peer j. The first considered solution for implementing this protocol is to register a callback to handle the request in the background. This means that the first TCP connection used to send the request is closed and the receiver reopens a new TCP connection to send the reply. One obvious drawback is sub-optimal network usage, due to new handshakes required to establish the TCP connection. The main drawback of this approach is contention on objects such as the model store when callbacks are triggered aggravated by unpredictable calls to the TensorFlow operator (kernel) scheduled by the runtime we have no control over and for which there is no guaranteed compatibility with default synchronisation primitives (mutexes, semaphores, condition variables). Therefore, the solution maintains flow control by reusing TCP connections. Background request tasks (blocking) are dispatched as goroutines to receive the reply on the same connection.

Implementation of light-weight monitoring in Kungfu The networking stack has full control of concurrent execution and of the communication flow. This creates a proper setup for acquiring metrics for the quality of communication. The `destination_peer` is the randomly chosen peer where the request is directed, uniquely identified in the cluster abstraction. The reply is written to the model buffer using required metadata (count and data type). A background request task is spawned. When this terminates, a termination signal is generated through a callback. The request latency measured is the *time necessary to complete a request command* using a single TCP connection for bidirectional communication 4.1. A frequency table is built after every outbound request succeeds. This keeps track of frequency of communication with other peers.

4.3 Training statistics monitoring

The training quality in a distributed training system is determined by the progress of the learning algorithm per iteration, also known as *statistical efficiency*. This can be quantified by observing statistical properties of gradients. In order to enable a holistic view of the system, we aim to integrate efficient online monitoring of training statistics in the Kungfu system.

Training statistics and their possible applications in synchronisation. Training statistics capture the quality of training and there are two views on how these can be computed, a *temporal view* and a *spatial view*. The *temporal* view is the local view of a worker, which can compute variance of previous gradient tensors and decide when it is a good time to synchronise them with other workers (similar approach to [125]). It is important to clarify the meaning of variance when considering tensors of the same dimensions. As tensors are multi-dimensional, they need to be flattened into vectors. The statistic computed on vectors is the co-variance matrix, which intuitively captures how aligned are the values across the sample vectors. The variance metric of interest in this case is the aggregated statistic obtained as the trace of the co-variance matrix, which represents the sum of diagonal elements. When the variance is small, meaning that previous gradients are aligned, the peer should send its local gradients for synchronisation. When the variance is large, local gradients should only be applied locally. A sliding window can be used. However, in this approach a worker never regards the training quality of other workers, being responsible only for sending its local gradients, approach likely to lead to divergence. The *spatial* perspective considers to quantify the variance between its local gradient and other peers' local gradients. This is hard to achieve efficiently without centralisation, but there are efficient computations of metrics such as gradient noise, which exhibit predictive properties. *Gradient noise predicts the critical point of diminishing return for the largest batch size that the system can use to find the trade-off between statistical efficiency and hardware utilisation* [118]. These predictive properties can be similarly leveraged by synchronisation for devising adaptation policies.

Main requirement of monitoring in DL systems Robust monitoring schemes for distributed training systems should provide *efficient* monitoring at local and global levels. These need to be computed in a decentralised manner and if needed, global metrics should be expressed in a collective communication paradigm like all-reduce.

Gradient noise in distributed training Because we are considering a data-parallel distributed training system, the spatial view is more suitable and the only way to achieve computation of a global variance metric is by piggybacking computation on existing communication, such that additional traffic is not generated. McCandlish, et al. provide a solution in the paper titled *An Empirical Model of Large-Batch Training* [118]. This solution is based on a training statistic called gradient noise which can be efficiently approximated.

The *key intuition* behind gradient noise is that it represents a noise-to-signal ratio (mathematically), where the noise quantifies the dissimilarity between locally computed gradients and globally aggregated gradients and the signal is the magnitude of updates - how much new information is the worker incorporating after training on a new batch. The is essential for developing an understanding of how gradient noise can serve synchronisation. Of interest is the stability of training, which can be easily quantified using the gradient noise. When the gradient noise is large, the gradient signal is low and vice-versa. Therefore, we can infer that training becomes more stable when the gradient noise is large, because the magnitude of updates is small meaning that learning plateaus. This moment can be exploited by flexible synchronisation techniques such as Partial Gradient Exchange.

Implementation of gradient noise monitoring in Kungfu The implementation relies on the approximation tailored for distributed training proposed by OpenAI [118], detailed in Appendix A.3. This presents the computation of the noise-to-signal ratio captured by the formula [118]:

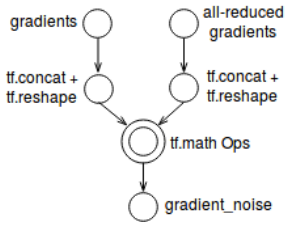
$$B_{simplified} = \frac{tr(\Sigma)}{|G|^2}$$

where Σ is the co-variance matrix of the gradients across workers and the denominator represents the square of the L_2 norm of the gradient vector $|G|^2 = \sum_{i=1}^{D} |G_i^2|$. One of the challenges was the computation of $|\mathcal{G}|$ from two sets of gradients: the locally computed gradients (denoted as $G_{B_{small}}$) and the the set of all gradients (denoted as $G_{B_{big}}$) obtained via all-reduce. The proposed solution is to concatenate and flatten the gradients in each of these lists, obtaining $G_{B_{small}}$ and $G_{B_{big}}$ respectively. This can now enable computation of the biased estimates.

```
def all_reduce_with_monitor(grads, b_small, n_workers):
    """Intercept local gradients for all-reduce and noise monitoring."""
    negotiated_grads = [all_reduce(t) for t in grads]
    G_b_small        = _concat_reshape(grads)
    G_b_big          = _concat_reshape(negotiated_grads)

    noise_op = build_noise_op(b_small, G_b_small, b_big, G_b_big)

    with tf.control_dependencies([noise_op]):
        return tf.group(negotiated_grads)
```



Listing 4.2: TensorFlow dataflow specification and dataflow graph

The Listing 4.3 and the accompanying conceptual dataflow isolate the implementation contribution. These show how the abstractions enabled by our system are used to create an efficient trainining monitoring framework. In order to implement the computation formally defined above, the TensorFlow API is used to compute only the biased estimators of the gradient variance and the gradient norm respectively. These estimators are not sufficient for computing an unbiased estimation of the gradient noise. The paper [118] indicates that separate exponentially moving averages (EMAs) should be computed. This requires state keeping across iterations, which best translates into the need to create a new TensorFlow operator in C++. Its functionality is rather simple and involves computing the EMAs and the noise-to-signal ratio that yields the final approximation of the gradient noise.

We have shown how the implementation of the gradient noise is expressed in the dataflow paradigm, isolating the contribution. The computation is achieved locally by each worker using local gradient information and aggregated gradient information. This provides each worker with statistic useful for further control of the training process (e.g., for a possible adaptation scheme conditioned on gradient noise values).

4.4 Adaptation of synchronisation parameters

Existing systems such as TensorFlow provide support for hyper-parameter adaptation during training. Most commonly, adaptation techniques are required for hyper-parameters responsible for driving accuracy of the model, such as the learning rate. Learning rate adaptation requires mathematical understanding of the model and schedules are often predefined or explored by tuning. Tensorflow enables the user to define specific epochs/iterations for changing hyper-parameters into new constants or for decaying hyper-parameters throughout training [126]. This is realised through a simple API that requires only the boundaries of training and the specific changes desired for the hyper-parameter. The key limitation of the TensorFlow approach is lack of expressivity, as it considers changes in tensors only, disregarding how complex configuration changes can be otherwise transmitted to the system. Moreover, the approach may not fit in a system where adaptation is done based on conditions that locally characterise the worker's state only. We propose a new system design with *generic* applicability to dynamic runtime changes in TensorFlow programs. For this purpose, we use Partial Gradient Exchange as a concrete example of how dynamic changes can be achieved for hyper-parameters specific to synchronisation.

Overview One requirement of the system refers to low cost hyper-parameter changes. Meeting this requirement provides more flexible behaviour which can be controlled by user-specified policies. This constitutes a preliminary step in building adaptive algorithms. Dynamic hyper-parameter changes are implemented for Partial Gradient Exchange, extending the static partitioning scheme by schedule-based re-partitioning. People have used schedules for distinct hyper-paramters of training algorithms to release full learning capabilities in Neural Networks. These are often empirically determined [114] and have limited applicability across wider families of models, but their narrow effect is a beam towards better learning.

Towards generic adaptation In order to provide schedule support in Partial Gradient Exchange, it is important to identify where this could fit within the system, starting from the highest level of abstraction. The API is a non-invasive wrapping optimizer. This requires a string matching a simple regular expression: `(EPOCH:FRACTION)(;EPOCH:FRACTION)*`, which is called *schedule*.

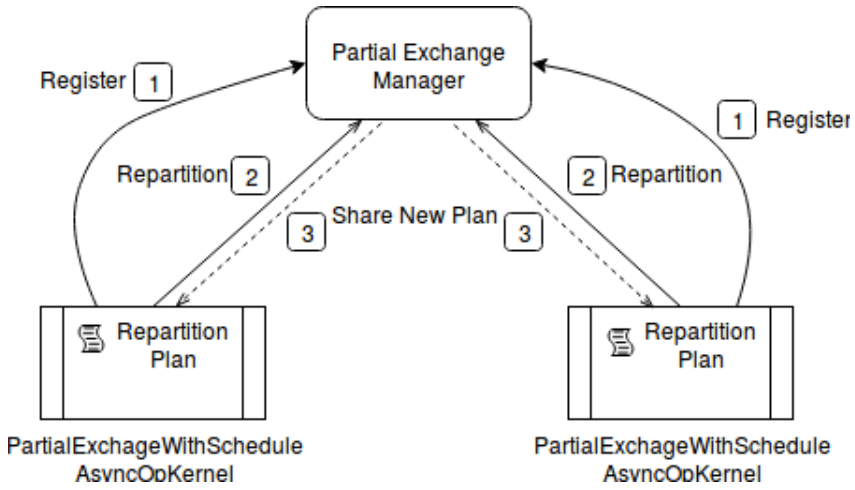


Figure 4.2: Dynamic partitioning for Partial Gradient Exchange

This indicates the ranges where a specific partitioning budget fraction is enabled. This means that there exists a partitioning scheme enabled for pre-defined intervals, which controls the number of gradients synchronised at every iteration. This flexibility is enabled through the lower system abstraction responsible for intercepting gradients. The TensorFlow API does not easily support *generic* design for periodic runtime changes. TensorFlow is built around the dataflow paradigm, which builds all operators before executing the computation graph. This is analogous to compilation of the program logic for every training iteration, such that the policy can be enabled by directly using TensorFlow operators. There are multiple drawbacks to this approach: (1) all operator calls happen on the critical path of training, (2) the number of dependencies for execution order of operators creates execution barriers, which can last as long as an all-reduce, as empirically determined and (3) further support for adaptation is impeded by the rigid datflow model.

