ELASTIC SYNCHRONIZATION FOR EFFICIENT AND EFFECTIVE DISTRIBUTED DEEP LEARNING

XING ZHAO

A THESIS SUBMITTED TO
THE FACULTY OF GRADUATE STUDIES
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTER OF SCIENCE

GRADUATE PROGRAM IN ELECTRICAL
ENGINEERING AND COMPUTER SCIENCE
YORK UNIVERSITY
TORONTO, ONTARIO

JULY, 2020

Abstract

Training deep neural networks (DNNs) using a large-scale cluster with an efficient distributed paradigm reduces the training time from weeks on a single server to hours. However, an “efficient” distributed paradigm developed only from system engineering perspective is most likely to hindering the model from learning since it fails to consider the intrinsic optimization properties of machine learning. In this thesis, we present two efficient and effective models in the parameter server setting based on the limitations of the state-of-the-art distributed models such as staleness synchronous parallel (SSP) and bulk synchronous parallel (BSP) models.

We introduce dynamic staleness synchronous parallel (DynamicSSP) model that adds smart dynamic communication to SSP, improves its communication efficiency and replaces its fixed staleness threshold (a hyperparameter) with a dynamic threshold in a given range. DynamicSSP converges faster and to a higher accuracy than SSP in the heterogeneous environment. Having recognized the importance of bulk synchronization in training via experiments of running large DNNs on large datasets, we also propose the elastic bulk synchronous parallel (ElasticBSP) model which shares the proprieties of bulk synchronization and elastic synchronization. We develop fast online optimization algorithms with look-ahead mechanisms to materialise ElasticBSP. Empirically, ElasticBSP achieves the convergence speed 1.77 times faster and an overall accuracy 12.6% higher than BSP.

To my father for all the years of your love and support.

In memory of my mother

who always believed in my ability to be successful in the academic arena.

You are gone but your belief in me has made this journey possible.

Acknowledgements

I am grateful for my supervisor Prof. Aijun An for her continuous support and guidance throughout my Master's program, and for providing me the opportunity to work on the elastic deep learning project with IBM. I am also thankful for my committee member Prof. Ruth Urner for her support, guidance and fruitful conversation on the optimization problem of the distributed deep learning.

It is a pleasant experience for me to work and collaborate with my peer students, faculty and IBM researchers during the program of my Master's. For the work in this thesis, I enjoyed working with Bao Xin Chen, Prof. Manos Papagelis and my IBM colleague, Junfeng Liu. I very appreciated Manos for spending tremendous time on the collaboration of my project with IBM which resulted in a new chapter of this thesis. Thank you to all of the graduated students in the data mining lab for creating a friendly environment. You all will make me miss our lab's weekly seminar meeting and the interesting conversations.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

ADMM	Alternating Direction Method of Multipliers
ASP	Asynchronous Parallel
BSP	Bulk Synchronous Parallel
CNN	Convolutional Neural Networks
DNN	Deep Neural Networks
DRAM	Dynamic Random-Access Memory
DSSP	Dynamic Stale Synchronous Parallel
ElasticBSP	Elastic Bulk Synchronous Parallel
GPU	Graphics Processing Unit
ML	Machine Learning
PS	Parameter Server

P2P	Point to Point
SGD	Stochastic Gradient Descent
SIMD	Single Instruction Multiple Data
SSP	Stale Synchronous Parallel

Chapter 1

Introduction

1.1 Motivation

Deep learning is a popular machine learning technique and has been applied to many real-world problems, ranging from computer vision to natural language processing. However, training a deep neural network is very time-consuming, especially on large models and big data. It has become difficult for a single machine to train a large model over large datasets. A popular solution is to *distribute* and *parallelize* the training process across multiple machines using the parameter server framework. Like other job scheduling problems in distributed computing system, deep leaning tasks are also *bottle-necked* by the job dependencies in terms of the hardware utilization (i.e., computational) efficiency. It relies on the iterative convergent process (i.e., stochastic gradient descent works effectively in practice) to search the optimal parameters of the learning model.

Using STOCHASTIC GRADIENT DESCENT (SGD), a deep leaning model converges after a certain large number of iterations in each of which the model parameters (i.e., weights) are updated with the gradients of the lost function based on a mini-batch of samples. In distributed computing, SGD is running in parallel on multiple workers (machines) therefore the weight updates have to be synchronized per iteration so that all the workers can receive the same weights after the weight update. This weight synchronization per iteration ensures data (weights) consistency in distributed environment. It is also known as *synchronous SGD* [1] [2] in which the gradients from all workers are aggregated on the parameter server before the weights are updated per iteration. Its flip side is *asynchronous SGD* [3] [4] in which all workers running independently and no synchronizations are scheduled in training. The weights are updated on the parameter server once the gradients are available from any of the workers. We call this *asynchronous weight update.* ASYNCHRONOUS PARALLEL (ASP) [3] [5] is a straight distributed scheme of asynchronous SGD.

Synchronous SGD is strictly followed in BULK SYNCHRONOUS PARALLEL (BSP) model [6]. BSP makes the distributed system logically functioning as a single server, thus it guarantees the convergence of the model and can be used directly (or with minimum modification) by many non-parallel applications. This property endorses BSP the predominate model used in practice. Nonetheless, in BSP workers may take different computing time to calculate the gradients on a mini-batch due to their hardware configurations. The straggler makes the faster workers waiting for its completion of computing the gradients in an iteration. Consequently it elongates the training time and brings down the hardware utilization (more idling time for the faster workers). Meanwhile, the iteration throughput (the number of iterations processed per time unit) drops as well.

This raises the challenge of how to design an efficient synchronous parallel model for distributed deep learning training. Ho et al. [7] proposed STALE SYNCHRONOUS PARALLEL (SSP) which allows some constrained asynchronous parallel along the iterations in training. It allows workers running independently but uses a staleness threshold s to control that the fastest worker cannot run more than s iterations ahead of the slowest worker (the straggler). Thus, weight synchronization happens between the fastest and the slowest workers when the threshold s is exceeded, otherwise asynchronous weight updates dominate the iterative convergent process in training and render SSP behaves as ASP. However, asynchronous weight updates generate staled gradients which are harmful to the convergence [7] [8] and may lead to divergence [9]. For example, the gradients computed by the straggler based on the weights from the 2nd iteration are updated to the weights of the 5th iteration of the fastest worker which changes the direction of the original step. Intuitively, we know ASYNCHRONOUS PARALLEL (ASP) maximizes the hardware utilization of all workers. Its high computational efficiency expectedly brings down the training time since all workers are running independently across the entire training and no weight synchronizations are required. Yet theoretically, ASP lacks convergence guarantee [10] because of the staled gradients we mentioned. Despite BSP guarantees the convergence, it has the straggler problem. SSP tries to mitigate the straggler problem by adding some elasticity to BSP. It relaxes the rigid requirement of BSP that all the workers have to wait for each other by the end of each iteration for weight synchronization by allowing weight synchronization happens only when the staleness threshold is exceeded between the fastest worker and the slowest worker and only requiring weight synchronization takes place between both types of workers.

1.2 Contributions

From system engineering's perspective, ASP is the ideal choice to achieve the maximum hardware utilization. From machine learning's perspective, BSP is the safe and stable model to use for distributed training for deep learning. The core determinant that creates such a dilemma lies in the iterative convergent optimization process (i.e., SGD) used in deep learning model training. The purpose of this thesis is to speed up the convergence speed of the iterative convergent process, SGD in parallel and thereby develop efficient distributed training for deep learning with *elastic synchronization*. We materialize the elasticity by developing dynamic and adaptive online techniques which aim to elevate the efficiency of the prior synchronous parallel models of distributed training (i.e., SSP and BSP) while maintaining or even improving the convergence speed and the final accuracy by taking advantage of the fault torrent property of machine learning [11]. What's more, the staled gradients are generated from slower workers during the training of a large number of iterations in the heterogeneous distributed computing environment. We design the elasticity following the theoretical analysis on the impact of the dynamic changing of the staled gradients in training on the convergence of DNN models to ensure the efficacy of our proposed models.

In this thesis, we explore three different types of the state-of-the-art distributed parallel models (i.e., BSP, SSP and ASP) used in practice and their scheduling to the iterative convergent process of parallel SGD (including synchronous and asynchronous SGD) in training. In particular, we analyze the elasticity that SSP has comparing to BSP and the trade-offs between SSP and BSP. Then, we extend the elasticity of SSP further and propose DynamicSSP by adding run-time information of workers to the elasticity. Unlike SSP which simply counts the number of iterations of workers to compute the staleness threshold, DynamicSSP is aware of the workers' computational capacity on a mini-batch at run-time and therefore adjusts the threshold dynamically with the goal of minimizing the idling time of the fastest worker.

Having observed the behavior and the performance of DynamicSSP, we try to interpret the role of staled gradients is playing in parallel SGD: a regularizer by adding gradient noise to the optimization of the objective function. When the staled gradients are accumulated to a large amount in training, they postpone the convergence and even causes the convergence uncertain. But with careful controlling on the degree of the staleness and the amount of the staled gradients, the convergence rate can be boosted and are prone to reach a higher accuracy in practice.

In the extensive experiments on evaluating DynamicSSP, we observed SSP and ASP were struggling to learn and converging very slow on the large-sized DNN models (e.g., Vgg-16 [12]) on the large dataset (e.g., IMAGENET-1K [13]). To the contrary, BSP achieved a steady increasing convergence (learning) curve on the test accuracy. Hence, we start considering to design a distributed parallel model like BSP but having the ability of controlling the dynamic changing of staled gradients generated by slower workers in training, in particular, limiting the degree of staleness and the amount of staled gradients using synchronization. In addition, we want to take advantage of the dynamic technique from DynamicSSP that reduces the waiting time of the fastest worker in training.

Accordingly we propose ElasticBSP which converges faster and to a higher accuracy than BSP empirically. In the case of training large deep neural networks (with fully connected layers) on the large dataset where ASP, DynamicSSP and SSP fail to learn or deliver the same convergence rate as BSP, ElasticBSP demonstrates its superior performance than those including BSP in the experiments. ElasticBSP is a novel synchronization model for scaling the training of distributed deep learning models. ElasticBSP replaces the strict synchronization requirement of other BSP-like models with an online decision making about the best time to impose the next synchronization barrier. The model guarantees the convergence when the number of iterations in training is large. The empirical evidence shows that ElasticBSP offers strong generalization ability as BSP but better accuracy.

Parts of the thesis have been published in the following conference papers:

- Zhao, X., An, A., Liu, J., & Chen, B. X. (2019). Dynamic Stale Synchronous Parallel Distributed Training for Deep Learning. In Proceedings of the 39th IEEE International Conference on Distributed Computing Systems, pp. 1508-1517 (ICDCS 2019). [14] (included in Chapter 5)

- Zhao, X., Papagelis, M., An, A., Chen, B. X., Liu, J., & Hu, Y. (2019). Elastic Bulk Synchronous Parallel for Distributed Deep Learning. In Proceedings of the 19th IEEE International Conference on Data Mining, pp. 1504-1509 (ICDM 2019). [15] (included in Chapter 6)

1.3 Thesis Structure

The thesis consists of 7 chapters.

- **Chapter 1** introduces the motivation of this thesis and briefs our contributions.

- **Chapter 2** introduces the basic background of the distributed environment for deep learning, and describes the optimization process for distributed DNN training and the definition of convergence speed based on wall-clock time unit.

- **Chapter 3** reviews the three essential yet prevalent state-of-the-arts distributed paradigms in practice — BSP, ASP and SSP.
- **Chapter 4** discusses the challenges of optimizing the parallel models for efficient yet effective distributed training and points out the synchronizations in distributed deep learning could be optimized with respect to the elasticity.
- **Chapter 5** presents Dynamic Stale Synchronous Parallel model which provides dynamic online staleness threshold determination based on the information collected from workers so that this model can adapt to the local environment at run time.
- **Chapter 6** introduces Elastic Bulk Synchronous Parallel model which finds the optimal synchronization time for all workers at run time and offers significantly better convergence speed and accuracy than Bulk Synchronous Parallel model empirically.
- **Chapter 7** concludes the thesis by addressing the significance of the presented works, illuminating the insight of staled gradients in the SGD optimization, and provides potential future direction on the topic of this thesis.

Chapter 2

Basic Background

2.1 Topology of distributed computing system

The distributed computing system can be designed in two topology structures: centralized structure and decentralized structure.

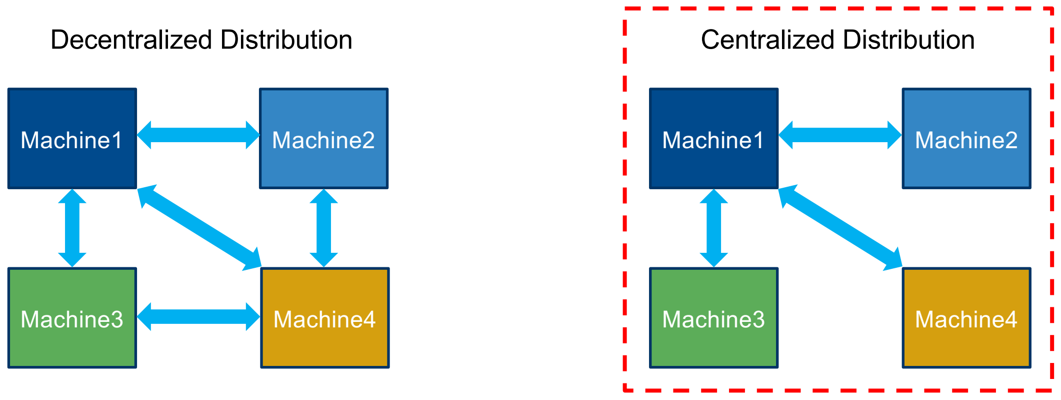


Figure 2.1: Topology of distributed computing system. Parameter server framework is in centralized distribution structure.

2.1.1 Centralized Distribution Structure

In *centralized structure*, client machines only communicate to server machine. Client machine is also known as worker in distributed machine learning literature [11] [16] [17]. This structure uses the least communication connections among available machines ($n - 1$ connections for n machines) and can be easily scaled to a large size of a server farm. The most unpleasant drawback of this structure is the single point of failure where the only server goes offline. The common workaround in practice is to have standby backup servers ready for failover and or to distribute a logical server into multiple physical servers. Even with standby backup servers for failover, a single point of failure will cause the loss of the gradients of the current computing iterations from all the workers on the server side. Re-transmission of gradients from all the workers is required. To satisfy the parallel (iterations) tasks with continuous dependency (between the current gradients and the previous weights) and the highly computation-intensive training for deep learning where each iteration is expensive to compute for a large model, both aforementioned workarounds are necessary in the design of the distributed training system for deep learning. For example, the ring structure [18] of parameter servers in MXNet [19] is able to resist the server failure. In the ring structure, weights, i.e., the original ring is first segmented into k parts, then each part are replicated to m copies (m rings). m rings share the same center point. Each ring is shifted in different directions to ensure their each k part is partially overlapped with its neighbours of the other $m - 1$ rings. Thereafter, the shifted m rings are segmented into n parts where n is the number of servers and each server is in charge of maintaining each part.

2.1.2 Decentralized Distribution Structure

In *decentralized structure*, machines communicate to each other directly and no centralized server exists in the structure. The communication connections cost of the structure grows in polynomial ($\frac{n^2-n}{2}$ connections required for n machines). Thus, its scaling cost is expensive. This structure has other names: P2P (point to point) and C2C (client to client). Its worst case, *full mesh* where all machines are connected to each other, is infeasible to scale due to the cost of $\frac{n^2-n}{2}$. *Partial mesh* is usually encouraged in practice where each machine only connected to few others. It has been the topic of much research on the optimization of partial mesh of decentralized structure as the prevailing of the distributed deep learning for large scaled network, for example, [20] and [21].

In this thesis, we focus on the centralized distribution structure into which the parameter server framework is categorized.

2.2 Parameter Server Framework

Parameter Server Framework [3, 7, 18, 22, 23] has been widely adopted to the distributed training platform [17, 22, 24, 25] for deep learning since 2012 [3] in which it is called DistBelief that uses large clusters with more than 5000 computers to distribute the training of large deep neural networks. The framework (see Figure 2.2) consists of multiple *workers* and a logical *server* which can be distributed into multiple physical servers for load balancing [26] in the case that the number of workers and the size of the training model are too large. Workers are all connected to the server and are responsible for the heavy computing tasks such as convolutional computing and backpropagation. The server usually maintains the *globally shared weights* by aggregating gradients from all the workers and updating the global weights. No complicated computing tasks are executed on the server. It provides a central storage for the workers to upload their computed gradient updates (by the `push` operation) and fetch the up-to-date global weights (by the `pull` operation). The parameter server framework supports two approaches on the deep neural networks (DNNs) distributed training: *model parallelism* and *data parallelism* [27]. Figure 2.3 illustrates a comparison on both parallelisms.

2.2.1 Model Parallelism

Model parallelism is to partition a large size DNN model into small parts and distribute the small parts to multiple computing resources (workers) for parallel training (see Figure 2.3). Each worker computes the gradients for the server based on its assigned model partition and training data. In [3], a large size DNN model is segmented into 144 partitions and trained in 32 machines concurrently. Due to

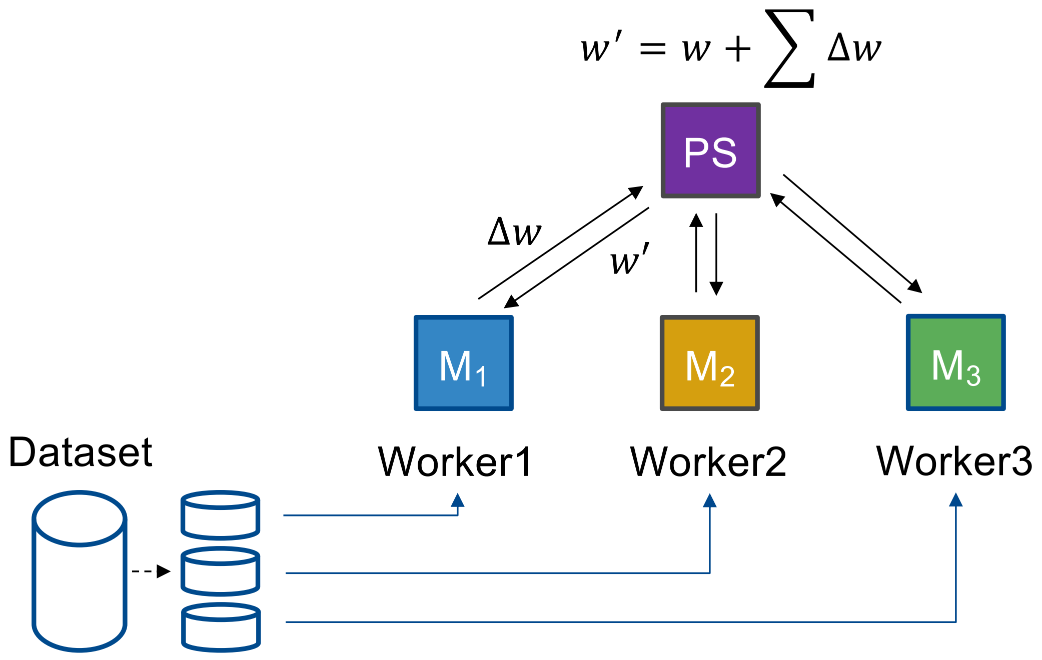


Figure 2.2: Parameter Server Framework for Data Parallelism. A deep learning model is replicated to all the workers whereas the model parameter w is maintained and updated by the parameter server (PS). The training dataset is partitioned into subsets and each subset is assigned to each worker. In each iteration, workers first compute the gradient Δw based on w and its assigned subset, then send Δw to the PS. PS aggregates Δw from all the workers, then updates w and sends updated w' to each worker for the next iteration.

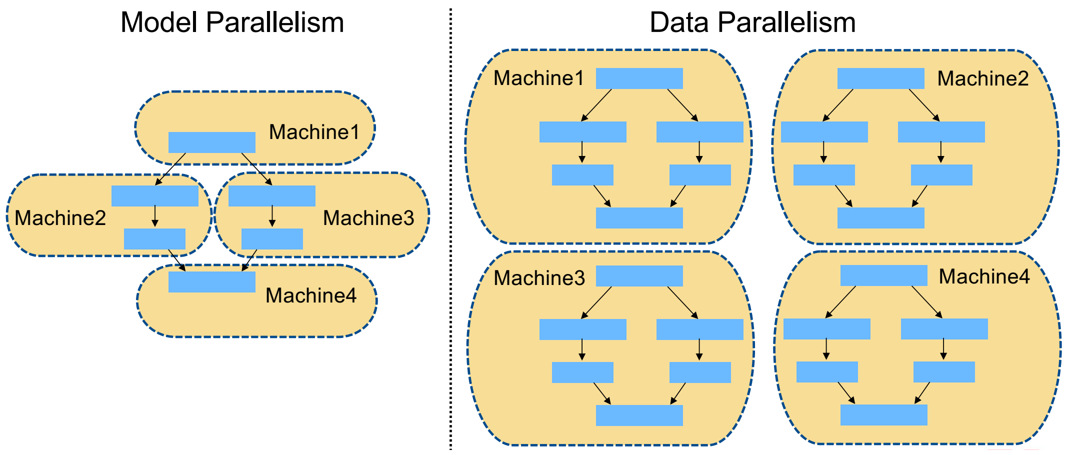


Figure 2.3: Model Parallelism and Data Parallelism.

the complicated dependencies between layers of parameters in contemporary DNNs, the nature of the iterative convergent optimization method (e.g., stochastic gradient descent) [11] and predominant use of multiple GPUs [23] [28] [29] [30], it is rather difficult to decouple the parameters of inter-layers or intra-layers for efficient concurrent training such as Vgg [12], ResNet [31] and Transformer [32].

The use of GPUs accelerates the DNNs training speed since GPU is specialized in SIMD (single instruction multiple data) parallel processing for large data. However, GPU has the data loading bottleneck [33] (overhead) on batching and moving the data in and offload from its computing girds (GPU shared memory) to computer global memory (DRAM). Additionally, the inter computer communication is bottle-necked by the connection bandwidth. It takes too long for some partitions in one computer to waiting for the output of their dependent partitions in another computer in just one time forward computing, not to mention that one iteration consists of forward and back-propagation computing for the gradients as well as large size DNN models training require a large number of iterations to converge. Model parallelism approach will only be advantageous when partitions of a model have loose dependency and can be trained almost concurrently such as the linear classifiers of which features can be assumed independent from each other. Otherwise, model parallelism is rather difficult to gain any benefit for large DNNs due to the dependency between layers. Thus, model parallelism has been rarely seen in practice for distributed deep learning. In this thesis, we focus on data parallelism.

2.2.2 Data Parallelism

Data parallelism (see Figure 2.3) has been prevailed in industry [34] [35] and implemented in many applications (e.g., Malt [36], SparkNet [37] and Poseidon [17]). It is very effective to handle the large size of the training data and easy to implement. The popular deep convolutional neural networks [13] [31] benchmark datasets such as COCO [38] has data size 44GB and IMAGENET-1K [39] has data size 150GB. *Data parallelism* replicates the entire DNN model to each worker but shards the training data into equal size (depend on the number of workers) and uniformly assigns each data shard to workers (see Figure 2.2). During the training, each *worker* trains the replica model using its assigned data shard, sends the computed *gradients* to the server (via `push` operation) and retrieves the most recent updated *weights* from the server (via `pull` operation). The *server* maintains a *global model weights* and updates the weights with the received *gradients*.

2.3 Iterative Convergence Optimization — SGD

In training, a DNN model searches the optimal parameters by minimizing the loss (of loss function) based on the given training data via the iterative convergent process, *stochastic gradient descent* (SGD) — the most effective optimizer used in practice for deep learning. SGD iteratively updates the model parameters $\mathbf{w}$ [40] as follow: $\mathbf{w}^t = \mathbf{w}^{t-1} - \eta \cdot (\mathbf{g}^{t-1} + \partial\Phi(\mathbf{w}^{t-1}))$. t represents the t_{th} iteration, $\mathbf{g}$ is the gradients of the chosen loss function with respect to the parameters $\mathbf{w}$ and Φ is the regularizer. In each iteration t, $\mathbf{g}^t$ is computed based on the loss function l and a mini-batch of the training data such that $\mathbf{g}^t = \frac{1}{m}\sum_{i=1}^{m} \partial l((x_i, y_i), \mathbf{w}^t))$ where m is the size of a mini-batch.

2.4 Parallel SGD

Parallel SGD [40] refers to SGD running in a distributed environment where the model (weight parameters) updates become tricky and complicated. Assuming a *distributed environment* and the *data parallelism* approach of a *parameter server framework* are used as shown in Figure 2.2, the entire DNN model is replicated to *n workers*, while the *server* maintains globally shared weight parameters. During the training, each *worker* trains the replica model using its assigned data shard, sends the computed *gradients* to the server (via a `push` operation) and retrieves updated *weights* from it (via a `pull` operation) per iteration. Different schemes of scheduling these communication operations of workers and the weight updates on the server will result in different learning performance of DNN models such as the convergence speed and the accuracy. We will introduce different communication schemes between workers and the server and discuss their performances with respect to the convergence in Section 3.1. Before that, we need to clarify the notion of convergence speed we use in the distributed parallel computing setting.

2.5 Convergence Speed in Wall-clock Time Unit

Conventionally, convergence speed [41] (also known as rate of convergence) is to measure how fast a model approaches its limit (with respect to accuracy or loss) in a number of iterations (by iteration unit). The less iterations required by a model to reaching its limit (regardless a good or bad solution), the better convergence speed the model has[1]. Convergence speed is one of the two crucial metrics to determine the efficacy of an optimizer. The other one is generalization which evaluates the performance of the model on new data. An optimizer demonstrates consistent good performance on different models and different datasets is considered having strong generalization ability [42]. This notion of generalization is orthogonal to the computing environment so we continue using it as is in the distributed setting.

The original notation of convergence speed was used to measuring the learning efficacy for machine learning in a non-parallel computing setting. Later, it was used for measuring learning performance of DNN models running serial SGD on a single computer. For distributed deep learning, a DNN model is trained in multiple computers (or workers). In data parallelism, a DNN model is replicated to each worker and the gradients of each worker are sent to the parameter server on which the weight updates is executed. As workers are sending gradients to the parameter server without synchronization per iteration, the weight updates on the server be-

[1]Note that a good convergence speed does not imply a good final accuracy. A fast convergence may end up at a very low accuracy or high loss due to the complexity of the model and or a problematic optimizer.

come random since weight update occurs whenever gradients of any worker arrive. One weight update (on the server) no longer represents an iteration of the entire system and is decoupled from the local iterations of all the workers. Thus, it is not reasonable to use convergence speed of original notion as a metric to evaluate DNN models in distributed setting. It is also not realistic to compute convergence speed of original notion for distributed deep learning since one worker may run more iterations than another in any time. Therefore, we cannot use the conventional measure of convergence speed used on a single computer for the parallel SGD in a distributed parallel computing setting. Instead of measuring the convergence speed based on *iteration unit* — the original notion, we define *convergence speed* of distributed parallel computing setting measures the speed a DNN model approaches its limit (convergence) of test accuracy based on wall-clock time unit (a time period).

Chapter 3

Literature Review

3.1 Parallel Paradigms for Distributed Training

There are three essential and prevalent distributed paradigms for updating the model parameters under data parallelism using parallel SGD in training. They are Bulk Synchronous Parallel (BSP) [6], Stale Synchronous Parallel (SSP) [7] [43] and Asynchronous Parallel (ASP) [3] [5] [44]. BSP is predominantly practised in industry and is supported by Pytorch, TensorFlow and MXNet. ASP is supported in TensorFlow and MXNet. SSP is exclusive to Petuum. Other state-of-the-art methods are only available in the research field and are immature to the industry practice.

Historically, BSP-like paradigms generally exist in many parallel and concurrent computing programs for managing the multiple concurrent threads in an intra-computing setting (i.e., single machine). Each thread computes few lines of code in parallel but has to wait for other threads at a join point because of the dependency of the lines of code in context. This join point can be considered as a synchronization point of threads. The idea has also been successfully implemented into the distributed computing system with multiple machines. Meanwhile, its drawback — the waiting time of processing units (e.g., threads) is also inherited and gets worse when running concurrent threads in the inter-computing setting (i.e., multiple machines to which threads are distributed). A strangler (the slowest machine) makes the others wait for it at the synchronization point, which is noxious. The goal of SSP and ASP is to increase the computing time (for iterations) or to reduce the waiting time (of workers) by allowing more weight updates on the server without weight synchronization within a time period (which increases iteration throughput) and decreasing the workers' waiting time for communication in the parameter server setting [45]. The waiting time for communication can be further divided into the waiting time for synchronization of workers and the data (i.e., parameters) transmission time between workers and the server. We cover BSP, ASP and SSP paradigms respectively in detail in the next section and discuss their limitations. Aiming to solve the limitations, we propose our synchronization models in Chapter 5 and Chapter 6 respectively.

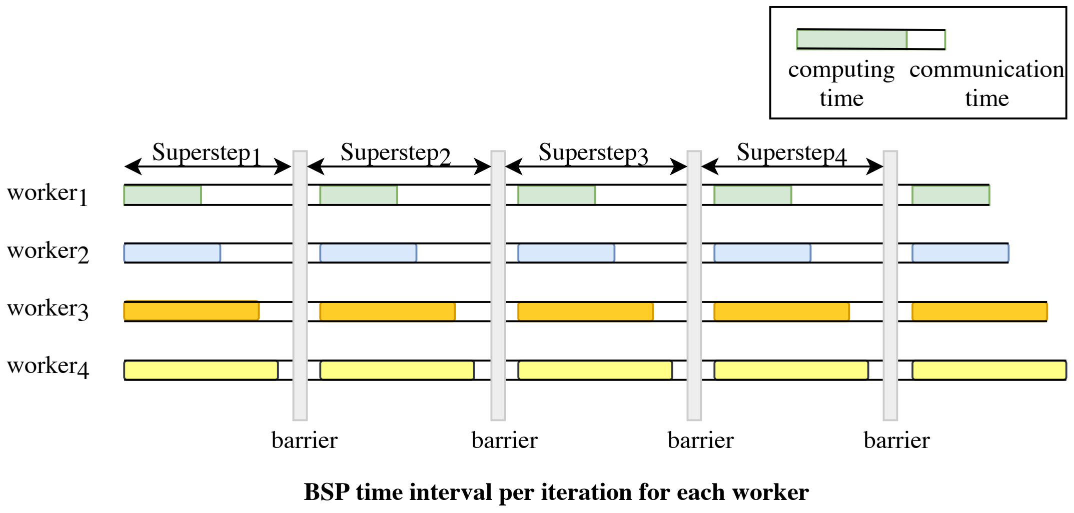


Figure 3.1: Vanilla BSP. All the workers have to wait for each other for synchronization at the end of every iteration. Barrier represents the weight synchronization. A superstep represents the interval between two contiguous weight synchronizations.

3.1.1 Bulk Synchronous Parallel (BSP)

In BSP, the server updates the weight w in two steps: First, it *aggregates* the gradients $g_p, p \in [1, n]$ from n workers in iteration $t-1$ such that $g^{t-1} = \frac{1}{n}\Sigma_{p=1}^{n} g_p^{t-1}$. Second, it updates the weight w^{t-1} for next iteration t,

$$w^t = w^{t-1} - \eta \cdot (g^{t-1} + \partial\Phi(w^{t-1})) \tag{3.1}$$

. After the weight update, it signals n workers that the weight w^t is available to `pull`. *Synchronization* of workers is necessitated but costs extra time on waiting for the straggler; that is because of the first aggregation step in which the server does not reduce the gradient g^{t-1} until it receives gradient g_p^{t-1} from *all* n workers. After the second step, all the workers receive the same (synchronized) weights. In this way, the server is controlling the gradient updates and makes parallel SGD logically function as serial SGD on a single computer. This property makes BSP easily be implemented into any existing serial SGD applications. The synchronization in BSP guarantees the data consistency of w as shown in Figure 3.1. This strategy works very well when all the workers have similar speed. However, the *waiting time* (in the first step) contribute an inevitable overhead to the training when workers have difference processing speed on an iteration.

With respect to the popularity, BSP is the predominant distributed model in applications of many disciplines other than ML [46] [47]. Its most profound application in the distributed framework field is MapReduce [48] [49]. Most general propose frameworks such as Hadoop [50] and Spark [51] all use MapReduce-like parallel training. That is, all machines (also called workers in distributed training framework) have to wait for each other to finish the computing and updates by the end of each iteration for synchronization. A significant amount of time is wasted in such model when workers have different processing speed.

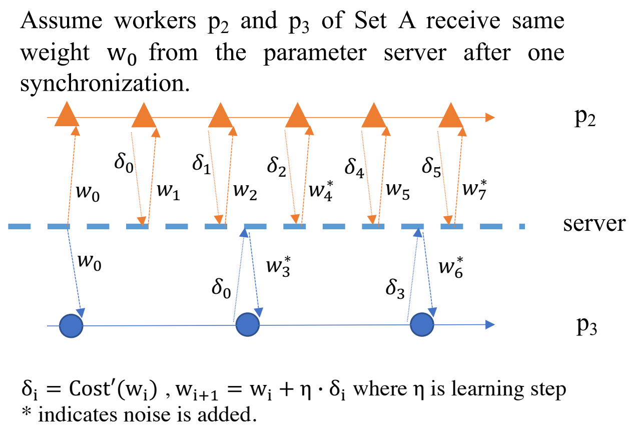
Nonetheless, since ML has the fault tolerant property [11] (that is, it is robust against minor errors in intermediate calculations) when it uses the iterative convergent optimization algorithm such as stochastic gradient descent (SGD) [40], a more flexible paradigm that uses less (i.e., sparse) synchronizations can be applied. This property is crucial to the topic of speeding up the distributed deep learning and is therefore taken into the consideration in the design of our proposed models.

3.1.2 Asynchronous Parallel (ASP)

ASP [3] [5] [44] is an alternative approach on updating the weight w. It is based on the idea of the workers working *independently* and the server updating the weights immediately as soon as any gradient becomes available. In this fashion, weight synchronization is not required in training and the waiting time of workers is thus eliminated. With no synchronizations, the weight updates on the server become *asynchronous gradient updates* which is potentially harmful to the convergence. See the examples in Figure 3.2(a) and 6.3. It appears as if ASP is better than BSP since it eliminates the control costs and the waiting time overhead. Besides, the number of weight updates of ASP becomes n times more than that of BSP for n workers (by equation 3.1 and 3.2), which means ASP enjoys larger iteration throughput. However, the weight updates become chaotic:

$$w^t = w^{t-1} - \eta \cdot (g_p^{s-1} + \partial\Phi(w^{t-1})), p \in [1, n], s \leq t \tag{3.2}$$

Here t represents the iteration of the fastest worker(s) whereas s is the iteration of the slower worker(s). The gradients computed at iteration $s = t - c$ by the slower worker(s) then become stale for c iterations, where c denotes the *iteration difference*; c also represents the *staleness value* of the staled gradients of the slower worker(s). If c is large, the staled gradient functions as noise in the iterative weight updates (see Figure 6.6(a)); when there are many c-staled gradients present, it renders the


Assume workers p_2 and p_3 of Set A receive same weight w_0 from the parameter server after one synchronization.
p_2
server
p_3
δ_0 δ_1 δ_2 δ_4 δ_5
w_0 w_1 w_2 w_4^* w_5 w_7^*
w_0 δ_0 w_3^* δ_3 w_6^*
$\delta_i = \mathrm{Cost}'(w_i)$, $w_{i+1} = w_i + \eta \cdot \delta_i$ where η is learning step
* indicates noise is added.

(a)

Workers	Set A	Set B
p_1	13	3
p_2	20	5
p_3	10	1
p_4	12	3
p_5	17	2
Iteration difference: max – min	10	4 ✓

(b)

Figure 3.2: (a) shows how staled gradients δ bring noise to the weight update. The noise may drift the weight convergence away from the optimal direction (e.g., local minima of loss function). (b) Suppose we have two sets of workers for training. Each column lists the iterations of each worker completed at time t. The set with smaller iteration difference has fewer staled gradients added to the weight. The staleness of gradients is bounded by the iteration difference.

convergence *uncertain* [10]. Without synchronization, w is *not consistent* to workers in training, each worker may receive different w at each iteration. Consequently, their gradients are computed based on different w since $g = \partial l(x, y; w)$ where l represents the loss function and (x, y) is the training data. It appears as if workers with different w are converging towards different directions. From optimization perspective, gradients in ASP at any time step are pointing to dissimilar directions, the convergence rate may be slow down [7] or divergence may happen. Staled gradients may drift the convergent step away from the original route of reaching local minima or global minimum.