A new partial negotiator The proposed solution relies on a flexible design which can be fully controlled. The logic is written as a C++ AsyncOpKernel. This decorates the behaviour of the base AllReduce operator. There is one such operator for every gradient tensor, in order to enable dispatch of background all-reduce tasks in parallel for all tensors, such that the critical training path is not blocked. To understand the mechanism for partial negotiation via all-reduce within the C++ operator, the simple schedule `0:0.1` is considered. This specifies that bin packing is executed once at the beginning of training (beginning of epoch zero, equivalent to iteration 0) - bin packing algorithm implementation is adapted from [111]. As discussed in the previous description of Partial Gradient Exchange, the global step is used for selecting which partition is negotiated in a round robin manner. Each operator can efficiently query an entity called the Partial Exchange Manager to learn if its gradient tensor belongs to a partition which requires negotiation. If it is the case for negotiation, the operator is responsible for returning the all-reduced tensor, otherwise it simply returns its unchanged input representing the locally computed gradient.

Runtime adaptation logic Extending the use-case for a more complex schedule which specifies more than one repartition, we can describe the full interaction between operators and the Partial Exchange Manager. Initially, each operator registers its tensor meta-data with the Partial Exchange Manger: tensor name, tensor size in bytes. They also inform the Manager what is the user-specified schedule, in parsed form (i.e., where epochs are converted to iterations). The schedule is globally unique and is initialised only once through the first registration message to arrive. The Plan is a class containing the *next re-partitioning step* as an integer and the *collection of partitions* as a vector of unordered sets (C++ standard data structure with constant time retrieval and insertion). Note that the Plan does not hold any memory-heavy data structures. The partitions contain tensor names only. The Plan is always shared by the Partial Exchange Manager with the follower operators. The main role of the Partial Exchange Manager is to supply a concise re-partitioning plan. The Plan can be efficiently implemented by all operators. These are called once during every training iteration. They are responsible for checking their local plan and if the global step has not reached the next re-partitioning step specified by the plan, partial negotiation

is realised on the current plan in effect. If re-partitioning should happen at the current global step, all operators notify the Partial Exchange Manger that re-partitioning is necessary. The interaction is summarised in Figure 4.2.

4.5 Use case: adaptive synchronisation based on training statistics

Leveraging flexible synchronisation strategies often comes at a cost in DL model quality. Such strategies require system-provided guidance for hyperparameter settings. We describe next the motivation to use gradient noise as a guiding metric for Partial Gradient Exchange. We then analyse the feasibility of approximating gradient noise for the ResNet model family and use the empirical observation to devise a schedule that closes the convergence gap for Partial Gradient Exchange.

4.5.1 Key idea

Fundamentally, Partial Gradient Exchange is based on the intuition that each peer has sufficient information locally to make progress and can only synchronise a subset of gradients which should suffice a global approximation of the true gradient at that iteration. As shown in the experimental results, this intuition does not enable complex models such as ResNet-50 to converge to high accuracies, because it maintins a constant rate of change of gradients throughout training. This is a *monotone* approach to Partial Gradient Exchange that *fails to leverage the current status of training* for decisions on partial synchronisation.

To address the above issue, the key idea is to leverage gradient noise to measure the noise of gradients in different stages of training, and optimise the configuration of the bin packing ratio in the partial gradient algorithm. The intuition here is: gradient noise as a statistic for the quality of distributed training is drawn from its basic mathematical property: it represents a noise-to-signal ratio (refer to Appendix A.3) which has low values at the beginning of training, correlated with more intense learning (higher gradient magnitudes) and which has high values at the end of training, correlated with a learning plateau. The aim is to synchronise fully in the beginning of training, in order to enable more meaningful gradient updates and to relax the synchronisation barrier and periodically rely only on the locally computed gradients, which provide sufficient information to make progress, in the later stages of training. Experiments have been conducted to analyse the properties of gradient noise, in order to conclude their relevance as guiding metric for synchronisation.

4.5.2 Dynamic policy design

This is an analysis of gradient noise properties aimed to check whether the intuition that gradient noise increases during training verifies in a real practical setting. The aim is to find good monotonous and increasing estimates that could steadily guide adaptation policies for flexible synchronisation. Observations of these estimates are used to empirically guide design of a fixed (user-specified) policy for the Partial Gradient Exchange partitioning scheme.

Related work: adaptation based on gradient noise The online prediction properties of the gradient noise have been previously analysed [118] in the context of adaptive batch training. Building on the previous argument, the authors argue that gradient noise provides a good approximation for a dynamic training hyperparameter called *critical batch size*, which aims to satisfy two constraints simultaneously: maximise hardware utilisation throughout the whole training duration and maximise the loss decrease per iteration for more effective learning. Training with a big batch size decreases the number of updates made to the parameters of the neural network [27], but it is hard to manually define good general policies for batch size increase without affecting convergence of the system.

Experimental setup Experiments are conducted to verify the claims made in [118] with the implementation made in our system. We use one server machine dedicated with 4 GeForce GTX

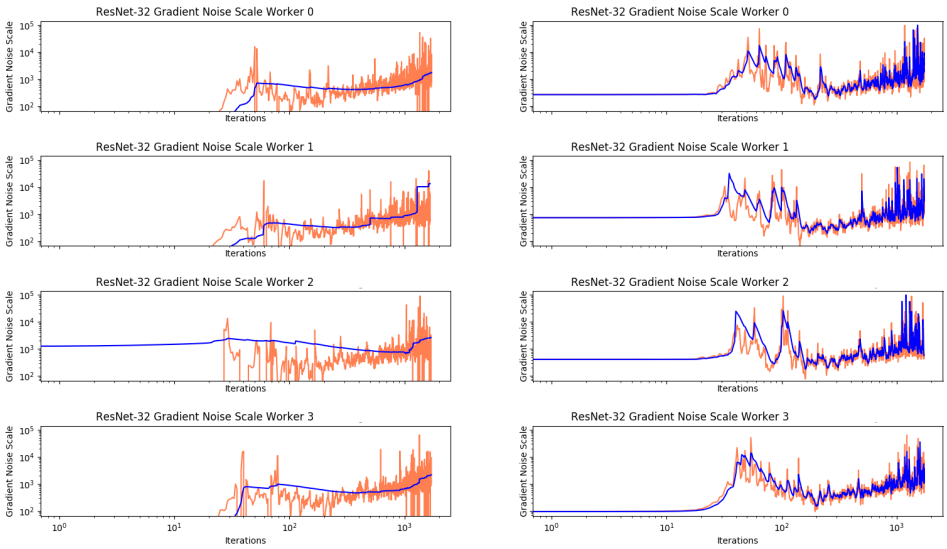


Figure 4.3: Gradient noise estimation for ResNet-32: raw gradient noise (orange), estimated critical batch size(blue). First column shows local worker estimations using running average over window of 100 iterations. Second column shows local worker estimations using exponentially moving average with decay $\alpha\prime = 0.2$. Gradient noise estimated using decay $\alpha = 0.8$. Warm-up batches discarded from noise estimation window.

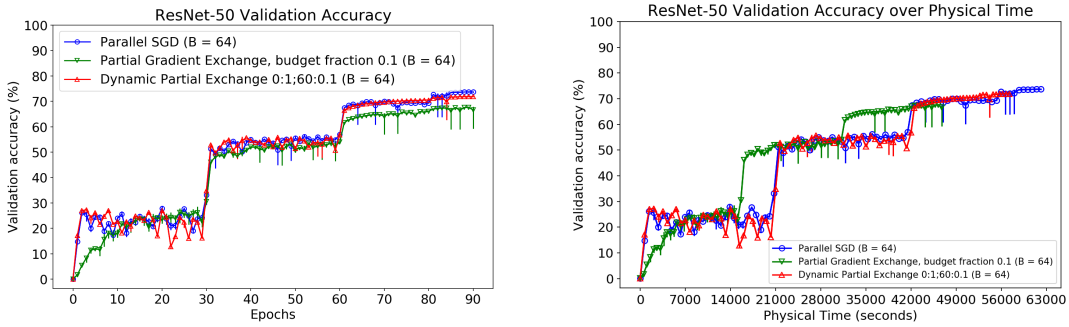
TITAN X (Pascal) GPUs and 20 Intel(R) Xeon(R) CPU E5-2640 v4 @ 2.40GHz (Cluster Setup 3). All experiments are run with four workers, each training on one GPU.

Smooth online estimation in Kungfu The first step involves obtaining the raw gradient noise (Figure 4.3), for which the implemented system provides direct support. However, the raw signal shows high variance. This is controlled by the decay parameter mentioned in the paper: small decay provides less variance. However, using this high level-approximation is biased to consider the noise at the current iteration with higher weight or otherwise to discard most of the noise at the current iteration. Therefore, another level of indirection is introduced for estimation. The blue line in Figure 4.3 is a smooth estimation of the critical batch size from gradient noise, as described in the paper. We compare two methods for obtaining this estimate: running average across a window of iterations or EMA on gradient noise values. As shown in the plot, the running average is smoother than the EMA for one particular setting of window and decay factor, but both require additional tuning for controlling desired curve shape.

Remarks on gradient noise for synchronisation The experiment shows that our system can efficiently do online estimations of training statistic metrics based on simple scalar computations. The guiding trend for adaptation captured using another light-weight estimation layer supported by the Kungfu system. The monotonous increase of gradient noise can be captured using the realised implementation and its value used for devising adaptive synchronisation techniques with awareness of the training progress.

The increase in gradient noise is common within and even across families of models [118]. As this work has shown results for the ResNet family, the further concern is improving the convergence gap obtained by Partial Gradient Exchange using ResNet-50. We can thus leverage the knowledge of gradient noise increase and determine an intuitive policy to *boost* training.

Policy design Training becomes more sensitive when partial exchange is enabled for large models. This strategy provides performance benefits, but degrades convergence. This means that Partial Gradient exchange should be selectively enabled during training in order close the convergence gap. In order to do this, we can use gradient noise to guide adaptation of hyperparameters



(a) Dynamic Partial Exchangence convergence

(b) Dynamic Partial Exchange over time

Figure 4.4: Closing the gap with dynamic partial exchange

introduced by flexible synchronisation. The gradient noise represents a noise-to-signal ratio. Because gradient magnitude decreases during training, the gradient noise will increase, meaning that training becomes stable. This could be a good moment to enable Partial Gradient Exchange. But how can this threshold be precisely determined? We use empirical observations to design a dynamic policy for Partial Gradient Exchange in order to test whether enabling this flexible synchronisation strategy late during training closes the convergence gap while still obtaining the benefit of faster training times. The policy designed changes the bin packing fraction from 1 to 0.1 at epoch 60 during training. This is equivalent to exchanging all gradients during the first 60 epochs, then only exchanging 0.1 of the total gradient set size.

4.5.3 Experimental result

This subsection is an extension to the evaluation result presented in Chapter 3 for Partial Gradient Exchange and shows that it is possible to close the convergence gap through policies guided by empirical metrics. This represents a starting point in developing adaptive algorithms based on monitored metrics. Figure 4.4 shows that using the designed policy, the accuracy gap is closed by approximately 5% (initial accuracy gap of approximately 9%) for the best performing workers. The validation accuracy achieved by the baseline is 76%, while simple Partial Gradient Exchange achieves 67%. With the new policy in effect, the validation accuracy is 71.9%. Moreover, training with a dynamic policy obtains a training time improvement of approximately 2 hours and 45 minutes.