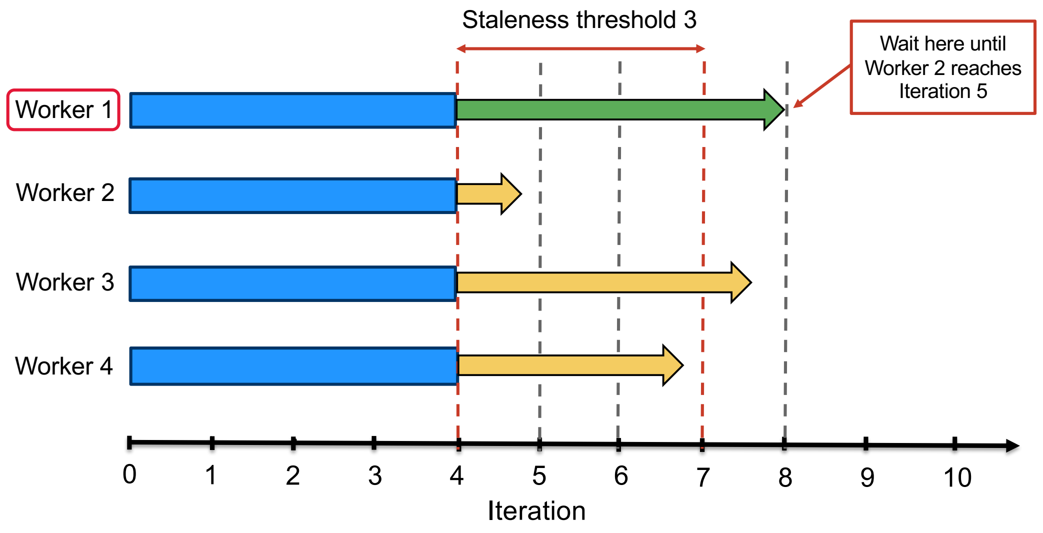


Figure 3.3: Stale synchronous parallel with the staleness threshold $s = 3$. The fastest worker, Worker 1 has to wait at iteration 8 till the slowest worker, Worker 2 reaches iteration 5.

3.1.3 Stale Synchronous Parallel (SSP)

SSP [7] [43] is a combination of BSP and ASP or a compromised solution of BSP to save some waiting time of the fastest worker. It allows workers running independently but ensures that the fastest workers do not run β iterations more than the slowest workers as shown in Figure 3.3. In other words, SSP uses a threshold β to constrain the iteration difference c among workers at any time step during the training so that $c \leq \beta$ where $c = t - s$ in equation 3.2. Note that β has to be fixed to a small number, otherwise SSP might behave as ASP. Since c is bounded by a small β, the harm that the staled gradients introduce to the convergence is reduced or upper bounded by β. In [7] authors provide theoretical analysis to prove that SSP guarantees convergence for a large number of iterations with a small β.

SSP model works as follow: it counts the number of iterations of each worker has completed. When the fastest worker is running at its t iteration and the slowest worker is running at its $t - \beta - 1$ iteration, the fastest worker has to waiting for the slowest worker reaches its $t-\beta$ iteration. In this way, SSP manages a synchronization between the fastest worker and the slowest worker by setting a synchronization barrier to stop the fastest worker running further upon the excess of the threshold. After the synchronization, the two workers compute the gradients based on the same weight w in a new iteration. Since β is small, the model converges near the late stage of the training for a large number of iterations [7].

Chapter 4

Challenges of Optimization for Distributed Deep Learning

4.1 Optimization for Distributed Deep Learning

In the previous chapter, we learned the advantages and the limitations of the three prevalent distributed models under the parameter server framework. When it comes to the accuracy, any given distributed model will be upper bounded by BSP and be lower bounded by ASP because of the rigorous synchronization principle of BSP and zero synchronizations requirement of ASP, in other words, the amount and the degree of the introduced staled gradients in training. Likewise, ASP upper bounds and BSP lower bounds any given distributed synchronous model with respect to the training time for a fixed epoch task. Should one simply set the goal of achieving the shortest (fastest) training time to complete a fixed epoch task, ASP is the winner. But that is not the goal we want to achieve due to the potential divergence of ASP we discussed in Section 3.1.2. In addition, large DNN models with fully connected layers can barely learn anything under ASP empirically which we will demonstrate in Section 6.6.2.

Ideally, we want to develop a distributed synchronous model that achieves high final test accuracy (as BSP) and fast speed to completing a fixed epoch task (as ASP). In reality, it is infeasible to implementing the advantages of both BSP and ASP into one hybrid distributed model due to their respective paradoxical intrinsic proprieties (contradictory nature of job schedules on the workers). Quite a few approaches in the literature only manage to either speed up the convergence of BSP [1] [41] [52] [53] or improve the accuracy of ASP [5] [54] [55] [56]. For instance, SSP [7] is a compromised version of BSP which trades off the accuracy for the speed by allowing workers partially running ASP under a staleness threshold restriction.

Nonetheless, in this thesis we will work on balancing the *quality* of iteration [7] and the *quantity* of the iterations within a time period (e.g., a period between consecutive synchronizations) to improve the convergence speed without sacrificing the accuracy. Remember the fault tolerant propriety of machine learning [11] we mentioned in Section 3.1.1. Also remind equations 3.1 and 3.2. It costs (waiting) time to maintain the quality of iteration as we see all the workers waiting for each other per iteration in BSP for the *consistency of weight updates* (i.e., consistent steps to the convergence). However, we can approximately maintain the quality of a large portion of the iterations of the entire system in training by ensuring a large portion of the workers not dropping below the minimum necessary synchronizations (or the maximum threshold of the fault tolerance of machine learning) so that the convergence speed is increased without sacrificing the accuracy (quality of learning). We expect that this may cause the increase of accuracy per iteration slows down (such as a slower learning curve) compared to the original one (of using BSP) and the final accuracy falls a little below the original one at convergence.

Knowing the importance of the quality of iteration (i.e., the quality of weight updates determined by the dense of synchronizations), we also ought to point out that increasing the iteration throughput does not necessary improve the quality of iteration. A fallacy that some system engineers believe is: to speed up the DNN model training is to increase the iteration throughput. Here we compare the number of weight updates on a training model given a fixed number of iterations T and the mini-batch size m and the total number of worker n in Table 4.1. It shows increasing the iteration throughput does not increase the accuracy since ASP can have the maximum iteration throughput in an ideal distributed environment but has problem to learn with zero synchronizations (which introduce a significant amount of staled gradients) in training.

Paradigms	# updates to w	# samples per update	Synchronizations	Accuracy	Training time
BSP	T	nm	T	high	maximum
ASP	nT	m	0	low	minimum

Table 4.1: Training a DNN model w with n workers for T iterations on the parameter server framework. Each worker has mini-batch size m.

4.2 Optimizing Synchronization and Limitation

The extra training time cost of the distributed deep learning in the parameter server setting comes from the *synchronizations* (which introduce the waiting time). It gets worse when the size of the workers is scaling up. With BSP, the most restrictive synchronization policy is employed (all workers have to wait for each other at the end of each iteration) which yields the maximum waiting time of the workers. To the contrary, ASP does not require synchronization at all which costs minimum (i.e., zero) waiting time of the workers. To speed up the training while not sacrificing the accuracy, we can optimize the synchronizations in training — to reduce the frequency of synchronization without harming the accuracy (thanks to the fault tolerance property of ML). Since the frequency of synchronization determines the amount and the degree of staled gradients as we mentioned in Section 3.1 and too many staled gradients may cause the divergence, we can maximize the sparsity of the synchronizations to a level where the number of staled gradients are limited to not jeopardising the convergence[1]. For example, BSP has the densest synchronizations in training so that it has zero staled gradients and achieves the highest accuracy

[1]The exact level of the sparsity of the synchronizations and the "hard" limit point of staled gradients are left as open questions for the future work since they are non-trivial and specific to the optimization area.

compared to any distributed model. SSP reduces the synchronizations to a level of sparsity where the staled gradients are limited to (or bounded by) the user specified threshold β, e.g., no staled gradients are staler than β iterations in training. In SSP, synchronization occurs between the fastest and the slowest workers in every β iterations, as a result, SSP converges faster but to a lower accuracy than BSP with a small β for a large number of iterations [7]. Nonetheless, the sparsity of the synchronizations in SSP is not yet optimized for at least two reasons. First, the threshold β is a fixed hyperparameter which requires fine-tuning. Second, the local (workers) computational capacities are varied in reality and not yet considered in determining the optimal synchronization time point of minimizing the waiting time of the fastest worker. We claim the threshold can be dynamically determined based on the workers' computational capacities at run time so that the waiting time of the fastest worker can be minimized. Hence, we introduce DynamicSSP model in Chapter 5 which optimizes the sparsity of the synchronizations in SSP or further minimizes the waiting time of the fastest worker. DynamicSSP enables dynamic threshold determination that adapts to the running environment by incorporating the information of the workers' computational capacities at run time.

Having observed the performance of DynamicSSP, SSP and BSP models in a variety of experiments, we learn the importance of the synchronizations in training to the convergence and the final test accuracy. Empirically, higher frequency of (dense) synchronizations usually gives better accuracy but has slower convergence speed, that is, more training time is required to reach a desired accuracy due to the waiting time of workers. SSP relaxes the rigid synchronization principle of BSP and imposes synchronizations only on the fastest and the slowest workers upon excess of the staleness threshold. The bulk (i.e., all workers) synchronization is absent in SSP. In the case of training a large DNN model with fully connected layers on a dataset with high dimensional samples, BSP is the only one that is able to train such DNN models and converge to a high test accuracy despite the cost of waiting time. Empirically the DNN model with fully connected layers shows its sensitivity to the staled gradients [14] [15]. Therefore, the bulk synchronization is crucial to such DNN models training. This observation gives us light to develop a novel distributed paradigm based on BSP that keeping the notion of bulk synchronization whilst optimizing the sparsity of the synchronizations. Thereupon, we propose ElasticBSP model with the aforementioned two properties in Chapter 6.

Chapter 5

Dynamic Stale Synchronous Parallel Model

5.1 Introduction

In Section 3.1, we introduced BSP, ASP and SSP models. We know that SSP is an intermediate solution between BSP and ASP. It is faster than BSP, and guarantees convergence, leading to a more accurate model than ASP. However, in SSP the user-specified staleness threshold is fixed, which leads to two problems. First, it is usually hard for the user to specify a good single threshold since user has no knowledge which value is the best. Choosing a good threshold may involve manually searching in an integer range via numerous trials. Also, a DNN model involves many other hyperparameters (such as the number of layers and the number of nodes in each layer). When these parameters change, the same searching trials have to be repeated again to fine-tuning the staleness threshold. Second, a single fixed value may not be suitable for the whole training process. An ill-specified value may cause the fastest workers to wait for longer time than necessary.

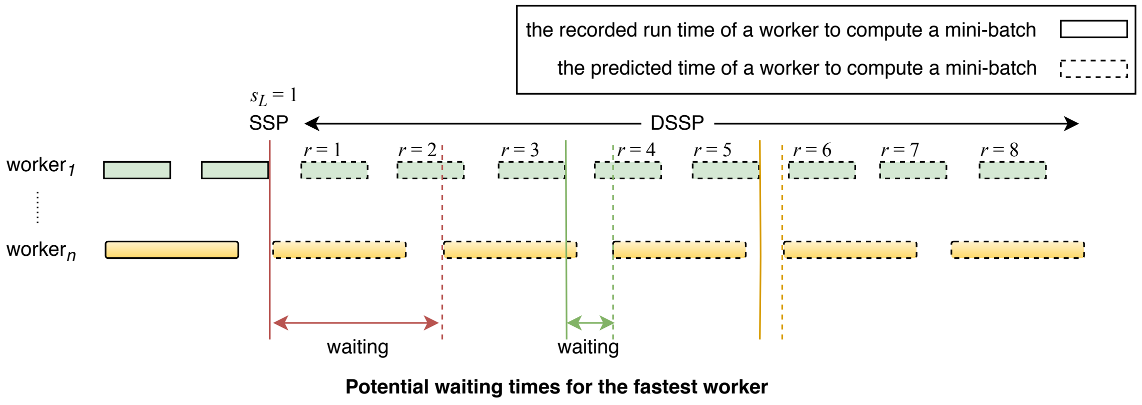

the recorded run time of a worker to compute a mini-batch
the predicted time of a worker to compute a mini-batch
$s_L = 1$
SSP
DSSP
worker$_1$
worker$_n$
$r = 1$
$r = 2$
$r = 3$
$r = 4$
$r = 5$
$r = 6$
$r = 7$
$r = 8$
waiting
waiting
Potential waiting times for the fastest worker

Figure 5.1: Prediction module finding the *least waiting time* for the *fastest worker* via iteration time intervals of workers. A solid line represents a boundary to stop the fastest workers continuing new iterations for *synchronization* and a dash line represents the end of waiting when the slowest worker completes its running iteration. The solid line is drawn upon a fastest worker sends a `push` request to the server and waits for the *`OK`* signal from the server. Once *`OK`* is received, it pulls the new updated weight from the server and starts a new iteration where the dash line is drawn. The dash line also indicates the time that the *slowest worker* receives a new updated weight via `pull` request and starts a new iteration. Worker$_1$ is the *fastest worker* and the worker$_n$ is the *slowest worker*. The colored block represents one iteration time. Following SSP, worker$_1$ has to stop at the red solid line. DynamicSSP compares each r value and finds the r^* which gives the least waiting time. Here, $r^* = 3$ if $r \in R = [0, 4]$. DynamicSSP allows worker$_1$ to run 3 more iterations and stop at the green solid line.

For example, Figure 5.1 shows that if the threshold is exceeded at the red solid line, the waiting time for the fastest worker if it starts waiting right away is more than the waiting time if it continues but starts waiting at the green solid line. In fact, the waiting time for it to start waiting at the yellow solid line is the minimum of the three. However, we have to make sure that the difference in iterations between the fastest and slowest workers is not too large. Otherwise, too many staled updates may delay the convergence of the ML model [16].

5.2 Contributions

To solve these problems, we propose an adaptive SSP scheme named Dynamic Stale Synchronous Parallel (DSSP). Our approach dynamically selects a threshold value from a given range in the training process based on the statistics of the real-time processing speed of distributed computing resources. It allows the threshold to change over time and also allows different workers to have different threshold values, adapting to the run-time environment. To achieve this purpose, we design a *synchronization controller* at the server side to determine how many iterations the current fastest worker should continue running at the end of its iteration upon the excess of the lower bound of a user-specified staleness threshold range. The decision is made at run time by estimating the future waiting times of workers based on the timestamps of their previous *push requests* and selecting a time point in the range that leads to the least estimated waiting time. In this way, we enable the parameter server to dynamically adjust or relax the synchronization of workers during the training based on the run-time environment of the distributed system.

In addition, although experiments have been reported on parameter servers with a variety of ML models, experiments of comparing DNN models under different distributed paradigms are rarely seen in the literature. In this Chapter, we look into four distributed paradigms (i.e., BSP, ASP, SSP and DSSP) and compare their performance by training three DNN models on two image datasets using MXNet [22] which provides BSP and ASP. We implemented both SSP and DSSP in MXNet, report and analyze our findings from the experiments.

5.3 Outline

In this chapter, we propose the *Dynamic Stale Synchronous Parallel* (**DSSP**) synchronization method. Instead of using a single and definite staleness threshold as in SSP, DSSP takes a range for the staleness threshold as input, and dynamically determines an optimal value for the staleness threshold during the run time. The value for the threshold can change over time and adapt to the run-time environment.

5.3.1 Problem statement

Given a *lower bound* and an *upper bound* of staleness thresholds s_L and s_U, DSSP finds an *optimal* threshold $s^* \in [s_L, s_U]$ for a worker dynamically, which yields the *minimum waiting time* for the worker to *synchronize* with others, based on the *iteration time* collected from each worker at the run time.

In other words, DSSP finds an integer $r^* \in R$, where $R = [0, r_{max}]$, $r_{max} = s_U - s_L$ and $r^* = s^* - s_L$. That is, DSSP finds an *optimal spot* on the time line R to bound the workers for synchronization. Empirically we can find a $r = s - s_L$ that is close to r^*, which is the same as finding $s \in [s_L, s_U]$ close to s^*.

5.3.2 Assumption

An *iteration interval* of a worker is the time period between two *consecutive* updates (i.e., push requests) the server receives from the worker. We can measure the length of an iteration interval of a worker by using the timestamps of the push requests sent by the worker (see Figure 5.2).

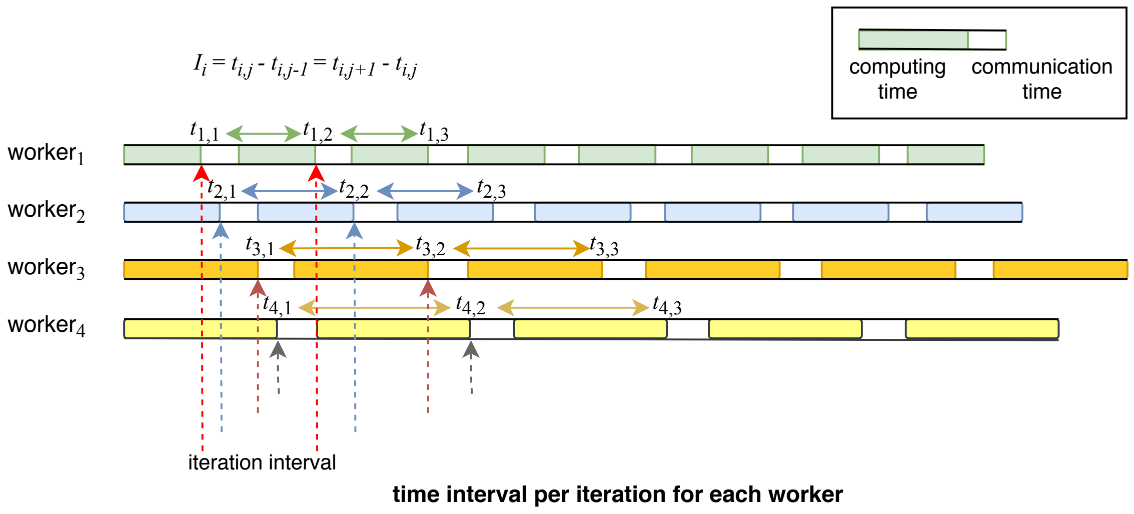


Figure 5.2: Iteration *interval*s measured by timestamps of push requests from workers. A dotted line represents the time for a push request from a worker to the server. An *interval* consists of *communication period* (blank block) and gradient *computation period* (solid block).

We assume that the iteration intervals of a worker in continuous iterations in a short time period are very similar. That is, for contiguous iterations of a worker in a short time period, each iteration has the similar processing time which includes computing gradients over a mini-batch, sending gradients to and receiving updated weights (parameters) from the parameter server.

5.3.3 Method

The proposed DSSP method is described in Algorithm 1. The algorithm contains two parts: one for workers and the other for the server. Each worker is assigned a partition of the training data, and computes parameter updates (i.e., gradients) iteratively with the partition using the stochastic gradient descent (SGD) method. In each iteration, a mini-batch of the partition is used to compute the gradients based on the current local weights. The worker then sends the gradients to the server through a *push request* and waits for the server to send back the `OK` signal. After receiving `OK`, the worker pulls the weights from the server and replaces its local weights with the global weights from the server. The training at the worker continues with the next mini-batch of the data partition based on the new weights.

On the server side, once the server receives a *push request* from a worker p, it updates its weights with the gradients from worker p. It then determines whether to allow worker p to continue. If yes, it will send worker p an `OK` signal; otherwise, it postpones sending the `OK` signal until the slowest worker catches up.

To determine whether to allow a worker p to continue, the server stores *the number* of *push requests* received from each worker and finds the *slowest worker*. If the number of push requests of worker p is no more than s_L iterations away from the slowest worker, the server allows worker p to continue by sending `OK` to worker

Algorithm 1 Dynamic Staled Gradient Method

```
Worker $p$ at iteration $t_p$
1: Wait until receiving OK from Server
2: pull weights $w^s$ from Sever
3: Replace local weights $w^{t_p}$ with $w^s$
4: Gradient $g^{t_p} \leftarrow \frac{1}{m}\sum_{i=1}^{m}\partial l_{loss}((x_i, y_i), w^{t_p})$
   ▷ $m$: size of mini-batch $M$ and $(x_i, y_i) \in M$
5: push $g^{t_p}$ to Server
Server at iteration $t_s$
 - Upon receiving push request with $g^{t_p}$ from worker $p$;
 - $r_p$ stores the number of extra iterations worker $p$ is allowed beyond $s_L$, initialized
   to zero at the very beginning;
 - $t_i$ stores the number of push requests received from worker $i$ so far
1: $t_p = t_p + 1$
2: Update the server weights $w^{t_s}$ with $g^{t_p}$. If some other workers send their updates
   at the same time, their gradients are aggregated before updating $w^{t_s}$
3: if $(r_p > 0)$ then
4:     $r_p = r_p - 1$
5:     Send OK to worker $p$
6: else
7:     Find the slowest and fastest workers based on array $t$
8:     if $(t_p - t_{slowest} \leq s_L)$ then
9:         Send OK to worker $p$
10:    else
11:        if $t_p$ is the fastest worker then
12:            $r_p \leftarrow$ synchronization_controller $(clock_p^{push}, r_p)$
               ▷ $clock_p^{push}$: timestamp of push request from worker $p$
13:            if $(r_p > 0)$ then
14:                Send OK to worker $p$
15:        Wait until the slowest worker sends the next push request(s) so that
           $t_p - t_{slowest} \leq s_L$. After updating the server weights $w^{t_s}$ with (aggregated)
           gradients, send OK to worker $p$
```

p (Lines 7-9 in Algorithm 1). Otherwise, if worker p is currently the *fastest worker*[1], the server calls the *synchronization_controller* procedure to determine whether it allows worker p to continue with extra iterations.

[1]The reason we call the procedure only for the current fastest worker is to save the server's computation time.

Algorithm 2 synchronization_controller

```
Input: $push_p^t$: timestamp of push request of worker $p$ for sending its iteration $t$'s
    update to the server
Output: $r^*$, number of extra iterations that worker $p$ is allowed to run
    {Table $\mathcal{A}$ stores the timestamps of two latest push requests by all workers, where
    $\mathcal{A}[i][0]$ stores the timestamp of the latest push request by worker $i$ and $\mathcal{A}[i][1]$
    stores the timestamp of the second latest push request by worker $i$}
 1: $\mathcal{A}[p][1] \leftarrow \mathcal{A}[p][0]$
 2: $\mathcal{A}[p][0] \leftarrow push_p^t$
 3: Find the *slowest worker* from table $\mathcal{A}$
 4: Compute the length of the latest iteration interval of worker $p$:
        $\mathcal{I}_p \leftarrow \mathcal{A}[p][0] - \mathcal{A}[p][1]$
 5: Compute the length of the latest iteration interval of the *slowest worker*:
        $\mathcal{I}_{slowest} \leftarrow \mathcal{A}[slowest][0] - \mathcal{A}[slowest][1]$
 6: Simulate next $r_{max}$ iterations for worker $p$ based on $\mathcal{I}_p$ and $\mathcal{A}[p][0]$ by storing
    the $r_{max}$ simulated timestamps in array $Sim_p$ so that:
 7: $Sim_p[0] \leftarrow \mathcal{A}[p][0]$
 8: $Sim_p[i] \leftarrow Sim_p[0] + i \times \mathcal{I}_p$ where $0 < i \leq r_{max}$
    $\triangleright$ $r_{max}$: the maximum extra iterations allowed
 9: Repeat the above step for the *slowest worker* and store the $r_{max}$ simulated times-
    tamps in array $Sim_{slowest}$ with $Sim_{slowest}[0] \leftarrow \mathcal{A}[slowest][0] + \mathcal{I}_{slowest}$
10: Find the simulated time point $r^*$ for the index of $Sim_p[r]$ that minimizes
    $|Sim_{slowest}[k] - Sim_p[r]|$ for all $k \in [0, r_{max}]$ and $r \in [0, r_{max}]$
11: return $r^*$
```

Algorithm 2 describes the *synchronization_controller* procedure. It stores in table $\mathcal{A}$ the *timestamps* of the *two* latest *push requests* from all workers, and uses the information in $\mathcal{A}$ to simulate the next r_{max} iterations of worker p and the slowest worker, where r_{max} is the maximum number of extra iterations allowed for a worker to be ahead of the slowest worker beyond the lower bound of the staleness threshold. With the simulated timestamps, it finds a time point r^* in the range of $[0, r_{max}]$ that minimizes the simulated waiting time of worker p (Line 8 in Algorithm 2). The value r^* is returned to the caller (the Server part of Algorithm 1) and stored as r_p for worker p. For example, in Figure 5.1, suppose worker n is the slowest worker and we are currently processing worker 1 (i.e., $p = 1$). The green boundary yields the least waiting time for worker 1. Then worker 1 should continue running 3 more iterations once s_L is exceeded. In this case, 3 is the r^* returned to the server procedure of Algorithm 1. If 0 is returned, it indicates that the current iteration yields the least waiting time, and the worker should wait for the slowest worker at the current iteration.

In future iterations when worker p sends a push request, if $r_p > 0$, the server sends the `OK` signal right after updating the global weights with the gradients sent by the worker and decreases r_p by 1. In this way, even if worker p is not the fastest worker in that iteration of the server, as long as its $r_p > 0$ (due to being the fastest worker in a previous iteration), it can still perform extra iterations beyond s_L. Thus, our method is flexible in that different workers may have different thresholds, and also the threshold for a worker can change over time, depending on the run-time environment.

5.4 Theoretical Analysis

In this section, we prove the convergence of SGD under DSSP by showing that DSSP shares the same *regret bound* $O(\sqrt{T})$ as SSP from [7]. That is, SGD converges in expectation when the number of iterations T is large under DSSP. We first present the theorem of SSP. Based on the theorem, we show that DSSP has a bound on regret. Therefore, DSSP supports SGD convergence following the same conditions as SSP.

Theorem 1 (adapted from [7]. **SGD under SSP).**

Suppose function $f(w) := \sum_{t=1}^{T} f_t(w)$ is a convex function and $\forall f_t(w)$ is also convex. We use iterative convergent optimization algorithm (gradient descent) on one component ∇f_t at a time to search for the minimizer w^* under SSP with the staleness threshold s and P workers. Let $v_t := -\eta_t \nabla f_t(\widetilde{w}_t)$ where $\eta_t = \frac{\sigma}{\sqrt{t}}$ and $\sigma = \frac{F}{L\sqrt{2(s+1)P}}$. Here $\widetilde{w}_t$ represents the noisy state of the globally shared weight. F and L are constants. Assume f_t are L-Lipschitz with constant L and the distance $D(w\|w')$ between two multidimensional points w and w' is bounded such that

$D(w\|w') := \frac{1}{2}\|w - w'\|_2^2 \leq F^2$ where F is constant. We have a bound on the regret

$$R[X] := \sum_{t=1}^{T} f_t(\widetilde{w}_t) - f(w^*) \leq 4FL\sqrt{2(s+1)PT} \tag{5.1}$$

Thus, $R[X] = O(\sqrt{T})$ since $\lim_{T\to\infty} \frac{R[X]}{T} = 0$

Theorem 2 (SGD under DSSP). Following all conditions and assumptions from Theorem 1, we add a new term $R = [0, s_U - s_L]$, the *range* of the staleness threshold. Let $r \in R$ and $r \geq \forall r' \in R$. We have a bound on the regret

$$R[X] := \sum_{t=1}^{T} f_t(\widetilde{w}_t) - f(w^*) \leq 4FL\sqrt{2(s_L + r + 1)PT} \tag{5.2}$$

Thus, $R[X] = O(\sqrt{T})$ since $\lim_{T\to\infty} \frac{R[X]}{T} = 0$

Proof. Since we follow all conditions and assumptions from Theorem 1, we need to show the newly added range R does not change the regret bound of SSP. In DSSP, the threshold is dynamically changing between s_L and s_U where $r \in R$ and $R = [0, s_U - s_L]$. We know that SSP with threshold s_L has a bound on regret according to (5.1). We only extend the threshold s_L of Theorem 1 to $s_L + r$ where r is the largest number from R. Suppose we set a fixed threshold s' for SSP, then our DSSP can be deducted to SSP when we set $s' = s_L + r$. Thus, we have a upper bound on regret of SSP with threshold s'. $\square$

5.5 Experiment

We evaluate the performance of DSSP compared to other three distributed paradigms. We aim to find whether DSSP converges faster than SSP on average and whether it can maintain the predictive accuracy of SSP.

5.5.1 Experiment setup

Hardware

We conducted experiments on the SOSCIP GPU cluster [57] with up to four IBM POWER8 servers running Ubuntu 16.04. Each server has four NVIDIA P100 GPUs. Each server has 512 GB ram and 2×10 cores. The servers are connected with Infiniband EDR. Each server connects directly to a switch with dedicated 100 Gbps bandwidth.

We also set up a virtual cluster with a mixed GPU models by creating two Docker containers running Ubuntu 16.04 on a server with NVIDIA GTX1060 and GTX1080 Ti. The server has 64 GB ram and 8 cores. Each container is assigned with a dedicated GPU.

Dataset

We used CIFAR-10 and CIFAR-100 datasets [58] for image classification tasks. Both datatsets have 50,000 training images and 10,000 test images. CIFAR-10 has 10 classes, while CIFAR-100 has 100 classes.

Models

We used a downsized AlexNet [13], ResNet-50 and ResNet-110 [31] as our deep neural network structure to evaluate the four distributed paradigms. We reduced the original AlexNet structure to a network with 3 convolutional layers and 2 fully connected layers to achieve faster convergence within 24 hours (which is the time limit we are allowed to run for each job on the SOSCIP cluster). We set the staleness threshold $s_L = 3$ and the range $R = [0, 12]$ for DSSP which is equivalent to the corresponding threshold range $[3, 15]$ for SSP.

We ran each paradigm on 4 servers. Each server represents a worker which has 4 GPUs. Each GPU loads a copy of the DNN model. Thus, there are 16 replica models for 4 workers. Each worker collects the computed gradients from 4 GPUs by the end of every iteration and sends the sum of the gradients to the parameter server. One of the 4 servers is also elected to run the parameter server when the training starts from the very beginning as defined in MXNet. We ran each experiment three times and chose the medium result based on the test accuracy.

5.5.2 Results and Discussion

We used batch size 128, learning rate 0.001 in 300 epochs to train the downsized AlexNet on CIFAR-10. Figure 5.3a shows that DSSP, SSP and ASP converge much faster than BSP, and that DSSP and SSP converge to a higher accuracy than ASP. BSP is the slowest to complete the 300 epochs. The performance of DSSP and averaged SSP are similar, with DSSP converging a little bit faster to a bit higher accuracy. DSSP and averaged SSP complete 300 epochs almost at the same time. Note that this result is expected because the result of averaged SSP is the average over the results from 13 different threshold values from 3 to 15, and when its threshold is large, it is very fast, much faster than DSSP with a threshold range of [3,15]. However, a larger threshold of SSP incurs more staler gradients, which implies more noises and decreases the quality of iterations [7]. Theoretically, as the threshold s of SSP increases, the rate of convergence decreases per iteration update [7]. Figure 5.3b compares DSSP with individual SSPs with different threshold values. It shows that DSSP converges a bit faster to a higher accuracy than almost all of the SSPs except for one.

For ResNet-50 and ResNet-110 training on CIFAR-100, we used batch size 128, learning rate 0.05 and decay 0.1 twice at epoch 200 and 250 in 300 epochs for both. In Figure 5.3c, DSSP has the same convergence rate as ASP and SSP, and they converges much faster than BSP although BSP completes 300 epochs faster than others on both ResNets. Again, DSSP converges a little faster and archives a bit higher accuracy than averaged SSP in Figure 5.3e. Four distributed paradigms on both ResNets behave in an opposite way compared to the downsized AlexNet which has fully connected layers in terms of the time taken to complete 300 epochs. The order from fastest to slowest is BSP, SSP, DSSP and ASP.

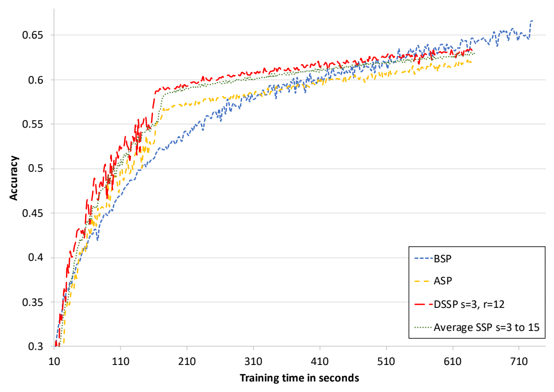
Based on the empirical results, we observe two opposite trends of ASP, DSSP, SSP and BSP with respect to their performance. The trends can be classified by the architecture of DNNs: ones that contain fully connected layers and ones that do not. Note that we do not count the final fully connected softmax layer as the fully connected layer over the discussion.

DNNs with fully connected layers (AlexNet)

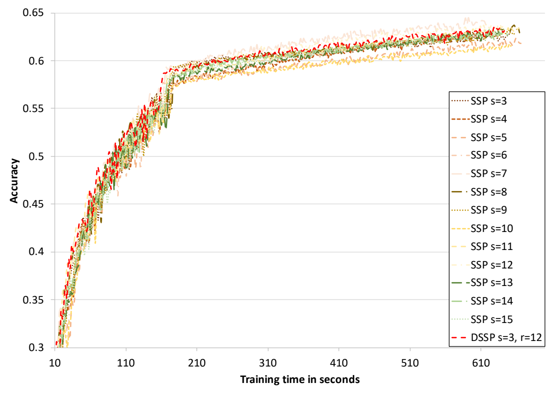
DSSP converges to a higher test accuracy faster than ASP, BSP and average SSPs in its corresponding staleness threshold range. ASP has the largest iteration throughput and its convergence rate is close to DSSP but it usually converges to a very low accuracy (the lowest of four paradigms) and diverges sometimes (see Figure 5.3a). DSSP performs between SSP and BSP in terms of final test accuracy. In this category, our DSSP converges faster than SSP and ASP to a higher accuracy. We know BSP guarantees the convergence and its accuracy is the same as using a single machine. Thus, it has no consistency errors caused by delayed updates. Given abundant time of training, BSP can reach the highest accuracy among all distributed paradigms. We do not discuss BSP in detail here since our focus is to show the benefits that our DSSP brings compared to SSP and ASP, both cost less training time than BSP.

DNNs without fully connected layers (ResNet-50, ResNet-110)

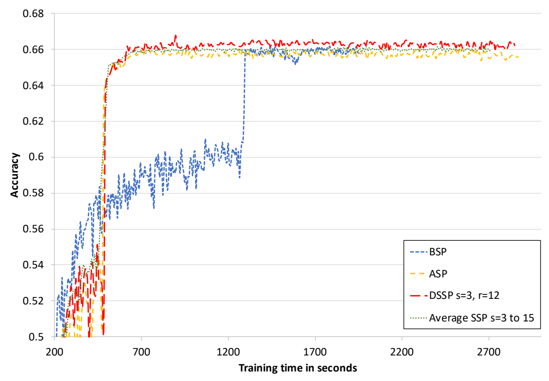
DSSP converges faster than average SSP in its corresponding range on very deep neural networks. ASP appears to be a strong rival but it has no guarantee to converge as addressed in Section 3.1.2. BSP delivers the highest iteration throughput. However, it converges slower and to a lower accuracy than other three paradigms mostly (see Figure 5.3c, 5.3e). The iteration throughputs of ASP, DSSP, SSP and BSP are in ascending order. In this category, the convergence rate of DSSP, ASP and SSP are very close. DSSP performs slightly above the average SSPs where the threshold s starts from 3 to 15 (see Figure 5.3d, 5.3f).



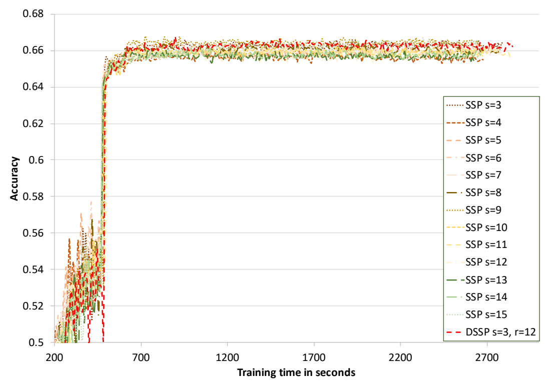
(a) All paradigms run on downsized AlexNet



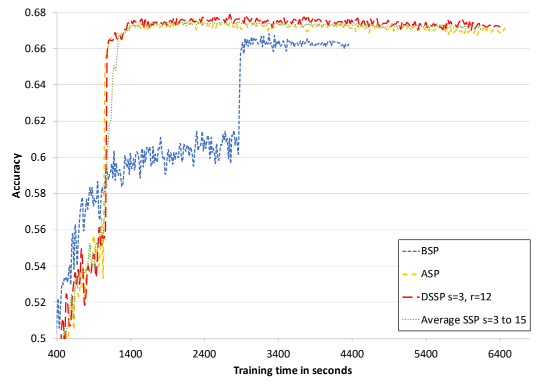
(b) DSSP and SSPs run on downsized AlexNet



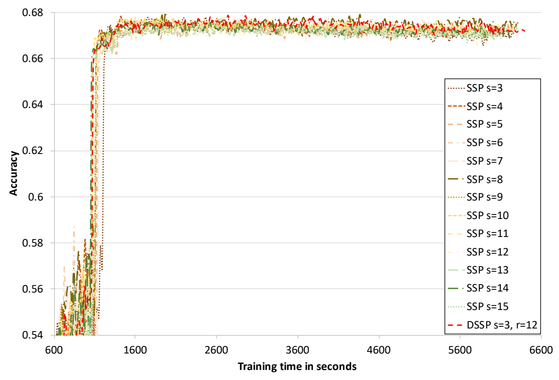
(c) All paradigms run on ResNet-50



(d) DSSP and SSPs run on ResNet-50



(e) All paradigms run on ResNet-110



(f) DSSP and SSPs run on ResNet-110

Figure 5.3: Distributed paradigms comparison on downsized AlexNet, ResNet-50 and ResNet-110 training for 300 epochs. Downsized AlexNet is trained on CIFAR-10 and both ResNets are trained on CIFAR-100. Average SSP on the right column is derived by averaging SSPs with threshold from 3 to 15 on the left column. Faster convergence to a targeted high accuracy indicates less training time is required for the paradigm.

5.5.3 Demystify the Difference

Below we answer two questions:

1. Why does the iteration throughput have the opposite trends for ASP, DSSP, SSP and BSP on DNNs with and without fully connected layers?