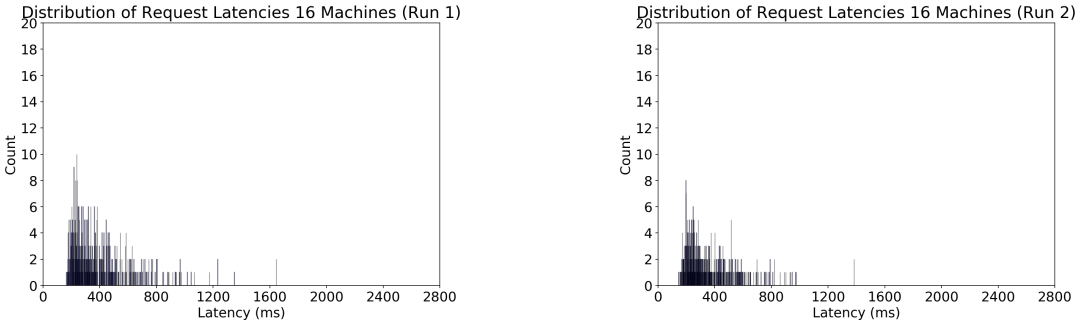
In spite of the promising result, it is cumbersome to determine the best training schedule for each model, given the large diversity of models intended to run on the implemented system. The presented approach provides a good direction for adaptation and future work could involve a study of the noise scale heuristic to dynamically adapt flexible synchronisation strategies.

4.6 Use case: optimising peer selection based on network monitoring

We have seen that training statistic monitoring is promising for flexible synchronisation and it can help convergence for partial gradient exchange. Next, we will present how adaptation can be leveraged when models are exchanged between peers. We first describe experimental results showing instability in commodity networks in the Cloud, then propose an adaptive approach that takes into account selective peer communication based on network link quality.

4.6.1 Key idea

Cloud environments often provide unpredictable network conditions when Quality of Service is not defined. One such example is the 16-machine cluster used to run scalability experiments in Chapter 3. This uses Docker network on top of modest interconnects. This unprivileged environment clearly



(a) Validation accuracy over epochs

(b) Validation accuracy over physical time

Figure 4.5: Distribution of request latencies registered by one peer across different runs

affected scalability for Synchronous Peer Model Averaging, where the scaling difficulty is visible when transitioning from 8 machines to 16 machines (Figure 3.15).

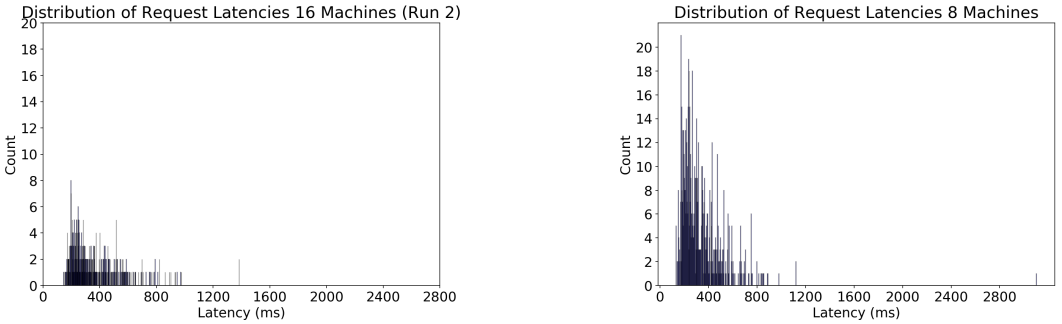
The scaling bottleneck can be addressed by employing network monitoring techniques to determine the optimal network conditions that allow model exchange between peers. This is based on the intuition that random or even round robin peer selection do not *smartly* determine which peer to send the request to. Instead, they blindly use links which may have poor quality. This problem can be tackled by enabling peers to monitor previous interactions and to take decisions for peer selection based on training statistics that indicate the quality of communication.

4.6.2 Measuring network variability in commodity cloud

The infrastructure facilitates efficient gathering of such statistics, which can be used for smarter peer selection strategies. For example, the new metrics can be aggregated to create a multi-attribute discriminator metric in communication. This would enable better use of network links and uniform peer-to-peer communication, where cliques are not likely to form. The example provides a key intuition of how these metrics can be used, but it is important to first understand specific insights gained from monitored metrics in a real cluster.

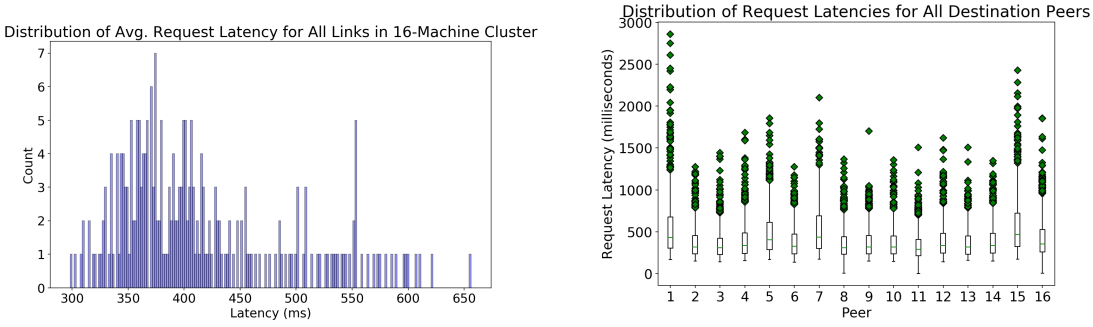
The metrics are gathered on a 16-machine cluster provided by Huawei Cloud Platform [41], each dedicated with one P100 GPU [70]. The cluster is part of a public resource pool, so there are no guarantees on the Quality of Service (QoS) of the network which is likely over-provisioned. This creates a good setting to study unpredictability in distributed training using flexible synchornisation. The DNN used is ResNet-50 (large, computationally intensive), which simulates training on synthetic data for 500 mini-batches, each of size 64. We isolate only Synchronous Peer Model Averaging (SPMA) for testing because: (i) the expected behaviour across peers is identical due to synchronous training and an assumed uniform distribution of peer choice, i.e., on expectation a specific peer is not flooded with requests and (ii) it exhibits scalability problems, for example when training on the same 16-machine cluster, the total system throughput increase when transitioning from 8 peers to 16 peers is only 42% and it is worth exploring if this problem is a network link problem.

The first experiment investigates the difference in request latency distributions for a single worker across two different runs. This experiment is relevant for modern Cloud envrionments where users are responsible for creating jobs, but the decision for job placement on machines is exclusively delegated to the cluster management software (e.g., Kubernetes [113]). This means that across two different runs, a worker can be placed on a machine whose switch is not over-provisioned, while another worker is placed on a machine affected by reduced bandwidth. The experiment shows that request latency (time to complete the request command) distributions differ slightly: on run 1, mean 389.2ms and standard deviation 207.8ms; on run 2, mean 356.5ms and standard deviation 176.5ms (Figures 4.6.2 and 4.5b). Computing the standard deviation on the whole population of request latencies confers robustness to outliers. However, the outliers' effect is likely responsible to introduce performance penalties in the system. The takeaway is that non-uniform distributions characterised by slightly different statistics are expected on repeated runs of a training job in a multi-machine commodity cluster.



(a) (b)

Figure 4.6: Distribution of request latencies when number of machines decreased



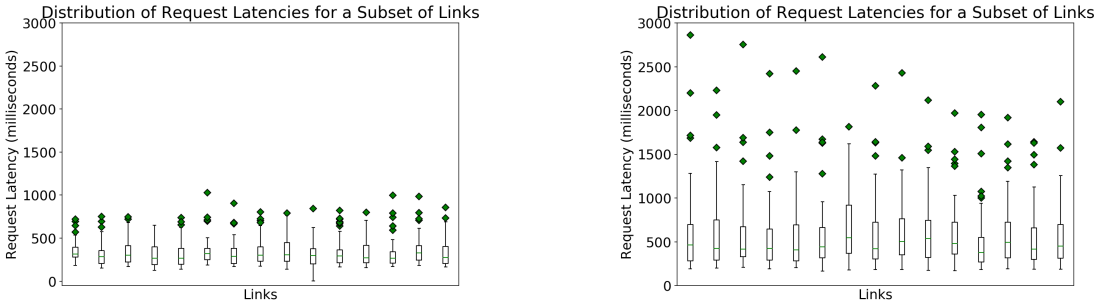
(a) (b)

Figure 4.7: (a) Average variance registered by all links, (b) All destination peers in the system

When decreasing the number of machines, the latencies center around a smaller value (mean 332.8ms), but the spread is similar (standard deviation 180.6ms). Also, the frequency of communication increases because system throughput increases, which means that on average the system benefits from better latencies. The closeness of standard deviations in the 8 machine case and the 16 machine case shows that request latency variance is preserved irrespective of the scale of computation.

A cluster of 16 machines is interconnected by 256 links. A link is a conceptual representation of the path between two workers, abstracting away networking elements in-between (e.g., switches, overlay handling and other layers of indirection). Monitoring all *pair-wise links* in the system provides a view of the overall network quality. Figure 4.7a shows the distribution (as a histogram) of the average network latencies registered for every link in the system. The distribution shows a bi-modal tendency, where the second, less populated mode consists of high latencies. This means that the average quality of some links is lower than the average of the distribution. This result shows that the variant behaviour is present in the global view of the links and provides a solid motivation to combat non-uniform link quality. When isolating subsets of links, as in Figure 4.7, we observe that some links have low latency variance, while others have a much higher variance. Links of both types are isolated in Figures 4.8a and 4.8b. The box plots show the quartiles of the distribution. The green diamonds represent outliers, preserved in the figure for illustrating their large number. Outliers are samples which are outside the inter-quartile range by a margin of $1.5 \cdot InterQuartileRange$ [127].

A *peer-wise view* of request latencies illustrates that *some peers have better QoS* for communication: (i) less variance in requests which addressed to themselves and (ii) less spread in outliers. This motivates the need to employ selective strategies and to communicate with peers which provide better service in terms of reachability and response. Selection should be fair (do not form cliques) and should consider network latencies and communication frequency to best discriminate where the model should be requested from.



(a) (b)

Figure 4.8: Isolated links: smallest request latency variance and largest request latency variance

4.6.3 Optimising the selection of peers online

Empirical measurements of request latency in two different cluster setups show that there is high latency variance. Measured locally on one server machine dedicated with 4 GeForce GTX TITAN X (Pascal) GPUs and 20 Intel(R) Xeon(R) CPU E5-2640 v4 @ 2.40GHz (Cluster Setup 3), the latency range is between 500 microseconds and 20 milliseconds. Measured in a 16-machine bare-metal cluster (provisioned by Huawei Cloud [41]) with Docker network bridges , the request latency varies between 100 microseconds and one second. High variance indicates a heterogeneous environment where communication instability should be a decision factor when issuing requests. Additionally workers should avoid imbalanced communication, where cliques are formed. The factor as stake is ineffective synchronisation through model averaging. The aim is to build a robust system which is able to perform well in varied environments, therefore it is important to integrate these observations when designing the algorithm for peer selection.