2. Why do pure convolutional neural networks (CNNs) receive a higher accuracy from DSSP, SSP and ASP than BSP?

We temporarily name DNNs with fully connected layers as DNNs and pure CNNs as CNNs for the convenience of discussion.

To answer the first question, we observe the difference between the two types of DNNs (with or without fully connected layers): ① A fully connected layer requires more parameters than a convolutional layer which uses shared parameters [59]. DNNs with fully connected layers have a large number of model parameters that need to be transmitted between workers and the server for updates. ② Convolutional layers require intensive computing time for matrix dot product operations while computing for fully connected layers involves simple linear algebra operations [60]. CNNs that only use convolutional layers take a lot of computing time, while their relatively smaller-size model parameters cost less data transmission time between workers and the server than DNNs. Moreover, when the ratio of computing time and communication time per iteration is small, less time can be saved per iteration for workers since computing time per iteration (or one mini-batch) is fixed for a model per worker. To the contrary, when the ratio is large, the communication time per iteration for each worker can be shifted by asynchronous-like parallel schemes and more time can be saved. Therefore, DSSP, SSP and ASP take less training time on DNNs whereas BSP costs the least training time on CNNs.

To the second question, the answer lies in the difference between fully connected layers and convolutional layers. Fully connected layers are easy to overfit the data set due to its large number of parameters [61]. Thus, any error introduced by staled updates can cause many parameters diverge in non-uniform convergence [11]. Informally, fully connected layers overfit to the errors injected by delayed updates or noise. Convolutional layers have less parameters due to the use of filters (shared parameters). For image classification tasks, a commonly used trick to train CNNs on a small data set is to increase the data by distorting the existing images and saving them [62] since CNNs are able to tolerate certain scale variations [63]. Distortion can be done by rotating the image, setting one or two of RGB pixels to zero or adding Gaussian noise to the image [64]. It is basically to introduce noise to images so that CNN models receive enhanced training and the predictions are improved. The errors caused by (not too) staled updates can give the same effect to the training model as the distortion. Figures 5.3c, 5.3e are good evidence to support that. Furthermore, [65] empirically shows that adding gradient noises improves the accuracy for training very deep neural networks which also happened in our ResNet-110 experiments (in Figure 5.3e).

5.5.4 Cluster with mixed GPU models

The results of DSSP and SSPs on ResNet-110 (see Figure5.3e) do not show a significant difference on convergence rate on a homogeneous environment where GPUs are identical. Nonetheless, on the heterogeneous environment where we have one GTX1060 and one GTX1080 Ti running on each worker, DSSP converges faster and to a higher accuracy than SSP. We repeated the exact same experiments on ResNet-110 as earlier: use the same hyperparameters setting, run 3 trials on each paradigm and choose the medium one based on the test accuracy. Figure 5.4 and Table 5.1 show that DSSP can reach a higher accuracy significantly faster than SSP.

The heterogeneous environment is very common in industry since new GPU models come to the market every year while the old models are still in use. ASP can fully utilize the individual GPU and achieves the largest iteration throughout. However, ASP also introduces the most staled updates among all distributed paradigms. It may converge to a lower accuracy than DSSP when the GPU models' processing capacities are significantly different since the iterations between the fastest worker and the slowest worker are dramatically different. In contrast, DSSP has consistent performance regardless the running environment since it adapts to the environment by adjusting the threshold dynamically for every iteration of workers.

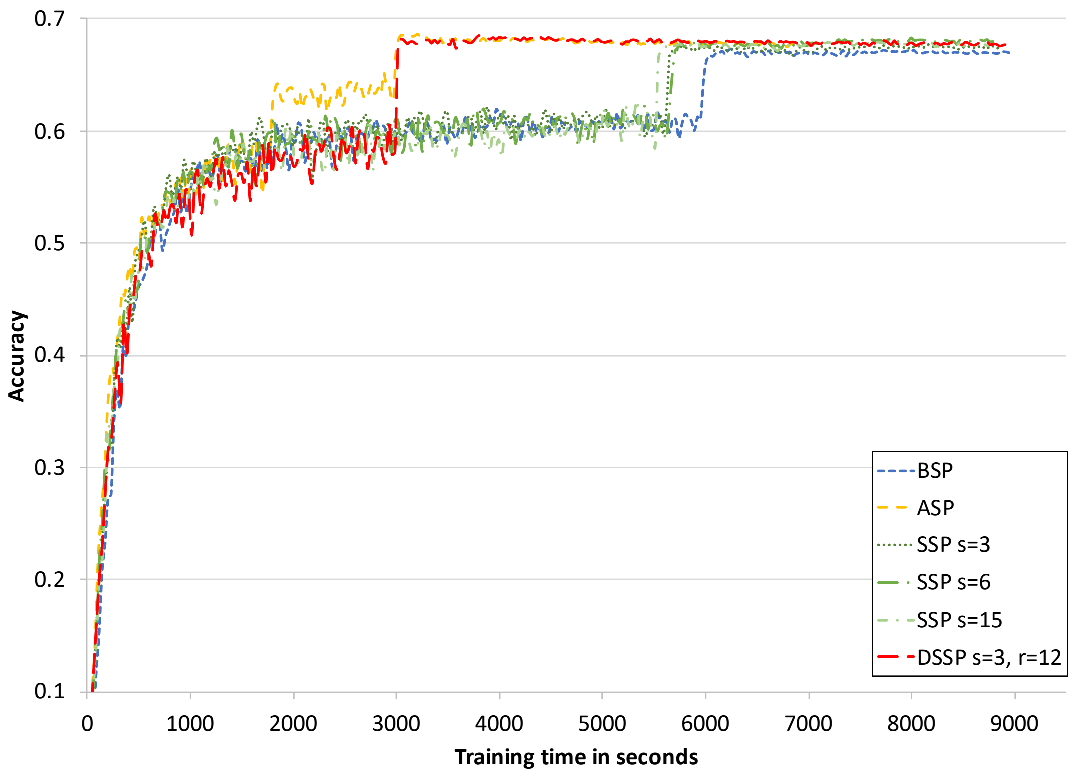


Figure 5.4: Trained ResNet-110 on CIFAR-100 with two workers on a mixed GPU cluster for 300 epochs. GTX1060 and GTX1080 Ti are assigned to individual worker. Our DSSP converges faster and achieves higher accuracy than SSP.

5.6 Related Work

There are variety of approaches to optimizing the distributed paradigms under the parameter server framework. Generally, they can be categorized into three basis streams: BSP, ASP and SSP. Chen et al. [1] try to optimize the BSP by adding few backup workers. That is to train N workers in BSP, they add c backup workers so that there are $N + c$ workers during the training. By the end of each iteration, the server only takes the first N arrived updates and drops the c slower arrived updates from the stranglers for weight synchronization. In this case, the training

Distributed Paradigm	Time to reach 0.67 accuracy	Time to reach 0.68 accuracy
BSP	6159.2	–
ASP	2993.1	3017.2
SSP s=3	5678.2	–
SSP s=6	5703.8	6908.2
SSP s=15	5564.9	7255.6
DSSP s_L=3, r=12	3016.4	3046.3

Table 5.1: Time in seconds to reach the targeted test accuracy in training. The maximum test accuracy of BSP and SSP with s=3 is 0.67. Trained ResNet-110 on CIFAR-100 with two workers for 300 epochs. Each worker has either GTX1080 Ti or GTX1060.

data allocated to the c random slower workers are partially wasted in each iteration.

Hadjis et al. [66] optimize ASP from machine learning's perspective. It adjusts the momentum based on the degree of asynchrony (staleness of the gradients). Then, it uses the tuned momentum to mitigate the divergent direction that staled gradients introduce. Meanwhile, model parallel computing is applied here for better performance where a DNN model is split into two parts: convolutional layers and fully connected layers. Both parts are computed parallelly and concurrently.

Zhang and Kwok [67] propose to use asynchronous distribution to optimize synchronous ADMM algorithm. However, it uses *partial barrier* to make the fastest workers wait for the slowest workers and *bounded delay* to guarantee that the iterations among workers do not exceed a user specified hyperparameter τ which is equivalent to the staleness threshold s of SSP in [7]. Then, all these make the approach rather close to SSP than ASP optimization. Bounded delay also appears in [16] and is elaborated in the rest of this section.

Li et al. [16] introduce *bounded delay* which is similar to SSP except that it takes all workers' iterations into account instead of letting each worker count its own iteration. In order to keep the ML model (global weights) consistent, iterations in sequence are allowed to run concurrently in parallel under the dependency restriction. Iteration t depends on iteration $t-k$ if iteration t requires the result of iteration $t-k$ in order to proceed. In the bounded delay approach, the number of bounded iterations k is specified by the user, similar to the staleness threshold s in SSP. k means that for a continuous k iterations, every iteration is independent of each other, and they can run concurrently in parallel without waiting for each other. When k is exceeded, the fastest iteration t has to wait for the slower iterations behind $t-k$. Imagine iterations are pre-assigned to workers as tasks, then the bounded delay is equivalent to SSP. For example, we have iterations $\{I_1, I_2, I_3, I_4, I_5, I_6\}$ and two workers P_1 and P_2. Each iteration I_i completes a min-batch of samples. P_1 receives $\{I_1, I_3, I_5\}$, P_2 receives $\{I_2, I_4, I_6\}$. If $k = 3$, then I_4 depends on I_1, I_5 depends on I_2, I_6 depends on I_3. So P_2 at I_4 has to wait for P_1 finishes I_1. P_1 at I_5 has to wait for P_2 completes I_2. Bounded delay is rather an inflexible scheme by pre-scheduling tasks to workers. Although the authors briefly claim that more consistent paradigms can be developed based on the dependency constraint, no further exploration is provided in their paper. Our work extends this direction and presents a flexible scheduling approach in which k is dynamically assigned at the training time. In our dynamic bounded delay paradigm, every optimal bound k yields the least waiting time for workers by optimizing the synchronization frequency. For the continuous k iterations in which every iteration is independent is adjusted dynamically in the running time to reduce waiting time of coming iterations which depend on the earlier ones.

5.7 Epilogue

In this chapter, we introduce Dynamic Staleness Synchronous Parallel (DSSP). Our approach improves SSP in the sense that with DSSP a user does not need to provide a specific staleness threshold which is hard to determine in practice, and also that DSSP can dynamically determine the value for the threshold from a range using a lightweight method according to the run-time environment. This does not only alleviate the burden of an exact manual staleness threshold selection or multiple trials of hyperparameter selection, but it also provides flexibility of selecting different thresholds for different workers at different times. We provided theoretical analysis on the expected convergence of DSSP which inherits the same regret bound of SSP to show that DSSP converges in theory as long as the range is constant. We evaluated DSSP by training three DNNs on two datasets and compared its results with other distributed paradigms. For DNNs without fully connected layers, DSSP achieves higher accuracy than BSP and slightly better accuracy than averaged SSP. For DNNs with fully connected layers, DSSP generally converges faster than BSP, ASP and averaged SSP to a higher accuracy even though BSP can eventually reach the highest accuracy if it is given more training time. Unlike ASP, DSSP ensures the convergence of DNNs by limiting the staled delays. DSSP gives significant improvement than SSP and BSP in a heterogeneous environment with mixed models of GPUs, converging much faster to a higher accuracy. DSSP also shows more stable performance on either homogeneous or heterogeneous environment compared to other three distributed paradigms. Furthermore, we discussed the difference in the trends of four distributed paradigms on DNNs with and without fully connected layers and the potential causes.

The most interesting observation from the comparison between DSSP and SSP is that we found DSSP restricting the staleness of the gradients in a range can boost the accuracy in training empirically. Note that the distribution of the staleness of the gradients exists over the entire training on both DSSP and SSP. Assuming the distribution is Gaussian distribution, it is moving along the timeline of the training on either DSSP or SSP. This triggers our curiosity and we wonder what would happen to the accuracy if we cut off the distribution at some time point or clear all the staled gradients by applying a weight synchronization on all workers (i.e., bulk synchronization) similar to BSP. In this way, regardless what type of distribution of the staleness is, it will be segmented by a synchronization operation which clears all the staled gradients. Hence, we introduce Elastic Bulk Synchronous Parallel model that does these in the next chapter which will fulfill our curiosity.

Chapter 6

Elastic Bulk Synchronous Parallel Model

6.1 Introduction

The BSP model guarantees the convergence on training the DNN model since it is logically functioning as a single server. However, it introduces a large *waiting time overhead* due to having to wait for the slowest worker in every single iteration (a mini-batch). On the other hand, the ASP model does not perform any synchronization, so waiting time for synchronization is minimal, however, it is risky to be used due to its asynchronous scheme that renders the convergence uncertain [10]. The SSP model offers an intermediate solution to the above two extremes. It guarantees the convergence [7] when the number of iterations is large and the user specified threshold s is small. However, it depends on manually fine-tuning the s hyper-parameter which is non-trivial and is blind to the computational capacity of workers.

Motivated by the limitations of the current state-of-the-art synchronization models, we introduce Elastic Bulk Synchronous Parallel model (ElasticBSP) in this chapter. ElasticBSP aims to relax the strict synchronization requirement of BSP (as shown in Figure 6.1). The **key properties** of ElasticBSP are the following:

- The server deals with sequential decision making regarding the *best* time that the next synchronization barrier should be imposed (a time when the minimum waiting time for the entire system is achieved). The decision is based on a prediction model that utilizes information about the most recent time interval of each worker available to the server to predict its future intervals. The prediction is based on an online optimization with lookahead and assumes a specific limit R on how many future intervals for each worker should be considered. The need for a specific limit comes from the need to control the algorithm's run time, since that can increase exponentially as the lookahead limit R increases.

- The model guarantees the convergence when the number of iterations is large. The convergence guarantee follows the theoretical analysis of SSP [7], where a small iteration difference s exists in some period τ (a superstep). In the case of ElasticBSP, the iteration difference is bounded by the lookahead limit R in some period τ that is defined by the next optimal time. By the end of the period τ, the synchronization barrier is posed to all the workers where gradients aggregation is carried out on the server, similarly to BSP. The model weights are synchronized and will be available to all workers in the next iteration.

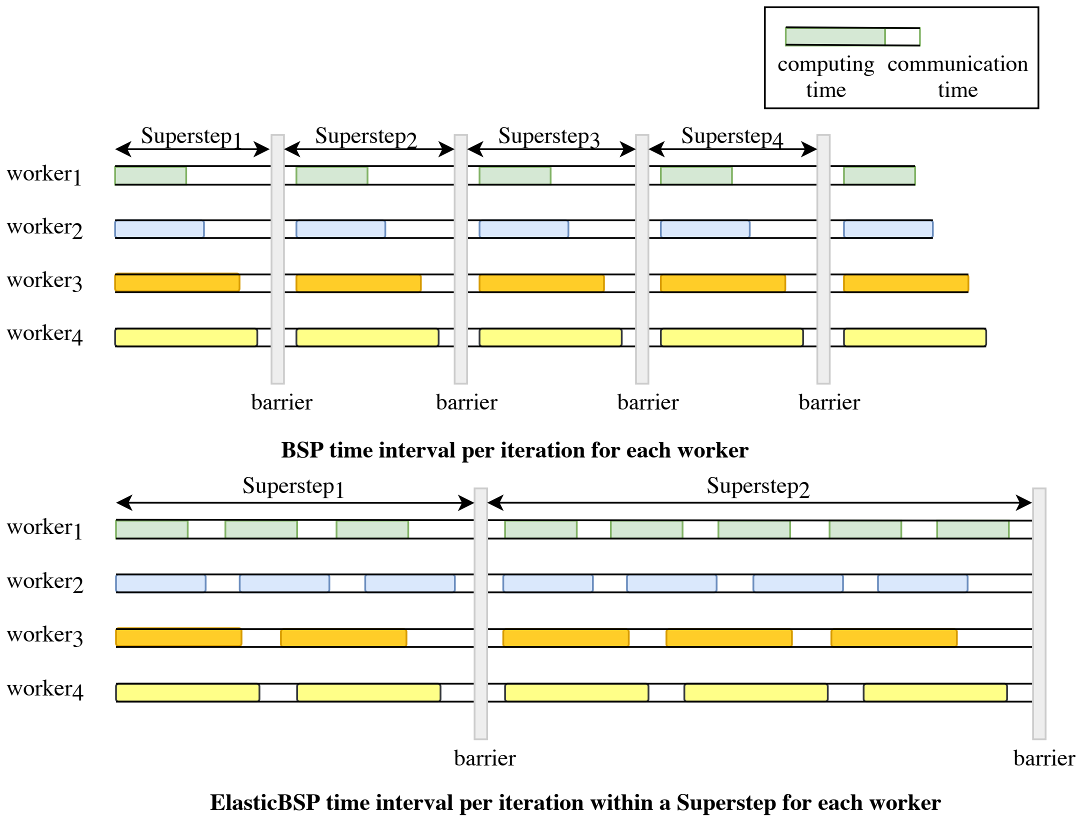


Figure 6.1: Vanilla BSP and our proposed ElasticBSP. Each *barrier* represents the time of weight synchronization among workers and a *superstep* represents the time between barriers. In BSP the superstep is fixed to a number of k iterations and all workers have to wait for each other at the end of their k iterations ($k = 1$ is shown, which is typical). In ElasticBSP, the time the barrier is imposed varies and each superstep can allow a different number of iterations per worker. These values are determined at runtime by our proposed ZipLine method that achieves minimum overall waiting time of all workers.

The decisions of ElasticBSP are the solutions to *an online optimization with lookahead problem* [68]. As such, the time a synchronization barrier is imposed *varies* and each superstep can permit a *different number* of iterations per worker, offering *elasticity* (see Figure 3.1). The waiting time is not determined by a fixed iteration difference between the fastest and the slowest workers (as in SSP), but based on the optimal time to synchronize in order to minimize the waiting time. In addition, the synchronization time is always bound within the lookahead limit R, so it will not behave as the ASP model.