Two metrics are necessary to implement a policy for effective synchronisation which can be enabled locally by each worker in a decentralised manner, without external interaction: request latency, which represents the time measured from the moment the request is sent until the moment when a reply is delivered and peer interaction frequency, which is a count of previous interactions with a peer. The algorithm is implemented in Go, within the networking stack. This provides sufficient cluster abstraction to support adaptive peer selection. In the first training iteration, each worker engages in unidirectional model exchange with one random peer. Subsequently, each interaction is registered in a priority queue (implementation based on the Go heap standard package [108]) as an Aggregated Metric, presented below:

```
type AggregatedMetric struct {
     Rank int
     Frequency int64
     Latency float64
     Index int
}
```

Listing 4.3: Aggregated metric used as peer discriminator in communication

This data structure holds the Frequency of synchronisation with peer identified by Rank and the last registered request latency. The Index refers to the index in the priority queue, implemented as an array. In order to decide which is the next peer to communicate with, this aggregated metric determines the Rank's priority. The first considered metric is frequency of communication. If this is small the Rank has high priority, as workers desire to receive models from peers they have seldom communicated with and incorporate their learnt information through model averaging. The second discriminating metric (in case of latency equality) is the communication frequency. Previous requests which register smaller latencies have priority.

Algorithm 5 is a proposal for adaptive peer model averaging. It presents the logical steps required for random peer selection, while abstracting away most of the model request logic. This is a global algorithm, where T iterations are executed in parallel and the data structures support concurrent access. The algorithm relies on a priority queue (min heap) which is an efficient data structure allowing insertion and removal of elements in logarithmic time. The comparator used for

Algorithm 5: Adaptive Peer Model Averaging (Logical Worker View)

input : $aggregatedMetricsPQ$ concurrent priority queue (min heap) of metrics;
$aggregatedMetricsComparator$ discriminator by latency and frequency;
$frequencyTable$ concurrent hash table representing a frequency table;
T total training iterations;
output: $V^\star = \{v_0^\star \dots v_{N-1}^\star\}$ updated model after T iterations

```
/* Lines 1-2 executed once, at the beginning of training */
```
1 initPriorityQueue(aggregatedMetricsPQ, aggregatedMetricsComparator);

```
/* Initial request sent to random peer, monitored and inserted in heap */
```
2 **for** $i \in 1 \dots T$ ***in parallel*** **do**
```
    /* Pop operation retrieves and removes top min heap element */
```
3 destRank $\leftarrow pop(aggregatedMetricsPQ)$
4 latency $\leftarrow launchTimedRequest(destRank)$
5 updateFrequencyTable(destRank);
6 updatedFrequency $\leftarrow get(frequencyTable, destRank)$
7 metric $\leftarrow newAggregatedMetric(destRank, updatedFrequency, latency)$
8 push(aggregatedMetricsPQ, metric);
9 **end**
10 **return** $V^\star$;

the priority queue (line 1) relies on the previously described discriminating metrics: latency and frequency of communication. When the priority queue is empty, the first request is directed to a random peer. Subsequently, a peer which previously had a small latency value (or in case of equality a lower communication frequency), is removed from the priority queue (line 3). Communication with this peer is likely to show good network conditions and the base logic for timed model request is executed (line 4). The frequency map is updated (lines 5 - 6) and a new metric is registered in the heap (lines 7 - 8).

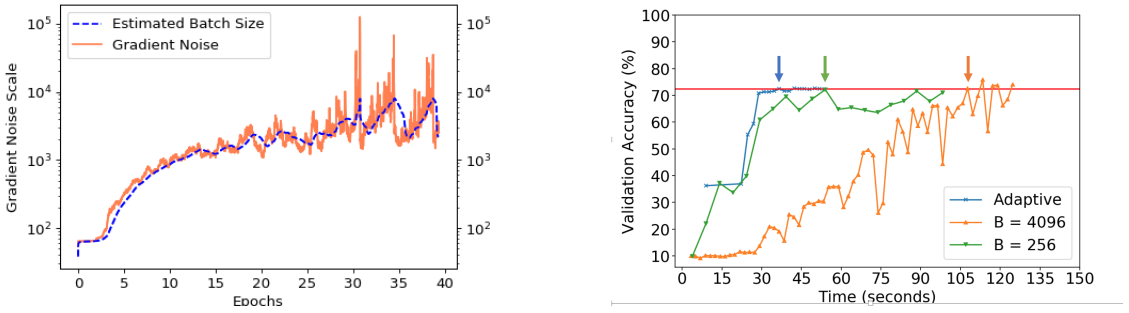
4.6.4 Ongoing work

This algorithm is presented as a proposal and there is ongoing work aimed at observing the benefits of adaptive peer model averaging. Features that could make the algorithm more robust involve: using a sliding window for considering only recent history, creating a more complex peer ranking algorithm ensuring that cliques are not formed in the system and that worker collaboration is uniform, edge case likely to occur if a subset of links are of significantly better quality. Conquering the adaptation mechanisms in peer-to-peer model synchronisation may become critical to scalability in constrained Cloud environments and we believe that dedicating effort towards solving this problem can push the boundaries of scalability even more.

4.7 Adaptation beyond synchronisation

The monitoring and adaptation techniques can benefit a broader range of hyperparameters, such as the batch size, the weight decay or the learning rate. In the following, we start from the learning rate adaptation intuition, then focus on batch size as the main metric that can benefit from adaptive training.

According to the authors of [114], the well-established learning rate schedules used for training Deep Neural Networks have equivalent batch increase schedules which touch the learning potential of several reference models such as ResNet-50. This means that all learning rate changes specified in the schedule are replaced by batch size changes. The authors argue that a batch size change during training which is proportional to the equivalent pre-set learning rate schedule has a similar effect on escaping learning plateaus (i.e., when learning rate is *decayed* by a constant, batch size should be *increased* by the same constant). When training with large batch sizes, there are less updates made to the model [114, 27], which is likely to affect the progress made per iteration.



(a) (b)

Figure 4.9: (a) Gradient noise scale and batch size estimation for ResNet-32, (b) Benefit of adaptive batch [6]

Training with such batch sizes often requires a large pool of GPUs [27] to ensure that aggregated updates increase the amount of gradient information incorporated in the model. As such, large batch training is not widely used in practice because it affects convergence in the early stages of training. Starting with small batch sizes which are gradually increased is a solution that best exploits the trade-off between accuracy and hardware utilisation and this can be effectively guided by adaptive batch training.

Experimental setup Experiments are conducted to verify the effectiveness of training with adaptive batch sizes based on the gradient noise and critical batch estimations implemented in Kungfu. We use one server machine dedicated with 4 GeForce GTX TITAN X (Pascal) GPUs and 20 Intel(R) Xeon(R) CPU E5-2640 v4 @ 2.40GHz (Cluster Setup 3). All experiments are run with four workers, each training on one GPU. We use ResNet-32 model as reference, trained on the Cifar-10 dataset [62].

Case study: benefit of adaptive algorithms To test the claims on the effectiveness of dynamic adaptation following successful noise and critical batch estimations, further experiments are conducted on ResNet-32. The starting point is finding smooth approximations for the raw gradient noise and for the estimated batch size, derived from the raw gradient noise by EMA. This is achieved by tuning the new EMA decay factors introduced in the system. Figure 4.9a shows the scale of estimations across epochs. The estimated (critical) batch size is used for successive 1-epoch training runs with a newly estimated batch size. After each epoch a model checkpoint is created and restored for a run on the validation set. There are no warm-up batches. Experiments show that adaptive batch indeed helps the model converge faster (Figure 4.9b): improvement of 47.5% when compared to the run with a static batch size of 256 and improvement of 99.4% when compared to a run with static batch size of 4096. The global batch size range during adaptation is $B \in \{256 \dots 4096\}$, but the transition between epochs is made by gradual batch size increase. The result proves benefits from two perspectives when training with adaptive batches: (1) model accuracy does not suffer, because small batches are used in the beginning of training and (2) hardware utilisation increases during training at the same time with batch size increases, thus ensuring that resources are not wasted.

4.8 Summary

By adding monitoring metrics and by enabling flexible synchronisation strategies, we provide a robust framework by which people can benefit from more scalable training systems. The system should enable users to run distributed training more efficiently, rather than create a burden for choosing synchronisation hyperparameters. The responsibility of specifying synchronisation hyperparameters shifts to the system, which smartly decides how to do synchronisation based on training quality and cluster conditions.

We present the principles for designing a system that supports the user in choosing the right synchronisation by providing monitoring and adaptation layers. The monitoring layer is designed

to provide a lightweight framework for enabling a complete view of communication quality and of training quality. Communication quality in the peer-to-peer paradigm adopted by Kungfu is determined mainly by request latency and frequency of communication. We use gradient noise [118] to determine training quality in the case of collective commmunication. It represents a noise-to-signal ratio quantifying the magnitude of gradient updates, with similar properties across families of DNN models. The adaptive layer proposes two use-cases that: (i) facilitate scalable Deep Learning training in commodity Cloud envrionments by actively using online network metrics to rank peers based on their link conditions and (ii) enable scalable synchronisation strategies with minimal convergence degradation due to poor choice of hyperparameters. Finally, we go beyond synchronisation and showcase an example of adaptive algorithm which leverages gradient noise to dynamically change batch sizes.

Chapter 5

Conclusion and Future Work

Working on Kungfu was an intense engineering experience which challenged us to combine techniques and knowledge from two distinct domains: Deep Learning and Systems, in order to design solutions for flexible synchronisation. Building distributed training systems requires to deal with aspects from both fields. Firstly, we become familiar with training pipelines, new Cloud environments which require in-depth understanding for explainability of performance results and new models and datasets, for which we dedicated time to understand from an algorithmic perspective as well (architecture, hyperparamter settings, optimization methods, schedules). Secondly, the system component dominated. The behaviour of Kungfu during training was mainly affected by our design and implementation decisions. These challenged us in: concurrency, software design in a project built using three programming languages (Python, C++ and Go), performance considerations on the critical training path. These challenges were driving the project forward and we enjoyed tackling such deep problems.

For future Deep Learning systems, flexible synchronisation can resolve current challenges by conquering scaling and large batch training. The new approaches we propose for flexible synchronisation show capability of improving training speeds on state-of-the art Deep Neural Networks by up to 40% and of increasing system-wide training throughput 22.5x. The achievement motivates practical applicability of flexible synchronisation in varied cluster environments, from high-performance GPU-accelerated hosts interconnected by ultra-low latency InfiniBand links, to clusters of tens of machines for which scarce network provisioning has become a true source of communication bottleneck for distributed training. Wide availability of such strategies, that can be non-invasively integrated into training programs would allow users to train their models faster and to better leverage *any type* of Cloud resource, providing ease of deployability through a *homogeneous* worker architecture where there is no single entity that controls synchronisation. Users are even supported by the system to use the correct synchronisation strategy, in order to release the burden of manual tuning. This is shown to close the convergence gap of 9% created by fixed hyperparameters in ResNet-50, favoring an accuracy gain of 5%.