From system engineering perspective, we will let the running environment determine the period τ to achieve the minimum waiting time for the entire system. Thus, ElasticBSP is adaptable to its running environment. Unlike SSP, we remove the hyperparameter s for user and instead use our one-pass algorithm to decide s dynamically at run time. We propose an efficient method that fast solves the optimization problem and materializes the ElasticBSP model named ZipLine. ZipLine consists of two phases. First, R future iteration intervals of each worker are predicted at run time based on their most recent intervals, assuming a stable environment (*the lookahead*). Then, a one-pass algorithm operates over the predicted intervals of all workers and determines the next optimal synchronization time (i.e., a time that minimizes the overall workers' waiting time overhead). The algorithm can effectively balance the trade-off between accuracy and convergence speed to accommodate different environments or applications. ZipLine finds the optimal τ^* which gives the least waiting time of all the workers. Consequently, τ^* determines s and can be considered as the upper bound of s. Thus, the convergence guarantee of

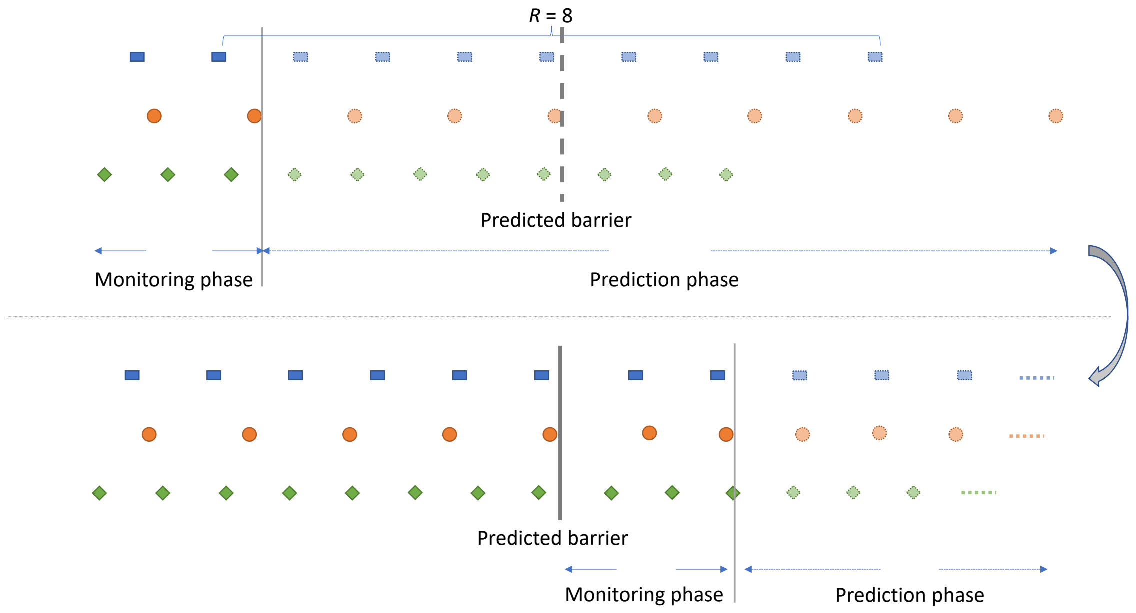


Figure 6.2: The flow of prediction and synchronization of ElasticBSP.

ElasticBSP is provided by SSP [7] when the number of iterations is large and τ is small. Notably, ElasticBSP, materialized by the proposed optimized version of ZipLine, converges faster and achieves higher accuracy than BSP, SSP and ASP for large-sized DNNs.

The flow of prediction and synchronization of ElasticBSP (as shown in Figure 6.2) consists of two phases: (i) the monitoring phase, and (ii) the prediction phase. The purpose of the monitoring phase is to allow the server to learn the iteration interval of all the workers. It requires at least two `push` requests per worker. The prediction phase consists of predicting the R future iterations of each worker and using ZipLine to get the optimal time to impose the next synchronization barrier. Once a synchronization occurs, the two phases repeat again. This example in Figure 6.2 illustrates that even of one of the workers (round orange) has different speeds between synchronizations (it gains faster speed in the lower figure), ElasticBSP can still find the optimal next synchronization time, since it always uses the most recent history of workers.

It is important to note that ElasticBSP focuses on optimization of the distributed training of DNN models assuming the parameter server framework and can be applied to other distributed machine learning methods based on gradient descent optimization. As such, it is orthogonal to other DNN optimization techniques, such as learning rate optimization [69].

The major contributions of this work are as follows:

- We propose ElasticBSP, a novel synchronization model for scaling the training of distributed deep learning models. ElasticBSP replaces the strict synchronization requirement of other BSP-like models with an online sequential decision making about the best time to impose the next synchronization barrier. The model guarantees convergence when the number of iterations of the training phase is large.

- We propose ZipLine, a one-pass algorithm that can efficiently materialize the ElasticBSP model. ZipLine performs online optimization with lookahead to decide the next best synchronization time. It outperforms sensible baselines that exhibit polynomial time complexity.

- We propose two optimizations of ZipLine, namely ZipLineOpt and ZipLineOptBS. The former is an algorithmic optimization that relies on a pruning technique to reduce the search space of the solution. The latter is an implementation optimization that employs an advanced data structure offering fast search operations and manages to achieve *linearithmic* time complexity.

- We present a thorough experimental evaluation of our ElasticBSP model materialized by the ZipLine on four deep learning models (vary in size and structure) on three popular image classification datasets (vary in class size, sample size and image resolution). The results show that ElasticBSP converges much faster than BSP and to a higher accuracy than BSP and other state-of-the-art alternatives. In particular, ElasticBSP demonstrates a superior performance on the large-sized DNNs with respect to convergence speed and accuracy.

The remainder of the chapter is organized as follows. Section 6.2 provides a brief background of the state-of-the-art synchronization models and their limitations under the parameter server framework. Section 6.3 introduces our proposed ElasticBSP synchronization model and its properties. Section 6.4 formally defines the problem of interest and provides problem analysis. In Section 6.5, we first present algorithmic details of sensible baselines; then we present our proposed method ZipLine, and its two optimized variants ZipLineOpt and ZipLineOptBS that can materialize ElasticBSP. Section 6.6 presents a thorough experimental evaluation of the methods. We review the related work in Section 6.7 and we conclude in

Section 6.8.

6.2 Background

In this section, we first present a *synchronization cost model* of training a deep learning model, assuming the *parameter server framework*; in particular, cost related to the wall-clock idle time of processors. Then, we briefly present state-of-the-art synchronization models employed by the server for achieving *parallel SGD* computation through *synchronous* (or *asynchronous*) data communication between the server and workers, and discuss their main advantages and limitations.

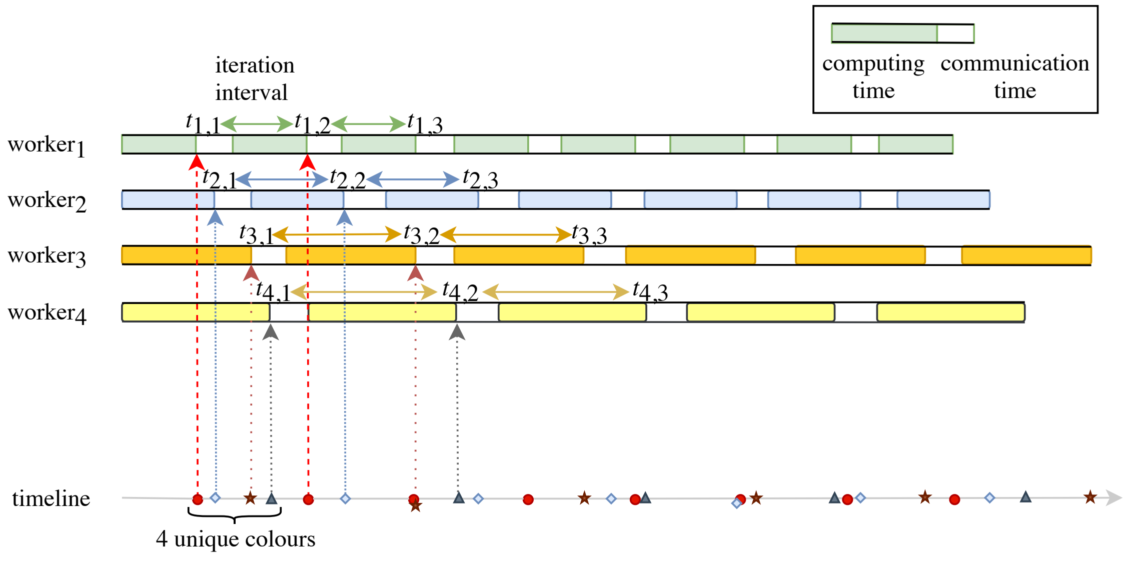
6.2.1 Synchronization Cost Model

In the *parameter server framework*, the *server* becomes aware of a worker's *iteration intervals* through the *timestamps* of the worker's `push` requests. Formally, given a worker $p \in [1, n]$, a single *iteration interval* T^p_{iter} consists of *computing time* T^p_{comp} and *communication time* T^p_{comm} (see examples in Figure 3.1 and 6.3). In addition, we know that the communication time can be broken down into *data transmission time* T^p_{trans} and *waiting time* T^p_{wait} until the synchronization occurs. During data transmission, a worker sends gradients to the server and the server sends weights back to the worker. We ignore lower level communication (e.g., TCP/IP) since the size of the exchanged data is significantly smaller than that of gradients and weights.

The time cost of a single iteration of worker p is:

$$\gamma^p_{iter} = \gamma^p_{comp} + \gamma^p_{trans} + \gamma^p_{wait} \tag{6.1}$$

where γ^p_{iter}, γ^p_{comp}, γ^p_{trans} and γ^p_{wait} represent the associated length of a period T^p_{iter}, T^p_{comp}, T^p_{trans} and T^p_{wait} respectively. Note that γ^p_{comp} is a constant since the hardware computational capacity is fixed and the batch size does not change — each iteration computes a single batch. Also, γ^p_{trans} is a constant since workers are training the same model and both the weights and the gradients are of the same data size. On the other hand, γ^p_{wait} is a variable that can be controlled by the *parameter server*. Recall that each worker has to wait for the signal from the server to `pull` the weights after a synchronization (aggregation) operation in BSP. Let us denote as t_p the time point that the worker p started waiting and t_{sync} the time point that the synchronization completes (i.e., imposed barrier ends). Then $T^p_{wait} = [t_p, t_{sync}]$ and its length $\gamma^p_{wait} = t_{sync} - t_p$. Therefore, from the optimization perspective, for n workers in a distributed system, the cost of applying one synchronization barrier b at time t_{sync} is dominated by the longest waiting time of any of the n workers, say $\tau^b_{wait} = \max(\gamma^p_{wait}), p \in [1, n]$ for a superstep τ ending with a barrier b. Assuming that there is a set B of synchronization barriers defined by $|B|$ synchronization timestamps during the



time interval per iteration for each worker

Figure 6.3: Iteration intervals measured by timestamps of push requests from workers. A dotted line represents the time a push request arrives at the server from a worker. An iteration interval consists of gradient computing period (solid block) and communication period (blank block). All workers' ending timestamps can be mapped onto a timeline. Each timestamp on the timeline is associated to one of the workers. A set which is represented by the bracket always keep n unique values (colors) of workers. ZIPLINE scans the points from left to right on the timeline, takes one color point into the set per iteration.

training period, the synchronization cost function of n workers is defined as:

$$c_{sync} = c(n, B) = \sum_{b=1}^{|B|} \tau_{wait}^{b} \tag{6.2}$$

We aim to find the optimal set B:

$$B^{*} = \operatorname*{argmin}_{B} c_{sync} \tag{6.3}$$

6.3 ElasticBSP Model

Motivated by the limitations of BSP, SSP and ASP synchronization models in the parameter server setting as we described in Section 3.1 and the observations on the performance of DSSP, we propose Elastic Bulk Synchronous Parallel (ElasticBSP), a novel synchronization model that has the premise to ameliorate drawbacks of current models, without sacrificing their benefits.

ElasticBSP offers *elasticity* in the sense that the distance between two consequent synchronization barriers (or any period τ) is not fixed (as in BSP), but it is determined online (at runtime). In addition, the waiting time is not determined by *a fixed iteration difference* between the fastest and the slowest workers (as in SSP), but based on an optimal synchronization time that minimizes the overall worker waiting time.

The key properties of ElasticBSP are:

- The server deals with online sequential decision making regarding the *optimal* time that the next synchronization barrier should be imposed (a time when the minimum waiting time for the entire system is achieved). Each decision is the solution of an online optimization with lookahead problem that utilizes information about the most recent time interval of each worker available to the server to predict their R future intervals (the *lookahead*). The need for a specific limit R comes from the need to control the algorithm's run time, since that can increase exponentially as R increases. A bound in R also ensures that ElasticBSP does not behave as the ASP model, but provides convergence guarantee and accuracy (similar to SSP).

- The convergence guarantee of the model follows the theoretical analysis of SSP [7], where a small iteration difference β exists in some period τ (a superstep). In the case of ElasticBSP, the iteration difference is bounded by the limit R to some period τ that is defined by the next best synchronization time. By the end of the period τ, the synchronization barrier b is posed to all the workers where gradient aggregation is carried out on the server, similarly to BSP — the model weights are then synchronized and will be available to all workers in the next iteration.

Consider the illustrative example of Figure 6.4, where n=10 workers need to synchronize. The server first predicts the R=15 future iteration intervals for each worker (dots in distinct color) based on their latest two contiguous `push` timestamps. Intervals between dots of the same color represent iteration intervals of a worker. Then, a decision needs to be made about the optimal time to impose a synchronization barrier that minimizes the overall waiting time for the 10 workers in wall-clock time. The squared dots (in red) in the example represent the intervals of each worker that inform the optimal synchronization time (minimum waiting time), determined by the distance between the leftmost and rightmost red square dots. In particular, the rightmost red square dot is where a barrier b shall be imposed. In Figure 6.4, worker 9, representing the leftmost red dot, will be the first to arrive in the barrier b and wait for the synchronization to occur. Note that for each worker, the respective red square dot might represent a `push` timestamp that occurs at a different iteration than that of other workers. In ElasticBSP, the server maintains this information and learns the best time to signal the workers to perform a `pull` operation and synchronize the weight parameters. In Figure 6.4, `pull` will be broadcasted by the server shortly after the rightmost red square dot, which represents the time that the slowest worker (i.e., worker 4) uploaded the gradients to the server, and therefore aggregation of gradients is possible.

6.4 The Problem

In this section, we formally define the problem of interest. Part of the originality of our work can be attributed to the fact that we formalize the problem as an online optimization with lookahead problem. Online optimization with lookahead is an optimization paradigm that utilizes a limited preview of future input data (lookahead) to inform sequential decision making under incomplete information. For distributed deep learning, this optimization paradigm provides a better description of a parameter server's informational state than other well-established synchronization models that assume nothing is known about the future. Specifically, the solution to the optimization problem will provide a *prediction of the optimal time* t^*_{sync} *to impose the next synchronization barrier* b. That is possible assuming a stable cluster run-

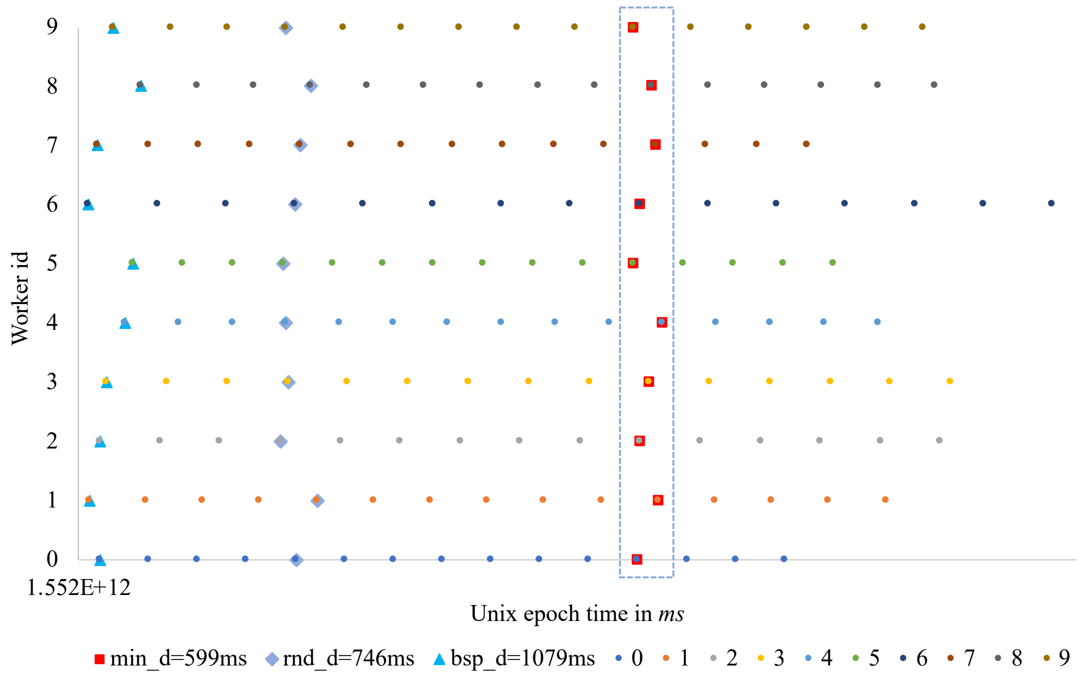


Figure 6.4: Predicting the time to synchronize. The sky blue triangles in the first column are the starting points of the predicted future iterations at which we have learned the most recent iteration intervals of all workers. Each dot represents the ending time of an iteration. Workers (labelled from 0 to 9) have unique color. Starting from them, we predict next R=15 future iteration intervals of the 10 workers. The objective is to find the dots of distinct colors that are closest to each other (i.e., dots vertically aligned near any time-spot). Three strategies are shown for comparison: ZipLine (*min_d* in red squares), a random barrier pick (*rnd_d* in grey blue diamonds) and classic BSP (*bsp_d* in sky blue triangles).*min_d*, *rnd_d* and *bsp_d* represent the overall workers' waiting time cost in milliseconds in wall-clock time. ZipLine has the minimum cost (599ms).

ning environment – a realistic assumption as we discuss very soon. Our approach is fundamentally different to current approaches that either strictly control the time to impose a synchronization barrier (i.e, BSP) or determine it based on ad hoc runtime decisions (i.e, SSP).

Recall that during execution the parameter server receives `push` requests coming from n workers. Once a request is received, the server keeps record of the worker p and associates it with a timestamp representing the time of the request. Most data

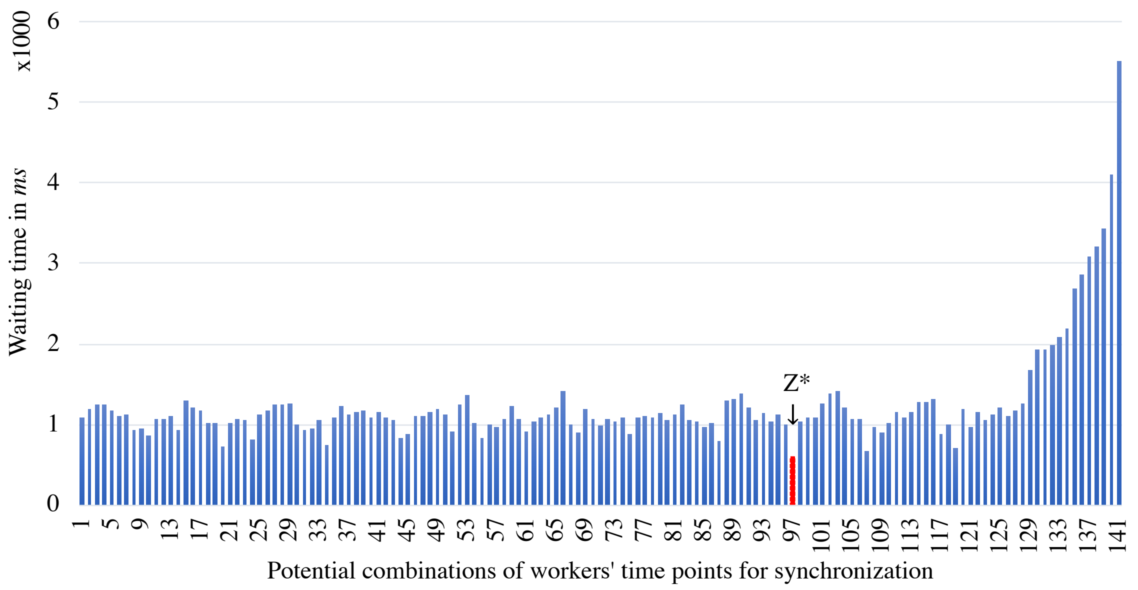


Figure 6.5: ZipLine scans the points (`push` timestamps) on the timeline as in Figure 6.7 and evaluates all 141 possible sets Z (each of which consists of distinct workers) of the example in Figure 6.4 in ascending order. For each set Z, the overall worker waiting time d_Z is obtained and plotted on y-axis. ZipLine finds the optimal set Z^* that minimizes d_{Z^*} (i.e., 599 milliseconds) which lies at the $97th$ combination (highlighted in red).

centers follow the high availability (99.999% available time) criteria practice [70]. Therefore, it is realistic to assume a stable running environment, where the duration of subsequent iterations of a worker (including batch processing and gradient computing) are unlikely to fluctuate in the foreseeable future.

This assumption allows to heuristically predict the R future iteration intervals of each worker (see Figure 6.3 and 6.4), based on their most recent iteration interval history. For instance, if a worker p arrives at time t and presents an iteration interval γ^p_{iter}, then the future R iterations will be predicted to occur at time $t + \gamma^p_{iter}, t + 2 \cdot \gamma^p_{iter}, \ldots, t + R \cdot \gamma^p_{iter}$. Given the R future iteration intervals of each worker, then the problem is to determine the best time in the future to impose the next synchronization barrier – a time that will minimize the waiting time for all workers.

Note that this assumption is not limiting our approach. In practice, if a worker does not behave in a predictable way, say due to a failure, it will be taken out of the distributed computation (and will be ignored by our method). Similarly, if a minor fluctuation in the duration of an interval occurs, for example due to a network glitch of any of the workers, our method might be affected by an erroneous prediction which might cause a sub-optimal imposition of a synchronization barrier. Nonetheless, that error will not be carried forward to the next decision. That is because we always use the most up-to-date history of worker time intervals to find the time to impose the next synchronization barrier. As such, the effect to the overall approach will be negligible (if any).

6.4.1 Problem Definition

Consider the parameter server framework and let n workers. For each worker $p \in [1, n]$, the server predicts R future iteration intervals and stores a set $S^p = \{e^p_1, e^p_2, ..., e^p_R\}$, where $e^p_i, i \in [1, R]$ represents the ending timestamp of the i-th iteration of worker p. We only need to store ending timestamps (and no starting timestamps), as they are the only ones required for determining a synchronization time. From each of the n sets $S^p, p \in [1, n]$, we pick one element $e^p_j, j \in [1, R]$ to construct a new set $Z = \{e^p_j\}, p \in [1, n]$ of $|Z| = n$ ending timestamps (one for each worker). The smaller timestamp in Z ($\min(Z)$) represents the faster worker and the larger timestamp in Z ($\max(Z)$) represents the slowest worker, respectively. Then, the difference $d_Z = \max(Z) - \min(Z)$ represents the waiting time of the fastest worker. Note that $t_{sync} = \max(Z)$ represents the time to impose the synchronization barrier b and d_Z represents the waiting time overhead related to the current barrier b; that is because other workers' waiting times are overlapped by the fastest worker's (i.e., $d_Z = \tau^b_{wait}$). As it becomes clear, any set Z can determine the time t_{sync} and represents one candidate solution to the optimization problem. From the space of all candidate combinations of Z, we are looking for the optimal one Z^* (as shown in Figure 6.5) that exhibits the minimum d_{Z^*} and determines the optimal t^*_{sync} at which we will impose the next synchronization barrier b. The following formalizes the problem.

Problem 1. *Given n workers, each represented by a set $S^p = \{e^p_1, e^p_2, ..., e^p_R\}$, where e^p_i is the ending time of the i-th future iteration of worker $p, p \in [1, n]$ and $R \in \mathbb{N}$, find a solution Z containing one element from each S^p, such that d_Z is minimized.*

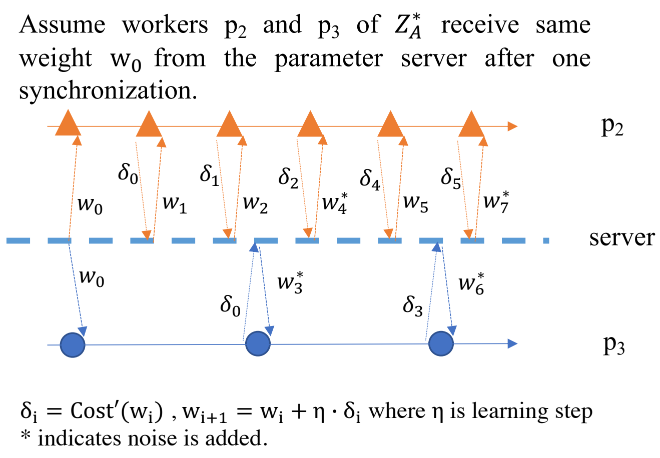
Hence, our objective function is:

$$Z^* = \operatorname*{argmin}_Z d_Z$$

Knowing Z^*, we can determine the optimal time $t^*_{sync} = max(Z^*)$ to impose the next synchronization barrier b. At the end of the distributed training of a model, a set B of synchronization barriers will have been imposed and the overall waiting time overhead can be derived by **(6.2)**.

6.4.2 Choosing Z^* for ElasticBSP

It is possible that more than one Z^* exists among all possible Z combinations derived from the n sets S^p (see example in Figure 6.8). From a machine learning perspective (i.e., considering an iterative convergence optimization), the Z^* with the *earliest (smallest)* timestamp is preferred for ElasticBSP. Figure 6.6 illustrates the reason by depicting two cases. Figure 6.6(a) illustrates how SGD is working asynchronously under the parameter server setting. For instance, two workers are shown with different processing speeds on a mini-batch (one iteration), where worker p_2 runs faster than worker p_3. If a synchronization between p_2 and p_3 happens at every iteration following BSP, then no delayed gradients updates are introduced. On the contrary, in the case of asynchronous gradient updates, noise might be injected (i.e., staled gradients with large staleness value [7]) to SGD and lead to divergence, especially when the noise is accumulated over a large number of iterations. Intuitively, synchronizing as early as possible reduces the staled gradients and decreases their staleness. Theorem 3 and Corollary 3.1 formalize on this intuition. In a nutshell, in the case of asynchronous gradient updates, the longer it takes for a synchronization barrier to be imposed, the more stale the gradients will be, potentially rendering the weight convergence uncertain. In Figure 6.6(b), there are 5 workers and two equivalent optimal solutions Z^* are considered (Z_A^* and Z_B^*). In each solution, a different number


Assume workers p2 and p3 of Z_A^* receive same weight w0 from the parameter server after one synchronization.
p2
server
p3
δ_0 δ_1 δ_2 δ_4 δ_5
w_0 w_1 w_2 w_4^* w_5 w_7^*
w_0 δ_0 w_3^* δ_3 w_6^*
δ_i = Cost′(w_i) , $w_{i+1} = w_i + \eta \cdot \delta_i$ where η is learning step
* indicates noise is added.

(a)

Workers	# iterations completed	
	Z_A^*	Z_B^*
p_1	13	3
p_2	20	5
p_3	10	1
p_4	12	3
p_5	17	2
d_Z	$d_{Z_A^*}(=d_{Z_B^*})$	$d_{Z_B^*}(=d_{Z_A^*})$
d^{iter}	10	4 ✓

(b)

Figure 6.6: (a) shows how staled gradients δ bring noise to the weight updates. The noise may drift the weight convergence away from the optimal direction (i.e., to poor local minima of the loss function). (b) Suppose we have two optimal solutions Z_A^* and Z_B^*. Each column lists the number of iterations each worker has completed in each solution. Both solutions Z_A^* and Z_B^* have equal d_Z. The solution with the smaller iteration difference d^{iter} (i.e., 4) introduces fewer staled gradients to the model weights and therefore is preferred.

of iterations have been completed by each worker since last synchronization. In Z_A^*, the iteration difference d^{iter} between the faster and slower workers is 10, whereas in Z_B^* the difference is 4. According to Figure 6.6(b), Z_B^* will have fewer staled gradient updates than Z_A^*, because its iteration difference d^{iter} (i.e., 4) is smaller than that of Z_A^* (i.e., 10). We show that if the workers have different processing speed (on an iteration), then the iteration difference d^{iter} among them increases as asynchronous gradient updates last longer (i.e., the synchronization barrier is imposed at later time). Therefore, if multiple Z^*s with the same d_{Z^*} exist, we prefer to impose a synchronization barrier represented by the *earliest time* t^*_{sync}.

Theorem 3 (The earlier the synchronization barrier is imposed, the less staler the gradients). Consider two workers p_i and p_j, $i, j \in [1, n]$ and $i \neq j$, and their iteration intervals t_{p_i} and t_{p_j}, respectively, where $t_{p_i} < t_{p_j}$ (i.e., p_i is faster than p_j). Let the difference of the number of iterations they complete within a time period t be given by $d^{iter} = \frac{t}{t_{p_i}} - \frac{t}{t_{p_j}}$. Then, within a time period $t' > t$, it is $d^{iter'} > d^{iter}$, where $d^{iter'}$ is the iteration difference within t'.

Proof. Suppose we have two workers $p_i, p_j, i, j \in [0, n], i \neq j$ and their iteration intervals being $t_{p_i} = \lambda t_{p_j}, \lambda \in (0, 1)$ (i.e., p_i being faster than p_j). Within a time period t, their iteration difference is given by $d^{iter} = \frac{t}{t_{p_i}} - \frac{t}{t_{p_j}} = \frac{t(t_{p_j} - t_{p_i})}{t_{p_i} t_{p_j}} = \frac{t(1-\lambda)}{t_{p_i}}$ ①. Within a longer time period $t' = t + k, k > 0$, their iteration difference is given by $d^{iter'} = \frac{(t+k)(1-\lambda)}{t_{p_i}}$ ②. Therefore, for a longer run time $t' > t$, the iteration difference is given by ② − ① $= \frac{k(1-\lambda)}{t_{p_i}} > 0$ since $\lambda \in (0, 1)$. □

Intuitively, if a synchronization period (a superstep) lasts longer, then a larger iteration difference among workers is introduced by asynchronous gradients updates.

Corollary 3.1. Since less staler gradients have less negative impact to the rate of convergence [7], early synchronization is preferred in asynchronous parallel training phase for better convergence with respect to the rate and the accuracy.

6.5 Methodology

To address **Problem 1**, we first investigate a brute-force approach, *naive search*. Since *naive search* cannot scale to a large number of workers, we develop an optimized version of it named *FullGridScan*. Then, we introduce our method ZipLine and its optimized variants ZipLineOpt and ZipLineOptBS. Table 6.1 summarizes the computation and space complexity of the different approaches.

6.5.1 Exhaustive Search Methods

Naive search. In order to find the minimum difference d_{Z^*}, a straightforward approach is to use a brute-force search method. The method determines the optimal solution Z^* by first constructing all candidate solutions Z. Each Z contains one element from each set $S^p = \{e_1^p, e_2^p, ..., e_R^p\}, p \in [1, n], R \in \mathbb{N}$. Since there are n sets S^p (one for each worker), and each set S^p has R elements, there are R^n candidate solutions in total. Then, for each candidate solution Z, it computes its d_Z value. The solution Z^* for which the minimum value d_{Z^*} is yielded, is the optimal solution. The computation complexity of *naive search* is $\mathcal{O}(R^n)$. The space complexity is $\mathcal{O}(R^n)$ to store the R^n possible solutions.

Algorithm 3 GridScan - finds the set Z^* with minimum d_{Z^*}

```
 1: procedure $\textsc{min}_d\textsc{Set}(\mathcal{M})$
    ▷ the $n \times R$ Matrix $\mathcal{M}$ with predicted points
 2:     $Z^* \leftarrow \varnothing$
    ▷ the set $Z^*$ takes $n$ elements with unique worker id $p$, $p \in [1, n]$
 3:     $d_{Z^*} \leftarrow \infty$
 4:     find the row $\mathcal{M}_{p_b}$ with the smallest initial time $\mathcal{M}_{p_b,1}$ from set $\{\mathcal{M}_{p,1}\}$
    ▷ $\mathcal{M}_{p_b,1} = \min(\{\mathcal{M}_{p,1}\}), p \in [1, n]$, $\{\mathcal{M}_{p,1}\}$ the first column of $\mathcal{M}$
 5:     for each point $e \in$ worker $\mathcal{M}_{p_b}$ do
 6:         $Z \leftarrow \varnothing$
 7:         add $e$ to $Z$
 8:         for each worker $\mathcal{M}_p \in \mathcal{M}, \mathcal{M}_p \neq \mathcal{M}_{p_b}$ do
 9:             for each point $\mathcal{M}_{p,i} \in \mathcal{M}_p$ do
10:                 $\mathcal{M}_{p,min} \leftarrow \operatorname{argmin}_{\mathcal{M}_{p,i}} |\mathcal{M}_{p,i} - e|$
                    ▷ the shortest distance point to $e$
11:                 add $\mathcal{M}_{p,min}$ to $Z$
12:         $d_Z \leftarrow \max(Z) - \min(Z)$
            ▷ compute minimum maximum difference of $Z$
13:         if $d_Z < d_{Z^*}$ then
14:             $Z^* \leftarrow Z$
15:             $d_{Z^*} \leftarrow d_Z$
16:     return $Z^*$                                    ▷ the set with $d_{Z^*}$
```

GridScan. Since *naive search* is not practical, we present an optimized heuristic brute-force method, *GridScan* (Algorithm 3). *GridScan* will eventually serve as the basic component of *FullGridScan*. Let the future R iteration timestamps of n workers form a $n \times R$ matrix $\mathcal{M}$, where each row of the matrix $\mathcal{M}_p$ represents a worker $p, p \in [1, n]$ and each row element $\mathcal{M}_{p,i} = e_i^p, i \in [1, R]$ represents each of the R predicted iteration interval points (timestamps). Now, observe that for any *designated* element in $\mathcal{M}$, we can search for elements belonging to the remaining rows that have timestamps close to the timestamp of the designated element (as shown in line 10). Then, the *elements found in the remaining rows* along with the original *designated element* form a candidate solution Z, for which we can obtain d_Z. Accordingly, we can consider a *designated row*, and we can iteratively consider all its R elements and define R candidate solutions Z, each one associated to each of the *designated elements* of the *designated row.* Following this procedure, the optimal solution Z^* will be the one that exhibits the minimum d_{Z^*}. To guarantee we do not miss any early element (on the timeline), we set the *designated row* to be that with the minimum (earliest) timestamp (i.e., $\mathcal{M}_{p,1}$) (see line 4). Finding the designated row costs $\Theta(n)$. The total computation complexity is $\mathcal{O}(R^2 n)$. The outer loop iterates over the R elements of the designated row and the inner loop iterates over the R elements of each of the remaining $n-1$ rows to construct candidate solutions. During the search, we only need to maintain the currently best solution Z^* (i.e., n elements) and the associated d_{Z^*}. Therefore, it requires storage space $\theta(n)$. Along

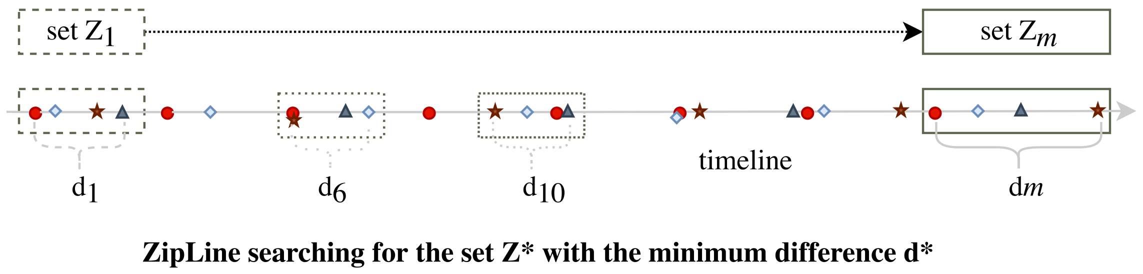

set Z1
set Zm
timeline
d1
d6
d10
dm
ZipLine searching for the set Z* with the minimum difference d*

Figure 6.7: ZipLine scans all elements on the timeline, from left to right, one element at a time. When a solution Z of n distinct elements is formed, d_Z is computed. At the end of the process the optimal solution Z^* is found that yields the minimum d^*_Z. If multiple solutions exhibit the same d^*_Zs, then the solution Z that occurred first (chronologically) is selected by Corollary 3. In this example, d_6 and d_{10} have the same minimum value — Z_6 associated with d_6 is chosen as the optimal solution.

with the storage of the Rn elements of the matrix $\mathcal{M}$, the space complexity is $\mathcal{O}(Rn)$.

FullGridScan. In *GridScan*, R candidate solutions Z are constructed, each associated with a d_Z. Due to its design, it is possible that *GridScan* will miss some solutions with a smaller d_Z. In order to discover potentially better solutions during the search, we also implement *FullGridScan*. *FullGridScan* iterates over all n rows (i.e., workers) of the matrix $\mathcal{M}$, each time defining a *designated row* $\mathcal{M}_p, p \in [1, n]$ and applying *GridScan* using Algorithm 3, but skipping line 4. Therefore, *FullGridScan* considers Rn candidate solutions compared to the R candidate solutions considered by *GridScan*. As *FullGridScan* needs to apply *GridScan* n times, its computation complexity is $\mathcal{O}(R^2n^2)$. The storage complexity of *FullGridScan* remains the same as *GridScan*.

6.5.2 Search by ZipLine

We are now in position to describe ZipLine, our proposed method to solve the optimization problem. ZipLine (Algorithm 4) determines the optimal solution Z^* in two steps:

Step 1 (Initialization). We merge the elements of all sets $S^p = \{e_1^p, e_2^p, ..., e_R^p\}, p \in [1, n], R \in \mathbb{N}$ into a set Ω and sort them in an ascending order; recall that elements e_i^p represent timestamps so this is an ordering of timestamps over the time axis (line 3), where the leftmost element represents the earliest event. Ω serves as the *lookahead* of the optimization problem. At the beginning of the method, we initialize Z with n elements of Ω, each one coming from a different set $S^p = \{e_1^p, e_2^p, ..., e_R^p\}, p \in [1, n], R \in \mathbb{N}$ (line 4-7); we pick the leftmost element of each S^p. Then, we compute d_Z of the candidate solution Z (i.e., the difference between minimum and maximum elements of Z representing a worker's waiting time). Initially, $Z^* = Z$ and $d_{Z^*} = d_Z$.