Throughout the project, our system achieved notable experimental results on state-of-the art Deep Neural Networks, but it is key to acknowledge that these are complemented by limitations. For example, we have identified breaking points in Partial Gradient Exchange, such as tight model dependence caused by the design choice to use budget-based partitioning. Breaking points are also identified in Synchronous Peer Model Averaging, where blocking model requests happen on the critical training path, so the system does not scale well. Our aim was to explore techniques that solve problems in the base system using smart heuristics based on monitored network and training statistics. Although devised, heuristics based on monitored metrics are currently leveraged in an empirical manner. Users need extra effort and successive training runs to be able to design an effective training policy: one run for monitoring interleaved with a policy design phase, followed eventually by the full training run with the policy in effect.

We look forward to extending the Kungfu project and with continued development effort, we aim to improve the system and to create a robust distributed training framework widely accessible to the public. For this, we need to address the main limitations concerning adaptive training and to fully address dynamic adaptation in distributed training.

So far, we provide interesting findings on monitored statistics to motivate more advanced flexible synchronisation strategies:

- **Peer Model Averaging** The proposed algorithm for selective peer interactions based on network monitoring metrics requires concrete integration in the Kungfu system. The current proposal creates a simple peer ranking algorithm, but there are many edge cases likely to occur in unpredictable Cloud environments, such as repetitive communication patterns or worker isolation. Isolating the benefits of such a system requires careful experimental set-up in a fully-controlled environment where adverse network conditions can be artificially generated.
- **Partial Gradient Exchange** The next step is to integrate the monitoring component with the partial exchange component and to build new algorithms for decision making in synchronisation. This avenue distinctively deserves dedication of future research resources and exploration for pushing the boundaries of smart synchronisation.

Bibliography

[1] Computer Science Dept University of Maryland. Neural loss functions with and without skip connections. URL https://www.cs.umd.edu/~tomg/projects/landscapes/. [Accessed 8 June 2019].

[2] Anna Choromanska, Mikael Henaff, Michael Mathieu, Gerard Ben Arous, and Yann LeCun. The loss surfaces of multilayer networks, Nov 30, 2014. URL https://arxiv.org/abs/1412.0233.

[3] TensorFlow Developers. TensorFlow Documentation - Piecewise Constant Decay, . URL https://www.tensorflow.org/guide/extend/architecture. [Accessed 8 June 2019].

[4] Luo Mai, Lukas Rupprecht, Abdul Alim, Paolo Costa, Matteo Migliavacca, Peter Pietzuch, and Alexander L. Wolf. Netagg: Using middleboxes for application-specific on-path aggregation in data centres. In *Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies*, CoNEXT '14, pages 249–262, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-3279-8. doi: 10.1145/2674005.2674996. URL http://doi.acm.org/10.1145/2674005.2674996.

[5] Alexander Sergeev and Mike Del Balso. Horovod: fast and easy distributed deep learning in tensorflow, Feb 15, 2018. URL https://arxiv.org/abs/1802.05799.

[6] Luo Mai, Alexandros Koliousis, Guo Li, Andrei-Octavian Brabete, and Peter Pietzuch. Taming hyper-parameters in deep learning systems. *Submitted for ACM Operating Systems Review*, May 2019.

[7] Schuller Bjoern Zafeiriou Stefanos and Bronstein Michael. Deep learning course, 2019. Imperial College London.

[8] Hubel D. and Wiesel T. Receptive fields, binocular interaction, and functional architecture in the cat's visual cortex. *Journal of Physiology*, 160:106–154, 1962.

[9] Alex Krizhevsky, Sutskever, Ilya, Hinton, and Geoffrey E. 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 25*, pages 1097–1105. Curran Associates, Inc., 2012. URL http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf.

[10] Russakovsky Olga, Deng Jia, Su Hao, Krause Jonathan, Satheesh Sanjeev, Sean Ma, Huang Zhiheng, Karpathy Andrej, Khosla Aditya, Bernstein Michael, Berg Alexander C., and Li Fei-Fei. Imagenet large scale visual recognition challenge. *Int. J. Comput. Vision*, 115(3): 211–252, December 2015. ISSN 0920-5691. doi: 10.1007/s11263-015-0816-y. URL http://dx.doi.org/10.1007/s11263-015-0816-y.

[11] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Jun 2016. doi: 10.1109/cvpr.2016.90. URL http://dx.doi.org/10.1109/CVPR.2016.90. [Accessed 9 June 2019].

[12] Goodfellow Ian, Pouget-Abadie Jean, Mirza Mehdi, Xu Bing, Warde-Farley David, Ozair Sherjil, Courville Aaron, and Bengio Yoshua. Generative adversarial nets. In Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger, editors, *Advances in Neural*

Information Processing Systems 27, pages 2672–2680. Curran Associates, Inc., 2014. URL http://papers.nips.cc/paper/5423-generative-adversarial-nets.pdf.

[13] Isola Phillip, Zhu Jun-Yan, Zhou Tinghui, and Efros Alexei A. Image-to-image translation with conditional adversarial networks. *arxiv*, 2016. URL https://arxiv.org/abs/1611.07004. [Accessed 9 June 2019].

[14] Karras Tero, Aila Timo, Laine Samuli, and Lehtinen Jaakko. Progressive growing of gans for improved quality, stability, and variation, 2017.

[15] Silver David, Huang Aja, Chris J. Maddison, Guez Arthur, and Sifre Laurent et al. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587):484–489, January 2016. ISSN 0028-0836. doi: 10.1038/nature16961.

[16] Li Mu, Andersen David G., Park Jun Woo, Smola Alexander J., Ahmed Amr, Josifovski Vanja, Long James, Shekita Eugene J., and Su Bor-Yiing. Scaling distributed machine learning with the parameter server. In *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014.*, pages 583–598, 2014. URL https://www.usenix.org/conference/osdi14/technical-sessions/presentation/li_mu.

[17] Kim Hazelwood, Sarah Bird, David Brooks, Soumith Chintala, Utku Diril, Dmytro Dzhulgakov, Mohamed Fawzy, Bill Jia, Yangqing Jia, Aditya Kalro, James Law, Kevin Lee, Jason Lu, Pieter Noordhuis, Misha Smelyanskiy, Liang Xiong, and Xiaodong Wang. Applied machine learning at facebook: A datacenter infrastructure perspective. pages 620–629. IEEE, 24 Feb 2018. doi: 10.1109/HPCA.2018.00059. URL https://ieeexplore.ieee.org/document/8327042.

[18] Rosenblatt F. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, pages 65–386, 1958.

[19] Kurt Hornik. Approximation capabilities of multilayer feedforward networks. *Neural Netw.*, 4(2):251–257, March 1991. ISSN 0893-6080. doi: 10.1016/0893-6080(91)90009-T. URL http://dx.doi.org/10.1016/0893-6080(91)90009-T.

[20] Herbert Robbins and Sutton Monro. A stochastic approximation method. *Ann. Math. Statist.*, 22(3):400–407, 09 1951. doi: 10.1214/aoms/1177729586. URL https://doi.org/10.1214/aoms/1177729586.

[21] Rumelhart David E., Hinton Geoffrey E., and Williams Ronald J. Learning representations by back-propagating errors. *Nature*, 323:533–, October 1986. URL http://dx.doi.org/10.1038/323533a0. [Accessed 10 June 2019].

[22] Deng J., Dong W., Socher R., Li L.-J., Li K., and L. Fei-Fei. ImageNet: A Large-Scale Hierarchical Image Database. In *CVPR09*, 2009.

[23] M. Meiss, F. Menczer, S. Fortunato, A. Flammini, and A. Vespignani. Ranking web sites with real user traffic. In *Proc. First ACM International Conference on Web Search and Data Mining (WSDM)*, pages 65–75, 2008. URL http://informatics.indiana.edu/fil/Papers/click.pdf. [Accessed 15 June 2019].

[24] Mu Li, Tong Zhang, Yuqiang Chen, and Alexander J. Smola. Efficient mini-batch training for stochastic optimization. In *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '14, pages 661–670, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2956-9. doi: 10.1145/2623330.2623612. URL http://doi.acm.org/10.1145/2623330.2623612. [Accessed 12 June 2019].

[25] Jeffrey Dean, Greg Corrado, Rajat Monga, Chen Kai, and Matthieu et al. Devin. Large scale distributed deep networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1223–1231. Curran Associates, Inc., 2012. URL http://papers.nips.cc/paper/4687-large-scale-distributed-deep-networks.pdf.

[26] TensorFlow. TensorFlow Federated: Machine Learning on Decentralized Data. URL https://www.tensorflow.org/federated. [Accessed 9 June 2019].

[27] Priya Goyal, Piotr Dollar, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. Accurate, large minibatch sgd: Training imagenet in 1 hour, Jun 8, 2017. URL https://arxiv.org/abs/1706.02677. [Accessed 15 June 2019].

[28] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, and Jeffrey Dean et al. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. URL https://www.tensorflow.org/. [Accessed 10 June 2019].

[29] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in PyTorch. In *NIPS Autodiff Workshop*, 2017. URL https://pytorch.org/. [Accessed 15 June 2019].

[30] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. *arXiv preprint arXiv:1408.5093*, 2014.

[31] Frank Seide and Amit Agarwal. Cntk: Microsoft's open-source deep-learning toolkit. In *Proceedings of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, pages 2135–2135, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4232-2. doi: 10.1145/2939672.2945397. URL http://doi.acm.org/10.1145/2939672.2945397.

[32] François Chollet et al. Keras. https://keras.io, 2015. [Accessed 12 June 2019].

[33] Pijika Watcharapichat, Victoria Lopez Morales, Raul Castro Fernandez, and Peter Pietzuch. Ako: Decentralised deep learning with partial gradient exchange. Santa Clara, CA, USA, Oct 10 2016. ACM. URL https://lsds.doc.ic.ac.uk/projects/ako. [Accessed 10 June 2019].

[34] Luo Mai, Chuntao Hong, and Paolo Costa. Optimizing network performance in distributed machine learning. In *7th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 15)*, Santa Clara, CA, 2015. USENIX Association. URL https://www.usenix.org/conference/hotcloud15/workshop-program/presentation/mai. [Accessed 9 June 2019].

[35] Wolfram Neural Network Repository. ResNet-152 Trained on ImageNet Competition Data. URL https://resources.wolframcloud.com/NeuralNetRepository/resources/ResNet-152-Trained-on-ImageNet-Competition-Data. [Accessed 8 June 2019].

[36] NVIDIA. NVIDIA Tesla V100, . URL https://www.nvidia.com/en-gb/data-center/tesla-v100. [Accessed 8 June 2019].

[37] NVIDIA. NVIDIA Tesla Deep Learning Product Performance, . URL https://developer.nvidia.com/deep-learning-performance-training-inference. [Accessed 12 June 2019].

[38] Martin A. Zinkevich, Markus Weimer, Alex Smola, and Lihong Li. Parallelized stochastic gradient descent. In *NIPS'10*, pages 2595–2603, Vancouver, British Columbia, Canada, 1 2010. Curran Associates Inc.

[39] Xinghao Pan, Jianmin Chen, Rajat Monga, Samy Bengio, and Rafal Jozefowicz. Revisiting distributed synchronous sgd, 2017. URL https://arxiv.org/abs/1604.00981. [Accessed 15 June 2019].

[40] Hussain Aljahdali, Abdulaziz Albatli, Peter Garraghan, Paul Townend, Lydia Lau, and Jie Xu. Multi-tenancy in cloud computing. 04 2014. doi: 10.1109/SOSE.2014.50.

[41] Huawei Technologies. Huawei cloud. URL https://www.huaweicloud.com. [Accessed 4 June 2019].

[42] Google Cloud Platform. Preemptible VM Instances, . URL https://cloud.google.com/compute/docs/instances/preemptible. [Accessed 9 June 2019].

[43] Amazon Web Services. Amzon EC2 Spot Instances, . URL https://aws.amazon.com/ec2/spot. [Accessed 12 June 2019].

[44] Google Cloud Platform. Reserving Compute Engine zonal resources, . URL https://cloud.google.com/compute/docs/instances/reserving-zonal-resources. [Accessed 12 June 2019].

[45] Amazon Web Services. Amzon EC2 Reserved Instances, . URL https://aws.amazon.com/ec2/pricing/reserved-instances. [Accessed 12 June 2019].

[46] Gregory F Pfister. An introduction to the infiniband architecture. URL http://gridbus.csse.unimelb.edu.au/~raj/superstorage/chap42.pdf. [Accessed 12 June 2019].

[47] NVIDIA. NVLink Fabric, . URL https://www.nvidia.com/en-gb/data-center/nvlink. [Accessed 12 June 2019].

[48] Ben Cotton. Adapting infiniband for high performance cloud computing. January 2017. URL https://www.nextplatform.com/2017/01/17/adapting-infiniband-high-performance-cloud-computing. [Accessed 12 June 2019].

[49] G.M. Shipman, T.S. Woodall, Richard Graham, Arthur Maccabe, and Patrick Bridges. Infiniband scalability in open mpi. volume 2006, pages 10 pp.–, 05 2006. ISBN 1-4244-0054-6. doi: 10.1109/IPDPS.2006.1639335.

[50] Baidu Research. Tensorflow-allreduce source code, 2015. URL https://github.com/baidu-research/baidu-allreduce. [Accessed 4 June 2019].

[51] Open Souce Contributors. Open-mpi, 2019. URL www.open-mpi.org. [Accessed 4 June 2019].

[52] Mu Li. Scaling distributed machine learning with the parameter server. In *OSDI'14*, BigDataScience '14, page 1. ACM, Aug 4, 2014. doi: 10.1145/2640087.2644155. URL http://dl.acm.org/citation.cfm?id=2644155.

[53] X. Zhang, N. Gu, R. Yasrab, and H. Ye. Gt-sgd: A novel gradient synchronization algorithm in training distributed recurrent neural network language models. In *2017 International Conference on Networking and Network Applications (NaNA)*, pages 274–278, Oct 2017. doi: 10.1109/NaNA.2017.22.

[54] Yusuke Tsuzuku, Hiroto Imachi, and Takuya Akiba. Variance-based gradient compression for efficient distributed deep learning, 2018. URL https://openreview.net/forum?id=rkEfPeZRb.

[55] Xianyan Jia, Shutao Song, Wei He, Yangzihao Wang, Haidong Rong, Feihu Zhou, Liqiang Xie, Zhenyu Guo, Yuanzhou Yang, Liwei Yu, Tiegang Chen, Guangxiao Hu, Shaohuai Shi, and Xiaowen Chu. Highly scalable deep learning training system with mixed-precision: Training imagenet in four minutes, 2018.

[56] Sixin Zhang, Anna Choromanska, and Yann LeCun. Deep learning with elastic averaging sgd, 2014. URL https://arxiv.org/abs/1412.6651. [Accessed 10 June 2019].

[57] Koliousis A., Watcharapichat P., Weidlich M., Mai L., Costa P, and Pietzuch P. Peter. Crossbow: Scaling deep learning with small batch sizes on multi-gpu servers. *arXiv:1901.02244*, Jan 8 2019.

[58] Xiangru Lian, Wei Zhang, Ce Zhang, and Ji Liu. Asynchronous decentralized parallel stochastic gradient descent. In *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018*, pages 3049–3058, 2018. URL http://proceedings.mlr.press/v80/lian18a.html.

[59] Open Source. TensorFlow benchmarks. URL https://github.com/tensorflow/benchmarks. [Accessed 9 June 2019].

[60] TensorFlow Documentation. URL https://www.tensorflow.org/api_docs/python/tf/train/Optimizer. [Accessed 5 June 2019].

[61] Bassam Bamieh. Discovering the fourier transform: A tutorial on circulant matrices, circular convolution, and the dft. *arXiv preprint arXiv:1805.05533*, 2018.

[62] Krizhevsky Alex, Nair Vinod, and Hinton Geoffrey. Cifar-10 (canadian institute for advanced research). URL http://www.cs.toronto.edu/~kriz/cifar.html.

[63] Goodfellow Ian, Bengio Yoshua, and Courville Aaron. *Deep Learning*. MIT Press, http://www.deeplearningbook.org, 2016.

[64] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. Technical Report UCB/EECS-2010-24, EECS Department, University of California, Berkeley, Mar 2010. URL http://www2.eecs.berkeley.edu/Pubs/TechRpts/2010/EECS-2010-24.html.

[65] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015. URL http://arxiv.org/abs/1412.6980.

[66] Ilya Sutskever, James Martens, George Dahl, and Geoffrey Hinton. On the importance of initialization and momentum in deep learning. In *Proceedings of the 30th International Conference on International Conference on Machine Learning - Volume 28*, ICML'13, pages III–1139–III–1147. JMLR.org, 2013. URL http://dl.acm.org/citation.cfm?id=3042817.3043064.

[67] Cauchy Augustin-Louis. Methode generale pour la resolution des systemes d'equations simultanees. Series A, No. 25(Compte Rendu des S'eances de L'Acad'emie des Sciences XXV): 536–538, Oct 18 1847.

[68] Lomont Chris. Introduction to intel advanced vector extensions. URL https://software.intel.com/en-us/articles/introduction-to-intel-advanced-vector-extension. [Accessed 4 June 2019].

[69] TensorFlow. Xla (accelerated linear algebra). URL https://www.tensorflow.org/xla. [Accessed 4 June 2019].

[70] NVIDIA. NVIDIA Tesla P100, . URL https://www.nvidia.com/en-gb/data-center/tesla-p100. [Accessed 8 June 2019].

[71] Fatahalian K., Sugerman J., and Hanrahan P. Understanding the efficiency of gpu algorithms for matrix-matrix multiplication. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, HWWS '04, pages 133–137, New York, NY, USA, 2004. ACM. ISBN 3-905673-15-0. doi: 10.1145/1058129.1058148. URL http://doi.acm.org/10.1145/1058129.1058148.

[72] Vincent Vanhoucke, Andrew Senior, and Mark Z. Mao. Improving the speed of neural networks on cpus. In *Deep Learning and Unsupervised Feature Learning Workshop, NIPS 2011*, 2011. URL https://static.googleusercontent.com/media/research.google.com/en//pubs/archive/37631.pdf. [Accessed 15 June 2019].

[73] Meng Xiangrui, Bradley Joseph, Yavuz Burak, Sparks Evan, Shivaram Venkataraman, Liu Davies, Freeman Jeremy, Tsai DB, Amde Manish, Owen Sean, Xin Doris, Xin Reynold, Franklin Michael J., Zadeh Reza, Zaharia Matei, and Talwalkar Ameet. Mllib: Machine learning in apache spark. *J. Mach. Learn. Res.*, 17(1):1235–1241, January 2016. ISSN 1532-4435. URL http://dl.acm.org/citation.cfm?id=2946645.2946679.

[74] Martin Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, and Jeffrey Dean et al. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283, 2016. URL https://www.usenix.org/system/files/conference/osdi16/osdi16-abadi.pdf.

[75] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation - Volume 6*, OSDI'04, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1251254.1251264.

[76] Rami Al-Rfou, Guillaume Alain, Amjad Almahairi, Christof Angermueller, and Dzmitry Bahdanau et al. Theano: A Python framework for fast computation of mathematical expressions. *arXiv e-prints*, abs/1605.02688, May 2016. URL http://arxiv.org/abs/1605.02688.

[77] Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. Squad: 100,000+ questions for machine comprehension of text. *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*, 2016. doi: 10.18653/v1/d16-1264. URL http://dx.doi.org/10.18653/v1/D16-1264. [Accessed 15 June 2019].

[78] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[79] Mosic Ranko. Google bert–pre-training and fine tuning for nlp tasks, 5 November 2018. URL https://medium.com/@ranko.mosic/googles-bert-nlp-5b2bb1236d78. [Accessed 14 June 2019].

[80] Amazon Web Services. Serverless, . URL https://aws.amazon.com/serverless. [Accessed 13 June 2019].

[81] Amazon Web Services. Amazon Deep Learing AMIs, . URL https://aws.amazon.com/machine-learning/amis. [Accessed 13 June 2019].

[82] M. T. Chung, A. Le, N. Quang-Hung, D. Nguyen, and N. Thoai. Provision of docker and infiniband in high performance computing. In *2016 International Conference on Advanced Computing and Applications (ACOMP)*, pages 127–134, Nov 2016. doi: 10.1109/ACOMP.2016.027.

[83] Roy Arjun, Zeng Hongyi, Bagga Jasmeet, Porter George, and Snoeren Alex C. Inside the social network's (datacenter) network. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, pages 123–137, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3542-3. doi: 10.1145/2785956.2787472. URL http://doi.acm.org/10.1145/2785956.2787472.

[84] Lucian Popa, Sylvia Ratnasamy, Gianluca Iannaccone, Arvind Krishnamurthy, and Ion Stoica. A cost comparison of datacenter network architectures. In *Proceedings of the 6th International COnference*, Co-NEXT '10, pages 16:1–16:12, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0448-1. doi: 10.1145/1921168.1921189. URL http://doi.acm.org/10.1145/1921168.1921189.

[85] Docker. Use overlay networks. URL https://aws.amazon.com/ec2/pricing/reserved-instances/. [Accessed 12 June 2019].

[86] Abdul Kabbani, Mohammad Alizadeh, Masato Yasuda, Rong Pan, and Balaji Prabhakar. AF-QCN: approximate fairness with quantized congestion notification for multi-tenanted data centers. In *IEEE 18th Annual Symposium on High Performance Interconnects, HOTI 2010, Google Campus, Mountain View, California, USA, August 18-20, 2010*, pages 58–65, 2010. doi: 10.1109/HOTI.2010.26. URL https://doi.org/10.1109/HOTI.2010.26.

[87] VMWare. Virtual Networking Concepts, . URL https://www.vmware.com/content/dam/digitalmarketing/vmware/en/pdf/techpaper/virtual_networking_concepts.pdf. [Accessed 12 June 2019].

[88] VMWare. VMware Workstation 5.0, Virtual Switch, . URL https://www.vmware.com/support/ws5/doc/ws_net_component_vswitch.html. [Accessed 12 June 2019].

[89] Merkel Dirk. Docker: Lightweight linux containers for consistent development and deployment. *Linux J.*, 2014(239), March 2014. ISSN 1075-3583. URL http://dl.acm.org/citation.cfm?id=2600239.2600241.

[90] Mellanox Technologies. How to Create a Docker Container with RDMA Accelerated Applications Over 100Gb InfiniBand Network, . URL https://community.mellanox.com/s/article/how-to-create-a-docker-container-with-rdma-accelerated-applications-over-100gb-infiniband-ne [Accessed 13 June 2019].

[91] Aleksandar Dragojević, Dushyanth Narayanan, Orion Hodson, and Miguel Castro. Farm: Fast remote memory. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, NSDI'14, pages 401–414, Berkeley, CA, USA, 2014. USENIX Association. ISBN 978-1-931971-09-6. URL http://dl.acm.org/citation.cfm?id=2616448.2616486.

[92] Advanced clustering technologies, Inc. Infiniband types and speeds. URL https://www.advancedclustering.com/act_kb/infiniband-types-speeds. [Accessed 13 June 2019].

[93] Dhabaleswar K. (DK) Panda and Hari Subramoni. URL InfiniBand,Omni-Path,andHigh-SpeedEthernet:AdvancedFeatures,ChallengesinDesigningHECSystems,andUsage. [Accessed 12 June 2019].

[94] Mellanox Technologies. InfiniBand Switch Systems, . URL http://www.mellanox.com/page/switch_systems_overview. [Accessed 13 June 2019].

[95] NVIDIA Developers. Nvidia collective communications library (nccl). URL https://developer.nvidia.com/nccl. [Accessed 4 June 2019].

[96] Luehr Nathan. Fast multi-gpu collectives with nccl, April 7 2016. URL https://devblogs.nvidia.com/fast-multi-gpu-collectives-nccl/. [Accessed 4 June 2019].

[97] Inc. Microsoft. Microsoft azure cloud computing platform and services. URL https://azure.microsoft.com/en-gb/. [Accessed 5 June 2019].

[98] Sergeev Alexander. Horovod source code, Aug 6 2017. URL https://github.com/uber/horovod. [Accessed 8 June 2019].

[99] Alexander Smola and Shravan Narayanamurthy. An architecture for parallel topic models. *Proceedings of the VLDB Endowment*, 3(1-2):703–710, Sep 1, 2010. doi: 10.14778/1920841.1920931.

[100] Aurick Qiao, Abutalib Aghayev, Weiren Yu, Haoyang Chen, Qirong Ho, Garth A. Gibson, and Eric P. Xing. Litz: Elastic framework for high-performance distributed machine learning. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 631–644, Boston, MA, 2018. USENIX Association. ISBN 978-1-931971-44-7. URL https://www.usenix.org/conference/atc18/presentation/qiao. [Accessed 15 June 2019].

[101] Pitch Patarasuk and Xin Yuan. Bandwidth optimal all-reduce algorithms for clusters of workstations. *Journal of Parallel and Distributed Computing*, 69(2):117–124, 2009. doi: 10.1016/j.jpdc.2008.09.002. URL https://www.sciencedirect.com/science/article/pii/S0743731508001767. [Accessed 10 June 2019].

[102] Dominic Masters and Carlo Luschi. Revisiting small batch training for deep neural networks, 2018. URL https://arxiv.org/abs/1804.07612. [Accessed 10 June 2019].

[103] H. Brendan McMahan, Eider Moore, Daniel Ramage, Seth Hampson, and Blaise Agüera y Arcas. Communication-efficient learning of deep networks from decentralized data. In *AISTATS*, 2017.

[104] Michael Kamp, Mario Boley, Daniel Keren, Assaf Schuster, and Izchak Sharfman. Communication-efficient distributed online prediction by dynamic model synchronization. In *Proceedings of the 2014th European Conference on Machine Learning and Knowledge Discovery in Databases - Volume Part I*, ECMLPKDD'14, pages 623–639, Berlin, Heidelberg, 2014. Springer-Verlag. ISBN 978-3-662-44847-2. doi: 10.1007/978-3-662-44848-9_40. URL https://doi.org/10.1007/978-3-662-44848-9_40. [Accessed 10 June 2019].

[105] Message Passing Interface Forum. Mpi: A message-passing interface standar. In *ACM/IEEE Conference on Supercomputing*, Portland, OR, USA, 1993 September 21, 2012. Updated standard.

[106] Distributed TensorFlow. Tensorflow-allreduce source code, 2019. URL https://github.com/tensorflow/examples/blob/master/community/en/docs/deploy/distributed.md#putting_it_all_together_example_trainer_program. [Accessed 15 June 2019].

[107] TensorFlow Developers. Adding a New Op, TensorFlow Documentation, . URL https://www.tensorflow.org/guide/extend/op. [Accessed 8 June 2019].

[108] Alan A.A. Donovan and Brian W. Kernighan. *The Go Programming Language*. Addison-Wesley Professional, 1st edition, 2015. ISBN 0134190440, 9780134190440.

[109] Skiena Steven S. *The Algorithm Design Manual*. Springer Publishing Company, Incorporated, 2nd edition, 2008. ISBN 1848000693, 9781848000698. URL https://www8.cs.umu.se/kurser/TDBA77/VT06/algorithms/BOOK/BOOK2/NODE45.HTM. [Online] Available in book section "The Partitioning Problem" [Accessed 6 June 2019].

[110] Skiena Steven S. *The Algorithm Design Manual*. Springer Publishing Company, Incorporated, 2nd edition, 2008. ISBN 1848000693, 9781848000698. URL https://www8.cs.umu.se/kurser/TDBA77/VT06/algorithms/BOOK/BOOK5/NODE192.HTM. Available in book section "Bin Packing" [Accessed 6 June 2019].

[111] Pham, Thuan. Chapter 5, Bin packing problem implementation [Video], 2017. URL https://www.youtube.com/watch?v=uDdMuUWf6h4. [Accessed 8 June 2019].

[112] NVIDIA. NVIDIA DGX1, 2019. URL https://www.nvidia.com/en-gb/data-center/dgx-1. [Accessed 8 June 2019].

[113] Kubernetes. Kubernetes: Production-grade container orchestration. URL https://kubernetes.io/. [Accessed 4 June 2019].

[114] L. Smith Samuel, Pieter-Jan Kindermans, and Quoc V. Le. Don't decay the learning rate, increase the batch size. *CoRR*, abs/1711.00489, 2017. URL http://arxiv.org/abs/1711.00489.

[115] Ioffe Sergey and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *CoRR*, abs/1502.03167, 2015. URL http://arxiv.org/abs/1502.03167.

[116] TensorFlow Benchmarks Project. Replicated Variable Manager for TensorFlow, . URL https://github.com/tensorflow/benchmarks/blob/master/scripts/tf_cnn_benchmarks/variable_mgr.py. [Accessed 13 June 2019].

[117] TensorFlow Benchmarks Project. All-reduce specification for TensorFlow Replicated, . URL https://github.com/tensorflow/benchmarks/blob/master/scripts/tf_cnn_benchmarks/allreduce.py. [Accessed 13 June 2019].

[118] McCandlish Sam, Jared Kaplan, Dario Amodei, and OpenAI Dota Team. An empirical model of large-batch training, 2018. URL https://arxiv.org/pdf/1812.06162.pdf. [Accessed 5 May 2019].

[119] Hanemann Andreas, Liakopoulos Athanassios, Maurizio Molina, and Martin Swany. A study on network performance metrics and their composition. *Campus-Wide Information Systems*, 23, 08 2006. doi: 10.1108/10650740610704135. URL https://pdfs.semanticscholar.org/7039/4e5acae3b04521bba3bb6691cc65ed288044.pdf. [Accessed 15th of June 2019].

[120] G. Tangari, D. Tuncer, M. Charalambides, Y. Qi, and G. Pavlou. Self-adaptive decentralized monitoring in software-defined networks. *IEEE Transactions on Network and Service Management*, 15(4):1277–1291, Dec 2018. ISSN 1932-4537. doi: 10.1109/TNSM.2018.2874813.

[121] Prometheus Open Source Project. Prometheus. URL https://prometheus.io/. [Accessed 9 June 2019].

[122] Warren Paul. iftop: display bandwidth usage on an interface. URL http://www.ex-parrot.com/pdw/iftop/. [Accessed 9 June 2019].

[123] Wireshark open source contributors. Wireshark. URL https://www.wireshark.org/. [Accessed 9 June 2019].

[124] Linux Netfilter Contributors. The netfilter.org project. URL https://www.netfilter.org/. [Accessed 9 June 2019].

[125] Yusuke Tsuzuku, Hiroto Imachi, and Takuya Akiba. Variance-based gradient compression for efficient distributed deep learning. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Workshop Track Proceedings*, 2018. URL https://openreview.net/forum?id=Sy6hd7kvM.

[126] TensorFlow Developers. TensorFlow Architecture, . URL https://www.tensorflow.org/api_docs/python/tf/train/piecewise_constant_decay. [Accessed 8 June 2019].

[127] Matplotlib Library. Boxplot documentation. URL https://matplotlib.org/3.1.0/api/_as_gen/matplotlib.pyplot.boxplot.html. [Accessed 8 June 2019].

[128] Li Guo. All-reduce formalization algorithm, Jan 20 2019. Personal Communication.

[129] Zwick Uri. Lecture notes on "analysis of algorithms": Directed minimum spanning trees, April 22 2013. URL http://www.cs.tau.ac.il/~zwick/grad-algo-13/directed-mst.pdf. Lecture Notes,[Accessed January 2019].

Appendix A

Appendix

A.1 Horovod All-reduce Protocol Analysis

The following provides a detailed explanation of the implementation [98] of Horovod confirming the observations made in the scalability experiments. These insights are made based on analysis of the publicly available Horovod source code [98].

As seen in the Horovod Timeline, the negotiation phase is the most expensive. We will focus on `NEGOTIATE_ALLREDUCE`, which is done for all gradient tensors computed during a training iteration. The distributed setup employed by Horovod consists of workers, one of which (Rank 0) plays the role of master.

Horovod runs the same copy of the training process on multiple 4-GPU machines, using a distinctly shuffled dataset per training process. After each forward pass through the neural network, the weight updates computed using gradient descent are back-propagated. Before each weight tensor is updated locally, the global negotiation phase is triggered, resulting in computation of the average of all local gradients. The steps followed to compute [98] all-reduce with operator `MPI_SUM` on $tensor_i$ are the following:

- Worker sends `MPI_Request` for $tensor_i$ to master.
- After worker sends the `MPI_Request` for all trainable tensors, it sends a `DONE` message so that master should stop expecting any more messages from that worker.
- **Master** updates its `message_table`, which is a hashmap from tensor name to all timestamped requests received for that tensor. This is done by the function `IncrementTensorCount`.
  - When the first request for $tensor_i$ has been received the Horovod Timeline registers the beginning of the negotiation phase for request type ALLREDUCE (start of NEGOTIATE_ALLREDUCE)
  - When the size of the requests vector for $tensor_i$ is equal to `MPI_SIZE`, i.e. the number of workers running copies of the program, it means that the tensor is ready to be reduced. This is the end of the negotiation.

  The three-phase message receive operation is synchronous and initiated by the master: get message lengths from every rank (`MPI_Gather`), compute offsets and collect messages from every rank (`MPI_Gatherv`, which is the generalized gather version which accounts for an uneven number of messages received from workers).
- The master creates a vector of tensors ready to be reduced and for each tensor, it constructs an `MPI_Response` containing also potential error messages related to the worker's request.
- The master performs the optimization called Tensor Fusion, assembling a single stacked response for tensors with the same response type, devices, data type and with a joint size less than
  `TensorFusionThresholdBytes()`
- The names of the tensors reduced at the current step are broadcast to all workers

- The all-reduce operation is executed by the function `PerformOperation` on all tensors ready to be reduced

  - In the `MPI_ALLREDUCE` case, all tensors of the TensorTable are copied to the Fusion Buffer. The TensorTable stores TensorTableEntries uniquely identified by tensor name, each containing all necessary data to do the reduction: input tensor, pre-allocated output tensor, the GPU/CPU ID to do reduction on, root rank for broadcast operation, etc.

  - The allreduce operation is done in place (`MPI_IN_PLACE` flag), so the result of the reduction is present in the same buffer at the end of the computation.

  - The resulting tensor is copied to the pre-allocated output Tensor of the TensorTableEntry (See Timeline `MEMCPY_IN_FUSION_BUFFER`)

- In the end, the Master performs a stall check on all tensors that were reported as ready for reduction by some ranks (workers), but not by others. Such tensors may cause deadlock of the system.

- **Worker** sends to root rank the message length and the message in `MPI_Gather` and `MPI_Gatherv` (response operations to the Master-initated operations presented previously). The message is received via the Worker's local message queue.

- When the Master has decided that a tensor is ready to be reduced, it sends an `MPI_Response` to the Worker containing tensor meta-information, received via the `MPI_Bcast` primitive. The worker performs the operation indicated by the master, which can be one of: `MPI Allreduce`, `NCCL Allreduce` or hierarchical all-reduce (MPI & NCCL).

A.2 Collective All-reduce Communication Theory

Peer-to-peer systems that implement Stochastic Gradient Descent rely on collective communication to average the gradients for all workers. The key technique is the collective all-reduce communication, whose theoretical framework is presented next.

The all-reduce algorithm is frequently used in parallel computing. Each process combines values from all other processes and broadcasts the result. To formalize [128], let $P_0, P_1 \dots P_{K-1}$ be the set of processes and assume each process holds a row vector v of N values $a_i^0 \dots a_i^{N-1}$ for all $i \in \{0 \dots K-1\}$. [101] We want to compute the reduce function

$$reduce_{\oplus} : A^K \to A$$

across processes, for all vector positions $i \in 1 \dots N$. The operation $\oplus$ is commutative and associative.

Without loss of generality, computing the final result for vector element at index $j \in \{0 \dots N-1\}$, yields $a^j_{0\dots K-1} = a^0_0 \oplus a^0_1 \oplus \dots \oplus a^0_{K-1}$. At the end of the computation, the result is gathered a single node called the *root*, which will then broadcast it to its peers.

The reduce operation can be represented as a linear transformation:

$$\left[a^j_0 a^j_{0\dots 1} \dots a^j_{0\dots K-1}\right] = \left[a^0_i \dots a^{N-1}_i\right] \cdot J_N$$

where J_N is an $N \times N$ upper diagonal transformation matrix used to select the previous partial result for the reduction with the current vector value. By associativity of matrix multiplication, we can represent the series of transformations done at each step in the distributed computation.

$$J_N = \begin{bmatrix} 1 & 1 & 1 & \dots & 1 \\ 0 & 1 & 1 & \dots & 1 \\ 0 & 0 & 1 & \dots & 1 \\ \dots & \dots & \dots & \dots & \dots \\ 0 & 0 & 0 & \dots & 1 \end{bmatrix}$$

$$= \begin{bmatrix} 1 & 1 & 0 & \dots & 0 \\ 0 & 1 & 0 & \dots & 0 \\ 0 & 0 & 1 & \dots & 0 \\ \dots & \dots & \dots & \dots & \dots \\ 0 & 0 & 0 & \dots & 0 \\ 0 & 0 & 0 & \dots & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & 0 & \dots & 0 \\ 0 & 1 & 1 & \dots & 0 \\ 0 & 0 & 1 & \dots & 0 \\ \dots & \dots & \dots & \dots & \dots \\ 0 & 0 & 0 & \dots & 0 \\ 0 & 0 & 0 & \dots & 1 \end{bmatrix} \dots \begin{bmatrix} 1 & 0 & 0 & \dots & 0 \\ 0 & 1 & 0 & \dots & 0 \\ 0 & 0 & 1 & \dots & 0 \\ \dots & \dots & \dots & \dots & \dots \\ 0 & 0 & 0 & \dots & 1 \\ 0 & 0 & 0 & \dots & 1 \end{bmatrix}$$

According to Patarsuk et. al [101], the number of partial results to be communicated to complete a one-item all-reduce is 2 * (K - 1), where K is the number of nodes. By the previous decomposition, there exists a series of K - 1 transformations that leads to the one-item all-reduce. For any connected undirected graph G with K nodes, there exists a spanning tree with K - 1 vertices. A directed spanning tree (DST) of G rooted at R, is a subgraph T of G such that the undirected version of T is a tree and T contains a directed path from R to any other vertex in the set of vertices. [129] T is called the broadcast graph. Reversing the directed edges of T yields the gather graph, T'. Therefore, the K - 1 transformations will gather the result at the sink of T', following a broadcast in T.

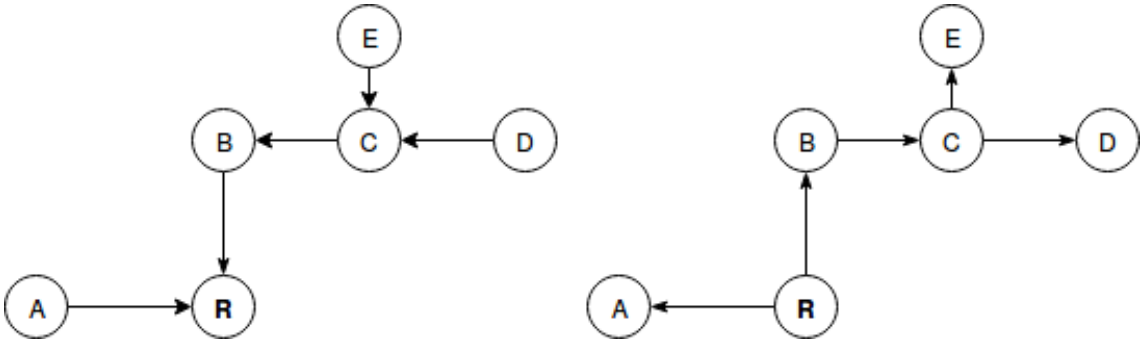


Figure A.1: Gather Graph with Sink R and Broadcast Graph with source R

We use the illustration in Figure A.1 to show how all-reduce works for one data item a_i with $i \in \{A, B, C, D, E, R\}$:

Step 1: All-gather

- C gathers values a_E and a_D. C computes intermediate value $a_C \oplus a_D \oplus a_E$
- B gathers intermediate value from C. B computes intermediate value $a_B \oplus a_C \oplus a_D \oplus a_E$
- R gathers value a_A. R computes final value $a_A \oplus a_R \oplus a_B \oplus a_C \oplus a_D \oplus a_E$
- R gathers intermediate value from B. R computes intermediate value $a_R \oplus a_B \oplus a_C \oplus a_D \oplus a_E$

Note that a non-root node does not send the intermediate result until it has received a message from all upstream nodes.

Step 2: Broadcast

- R sends final value to A and B. Both A and B assign this value to the data-item under reduction
- B sends the final value to C. C assigns the final data-item value.

- C sends the final value to D and E. Both D and E assign the final data-item value.

Similarly, no node forwards assigns the final value and forwards any message unless it has been notified by the upstream node. The result of the distributed computation is that all nodes have the final data-item value $a_A \oplus a_R \oplus a_B \oplus a_C \oplus a_D \oplus a_E$.

A.3 Gradient noise estimation proposed by OpenAI

The implementation of the gradient noise scale is done according to Appendix A in the OpenAI paper [118]:

$$B_{simplified} = \frac{tr(\Sigma)}{|G|^2}$$

where Σ is the covariance matrix of the gradients across workers and the denominator represents the square of the L_2 norm of the gradient vector $|G|^2 = \sum_{i=1}^{D} |G_i^2|$. This can be approximated by $\frac{S}{|\mathcal{G}|^2}$.

The paper specifies that $\theta \in \mathbb{R}^D$. As the shape of the parameters tensor is the same as the shape of the gradients tensor from the TensorFlow implementation perspective, it follows that $G \in \mathbb{R}^D$.

S and $|\mathcal{G}|^2$ are computed at every iteration:

$$|\mathcal{G}|^2 = \frac{1}{B_{big} - B_{small}}(B_{big} \cdot |G_{B_{big}}|^2 - B_{small} \cdot |G_{B_{small}}|^2)$$
$$S = \frac{1}{1/B_{small} - 1/B_{big}}(|G_{B_{small}}|^2 - |G_{B_{big}}|^2)$$

The computation is suitable for the collective communication paradigm of Kungfu in which the all-reduce operation is applied across devices. As such, B_{small} represents the batch size per worker which is constant for all workers, B_{big} is the aggregated batch size equal to $workerCount \cdot B_{small}$, $G_{B_{small}}$ is the gradient locally computed by each worker and $G_{B_{big}}$ is the negotiated gradient obtained through all-reduce.

The values S and $|\mathcal{G}|^2$ computed at every iteration are not unbiased estimators for $tr(\Sigma)$ and $|G|^2$ respectively. To obtain unbiased estimators, we need to calculate the exponentially moving average S_{EMA} and $|\mathcal{G}|^2_{EMA}$ as follows:

$$|\mathcal{G}|^2_{EMA_t} = \alpha|\mathcal{G}|^2_t + (1-\alpha)|\mathcal{G}|^2_{EMA_{t-1}}$$
$$S_{EMA_t} = \alpha S_t + (1-\alpha)S_{EMA_{t-1}}$$

where α represents the decay coefficient that weighs the contributions of the currently calculated value and of the previous exponentially moving average. It is an additional hyperparameter which needs to be tuned. The unbiased estimation of the noise scale is $\hat{B}_{noise} = \frac{S_{EMA}}{|\mathcal{G}|^2_{EMA}}$.