Step 2 (Iterative procedure). Starting from the leftmost element the method iteratively scans all elements of Ω one at a time, until Ω is empty (line 11-20). At each iteration, the method constructs a candidate solution Z. Recall that a candidate solution Z must contain one element from each set $S^p = \{e_1^p, e_2^p, ..., e_R^p\}, p \in [1, n], R \in \mathbb{N}$ (see Figure 6.7). As new elements are evaluated, the method only needs to check the superscript value p of each element e_i^p in the current solution Z. This is to prevent adding an element to the solution that comes from the same set S^p (i.e., same worker p). Whenever a new element is added to a solution Z, the previous element of that set S^p is removed/replaced. For each solution Z, the associated d_Z is computed. If d_Z is smaller than the current d_{Z^*}, then $Z^* = Z$ and $d_{Z^*} = d_Z$.

At the end of the process, after Rn iterations (the size of Ω), the optimal solution Z^* that exhibits the smaller d_Z^* is found. At each iteration, the removal/replacement operation of an element costs $\mathcal{O}(n)$. Therefore, the total computation complexity of ZipLine is $\mathcal{O}(Rn^2)$. The algorithm only uses $\Theta(n)$ space to store Z^* and $\mathcal{O}(Rn)$ for the storage of all elements in Ω, therefore the space complexity is $\mathcal{O}(Rn)$.

Algorithm 4 ZipLine - finds solution Z^* with minimum d_{Z^*}

```
1: procedure $\textsc{min}_d\textsc{Set}(\Omega)$                    $\triangleright$ the merged set $\Omega$
2:     $Z \leftarrow \varnothing$
   $\triangleright$ the set $Z$ takes $n$ elements with unique $p$ value, $p \in [1, n]$
3:     $\Omega \leftarrow \mathrm{sort}(\Omega)$
   $\triangleright$ sort $\Omega$ in ascending order by element's value (timestamp)
4:     while $|Z| < n$ do
5:         $\omega \leftarrow$ the most left element of $\Omega$
           $\triangleright$ $\omega$ is $e_i^p$ where $i \in [1, R]$ and $p \in [1, n]$
6:         add $\omega$ to $Z$
   $\triangleright$ old element of $Z$ is removed if its $p$ value is same as $\omega$
7:         $\Omega \leftarrow \Omega - \omega$
8:     $d_Z \leftarrow \max(Z) - \min(Z)$
   $\triangleright$ compute difference of minimum maximum elements of $Z$
9:     $Z^* \leftarrow Z$
10:    $d_{Z^*} \leftarrow d_Z$
11:    while $\Omega \neq \varnothing$ do
       $\triangleright$ the solution is obtained when $\Omega$ is empty
12:        $Z \leftarrow Z - \min(Z)$                 $\triangleright$ $Z$ is in ascending order as of $\Omega$
13:        while $|Z| < n$ do
14:            $\omega \leftarrow$ the most left element of $\Omega$
15:            add $\omega$ to $Z$
16:            $\Omega \leftarrow \Omega - \omega$
17:        $d_Z \leftarrow \max(Z) - \min(Z)$
18:        if $d_Z < d_{Z^*}$ then
19:            $Z^* \leftarrow Z$
20:            $d_{Z^*} \leftarrow d_Z$
21:    return $Z^*$                                  $\triangleright$ the set with $d_{Z^*}$
```

ZipLine Optimality

We claim that our greedy algorithm, ZipLine, leads to an optimal solution and we provide a formal proof of the claim.

Theorem 4. [**ZipLine optimality**] ZipLine leads to an optimal solution Z^*.

Proof. (Sketch) The proof is based on propositions of two lemmata. Lemma 5 claims that ZipLine always finds a legal solution. Lemma 6 claims that there is not a better one. $\square$

Lemma 5. ZipLine always finds a legal solution. A legal solution Z includes one element from each set $S^p = \{e_1^p, e_2^p, ..., e_R^p\}, p \in [1, n], R \in \mathbb{N}$.

Proof. The proof is based on contradiction. Suppose that it did not. Then, there will be at least two elements in Z that come from the same set S^p (i.e., two elements of the same color). Recall that by its definition ZipLine always begins with a legal solution Z that includes one element from each set S^p. As the algorithm scans new elements it performs a removal/replacement operation. For every new element, the algorithm first determines its color and then replaces it with the element of that same color that is currently in hand. Since we assumed that there are at least two elements in solution Z that come from the same set S^p, there must have occurred a replacement operation where the replaced element was of a different color (and not of the same color). However, this is an illegal operation/step. We have therefore reached a contradiction – our assumption was wrong and our algorithm always finds a legal solution. $\square$

Lemma 6. ZIPLINE produces a solution Z^* that is no worse than other solutions X. Otherwise, for any alternate solution $X \neq Z^*$, it holds that $d_X \geq d_{Z^*}$.

Proof. The proof is based on contradiction. Suppose there is an alternate legal solution $X \neq Z^*$ with $d_X < d_{Z^*}$. Since X is a legal solution, it consists of one element from each set $S^p = \{e_1^p, e_2^p, ..., e_R^p\}, p \in [1, n], R \in \mathbb{N}$. We also know that $X \subset \Omega$ and that elements in Ω are sorted, so let the elements $x_i \in X$ be $x_1 < x_2 < ... < x_n$, where $i \in [1, n]$ and $|X| = n$. Therefore, it is $d_X = x_n - x_1$. Now, assume there exists $o \in \Omega$, such that $o \notin X$ and $x_j < o < x_k$ where $j, k \in [1, n]$ and $j < k$. Note that o is an element of a set $S^p, p \in [1, n], R \in \mathbb{N}$, that is already represented in the current solution X (i.e., element o's color is identical to that of an $x_i \in X$, where $i \in [1, n]$). We distinguish two possible cases for o's color:

- If o's color is identical to that of x_1 (or x_n), then ZIPLINE would have replaced x_1 (or x_n) with o and have formed a solution X^* instead, where $X^* \neq X$. That would have resulted in a better solution, since now $d_{X^*} < d_X$. Thus, X cannot be the optimal solution in that case.

- If o's color is identical to that of an element $x \in X$, where $x_1 < x < x_n$ (i.e., $x \in \{x_2, ..., x_{n-1}\}$), then ZIPLINE would have either replaced x with o (or not) to form a solution X^*. In either case, $d_{X^*} = d_X$, so ZIPLINE would have found a solution as good as X.

Thus, our assumption was wrong and ZipLine produces a solution Z^* that is no worse than other solutions X. $\square$

Lemma 5 and lemma 6 prove our claim stated in Theorem 4 that ZipLine leads to an optimal solution Z^*.

6.5.3 ZipLine Optimizations

As ZipLine scans through new elements of Ω, a new candidate solution Z' is formulated every time by replacing an element of the previous candidate solution Z with the new element. Recall that a candidate solution Z must contain one element from each set $S^p = \{e_1^p, e_2^p, ..., e_R^p\}, p \in [1, n], R \in \mathbb{N}$. Satisfying this condition dominates the cost of ZipLine. That is because once the replacement is done, the new difference $d_{Z'}$ of the solution Z' needs to be computed and that requires searching for the potentially updated $min(Z')$. If $d_{Z'}$ is smaller than d_{Z^*}, then we store Z' to Z^* and continue. The replacement operation and the search of $min(Z')$ in Z' take $\mathcal{O}(n)$ to complete regardless using a hashtable or array data structure to store the elements of a candidate solution Z'. Thus, the cost of ZipLine becomes $\mathcal{O}(Rn^2)$ for scanning through the Rn elements in Ω. Below, we discuss two optimizations that manage to effectively reduce the ZipLine cost to $\mathcal{O}(n \log n)$, by reducing the cost of the replacement operation and the search of $min(Z')$ in Z' to $\mathcal{O}(\log n)$. The first optimization takes advantage of the fact that elements are sorted in the time axis, from the earliest to the latest to prune the search space of finding the $min(Z')$.

The second is an implementation optimization that utilizes a more efficient data structure to boost the search process.

Search Space Pruning Optimization (ZipLineOpt)

We utilize an array data structure to store the elements of a candidate solution Z' in temporal order of arrival. When a new element of Ω is evaluated by ZipLine, we know that the oldest element in the array has the minimum timestamp and the most recently added element has the maximum timestamp.

To determine the $min(Z')$ there are two cases:

(i) with probability $\alpha, \alpha \in (0, 1)$, the oldest element in the array has the same color as the newly added element, and therefore we simply remove the oldest element and set the new minimum timestamp to be the one succeeding it in the array. After adding the new element, we can compute $d_{Z'}$ directly due to random access to the array data structure. Thus, the cost of searching for $min(Z')$ is reduced to $\mathcal{O}(1)$ with probability α;

(ii) with probability $1 - \alpha, \alpha \in (0, 1)$, the oldest element in the array does not have the same color as the newly added element, and therefore there is no need to re-compute the $d_{Z'}$, since it is larger than d_Z by Theorem 7.

Theorem 7. Suppose $e_i \in Z$ and $e_1 < e_2 < ... < e_n$ where $i \in [1, n]$ and $|Z| = n$ and let $d_Z = e_n - e_1$. Now, assume we add a new element e', where $e' \geq e_n$ to formulate a new solution Z'. To satisfy that $|Z'| = n$, we need to remove the element e_i from

Z that has the same color as e'. If $i > 1$ of the removed e_i, then $d_{Z'} \geq d_Z$, where $d_{Z'} = e' - e_1$.

Proof. Let $d' = e' - e_n$. We know $d' \geq 0$ as $e' \geq e_n$. By definition, $d_{Z'} = e' - e_1 = d' + e_n - e_1 = d' + d_Z \geq d_Z$. $\square$

We still have to remove the element of the same color from the array data structure. To do so, we need to traverse the array Z until the redundant element is found and removed. This delete operation costs $\mathcal{O}(n)$ and occurs with probability $1 - \alpha$. At last, we add the new element to Z' and move on to the next element on Ω. We call this variant of ZipLine as ZipLineOpt. Since Ω has Rn elements, the computation cost of ZipLineOpt becomes $\alpha \cdot \mathcal{O}(Rn) + (1-\alpha) \cdot \mathcal{O}(Rn^2)$. Table 6.4 shows that empirically it is the fastest when $n \leq 100$.

Implementation Optimization (ZipLineOptBS)

To further optimize ZipLineOpt, we focus on the case of $1-\alpha$ which requires $\mathcal{O}(n)$ traversal of the array Z to remove the element of the same color as the newly added element. The optimization is achieved by employing an auxiliary data structure carrying extra information. Specifically, a table $\mathcal{M}$:$n \times R$, similar to the matrix $\mathcal{M}$ of *GridScan* in Section 6.5.1 that stores R future interval timestamps of n workers. $\mathcal{M}$ is constructed and stored in memory when the set Ω is constructed, as shown in Figure 6.4. For every element of Ω, its color (or worker id), its interval index i (the ith future iteration of a worker p) and the timestamp associated to that index

Table 6.1: Summary of computation and space complexities of methods.

Algorithm	**Computation**	**Space**
GridScan (heuristic)	$\mathcal{O}(R^2n)$	$\mathcal{O}(Rn)$
FullGridScan	$\mathcal{O}(R^2n^2)$	$\mathcal{O}(Rn)$
Zipline	$\mathcal{O}(Rn^2)$	$\mathcal{O}(Rn)$
ZiplineOpt	Best: $\mathcal{O}(Rn)$, Worst: $\mathcal{O}(Rn^2)$	$\mathcal{O}(Rn)$
ZiplineOptBS	Best: $\mathcal{O}(Rn)$, Wrost: $\mathcal{O}(Rn \log n)$	$\mathcal{O}(Rn)$

Table 6.2: Synthetic datasets with varying number of n and R.

SmallR	**n**	10	100	200	400	600	800	1000
	R	15	15	15	15	15	15	15
LargeR	**n**	10	100	200	400	600	800	1000
	R	150	150	150	150	150	150	150

i are stored as an element of the matrix $\mathcal{M}$. Thus, for any new element e_i^p of Ω to be added to Z, we can retrieve that worker's previous element e_{i-1}^p from $\mathcal{M}$ in $\mathcal{O}(1)$ based on its worker id $p \in [1, n]$ and iteration index $i \in [1, R]$. Once we get the timestamp of the worker's previous iteration e_{i-1}^p, we can locate it in the array Z by the timestamp value using a binary search (BS) algorithm, since the elements of Z are sorted (in an ascending order). The BS reduces the search cost to $\mathcal{O}(\log n)$. Thus, the cost of the $1 - \alpha$ case is reduced to $(1 - \alpha) \cdot \mathcal{O}(Rn \log n)$. We call this variant of ZipLine as ZipLineOptBS. The use of the auxiliary data structure $\mathcal{M}$ with binary search (BS) allows ZipLineOptBS to bring the cost of ZipLine down to $\alpha \cdot \mathcal{O}(Rn) + (1 - \alpha) \cdot \mathcal{O}(Rn \log n)$, where $\alpha \in (0, 1)$.

6.6 Experimental Evaluation

In this section, we run experiments that aim to evaluate:

A. The runtime performance of the ZipLine algorithm and its variants compared to two sensible baselines, FullGridScan and GridScan. We also evaluate the scalability performance of ZipLine as a function of the number of workers n and the parameter R.

B. The performance of the ElasticBSP model compared to the classic BSP, ASP and state-of-art synchronization models, such as SSP. We are interested to find which one converges faster and to a higher accuracy; also which one is able to complete a fixed number of epochs faster?

6.6.1 ZipLine Performance

In this section, we try to answer item A by evaluating ZipLine and its variants with the sensible baselines.

Dataset: To evaluate the performance of algorithms, we generate synthetic datasets based on various realistic scenarios of the number of workers n and values of the parameter R. For each worker we randomly define its iteration interval to be in the range of $1000ms$ to $1500ms$. Table 6.2 lists the different configurations of datasets. SmallR ($R = 15$) is used to evaluate the performance of algorithms as a function of the number of workers. LargeR ($R = 150$) is used to evaluate the scalability of the algorithms as a function of R.

Environment: All experiments about ZipLine's runtime performance are run on a server with 24x Intel(R) Xeon(R) CPU E5-2620 v3 @ 2.40GHz and 64GB ram.

The algorithms and the datasets generator are developed in `C++11`.

Zipline Ability to Find the Optimal Solution

We have provided theoretical results about the optimality of ZIPLINE in Section 6.5.2. Here, we provide empirical evidence of ZIPLINE's ability to find the optimal solution and compare it to the ones found by the competitive algorithms. We experiment with the datasets of Table 6.2. For each run scenario, we compute the d_{Z^*} of the solution Z^* that each algorithm is able to find (Table 6.3), along with the associated computation time cost (Table 6.4). The results reported are averages of 10 runs.

The results suggest that the family of the ZIPLINE methods always finds the optimal solution Z^* – same as the one found by the exhaustive method, *FullGridScan*. But, it is able to do so **one or two orders of time faster** (depending on the ZIPLINE variant considered). This is because the search space of candidate solutions evaluated by *FullGridScan* is much larger than that of ZIPLINE. On the other hand, the *GridScan* heuristic, while is comparable to ZIPLINE in terms of speed, consistently fails to find the optimal solution. Among the variants of ZIPLINE, ZIPLINEOPTBS adds some overhead compared to ZIPLINEOPT for smaller number of workers, but as the number of workers increases is able to outperform it.

Table 6.3: Search accuracy comparison on d_{Z^*} — the least waiting time in milliseconds/ms found by algorithms.

Number of workers (n)	10	50	100	500	1000	10	50	100	500	1000	Success
Predicted iterations (R)	15	15	15	15	15	150	150	150	150	150	rate
ZipLine	599	1103	1228	1351	1403	518	1055	1175	1307	1382	100%
ZipLineOpt	599	1103	1228	1351	1403	518	1055	1175	1307	1382	100%
ZipLineOptBS	599	1103	1228	1351	1403	518	1055	1175	1307	1382	100%
FullGridScan	599	1103	1228	1351	1403	518	1055	1175	1307	1382	100%
GridScan	599	1103	***1345***	***1357***	***1405***	518	***1103***	***1253***	***1344***	***1401***	**30%**

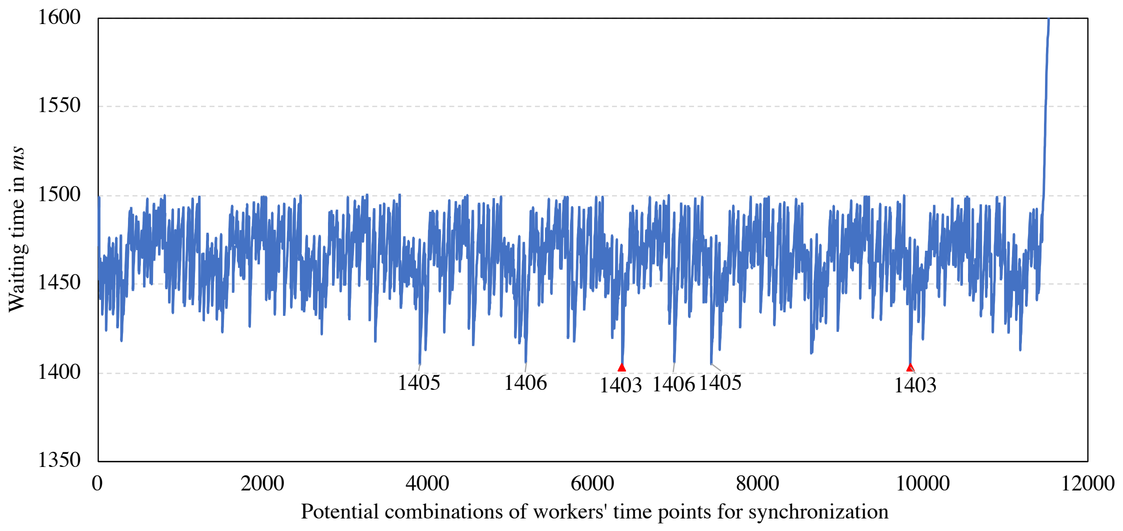


Figure 6.8: A plot of the cost d_Z of candidate solutions Z evaluated by ZipLine. As ZipLine iteratively scans the elements of Ω from the leftmost to the rightmost element on the timeline, we compute the cost of each candidate solution Z and its cost d_Zs (the smaller the cost d_Z the better the solution Z). The run is based on the SmallR dataset of Table 6.2, for n=1,000 and R=15. There are 15,000 elements in Ω. ZipLine evaluates a total of 13,795 candidate solutions, *FullGridScan* 15000 candidate solutions and *GridScan* only 15 candidate solutions. We only plot the d_Z values that have a cost of less than 1600 milliseconds; we highlight the optimal d_{Z^*} and a few sub-optimal d_Zs. Red triangles indicate d_{Z^*}.

ZipLine Scalability

The number of candidate solutions Z formed by elements of the Matrix $\mathcal{M}$:$n \times R$ increases exponentially to the number of workers n and polynomially to the value of the parameter R (i.e., R^n), as described in Section 6.5.1. We have showed in Table

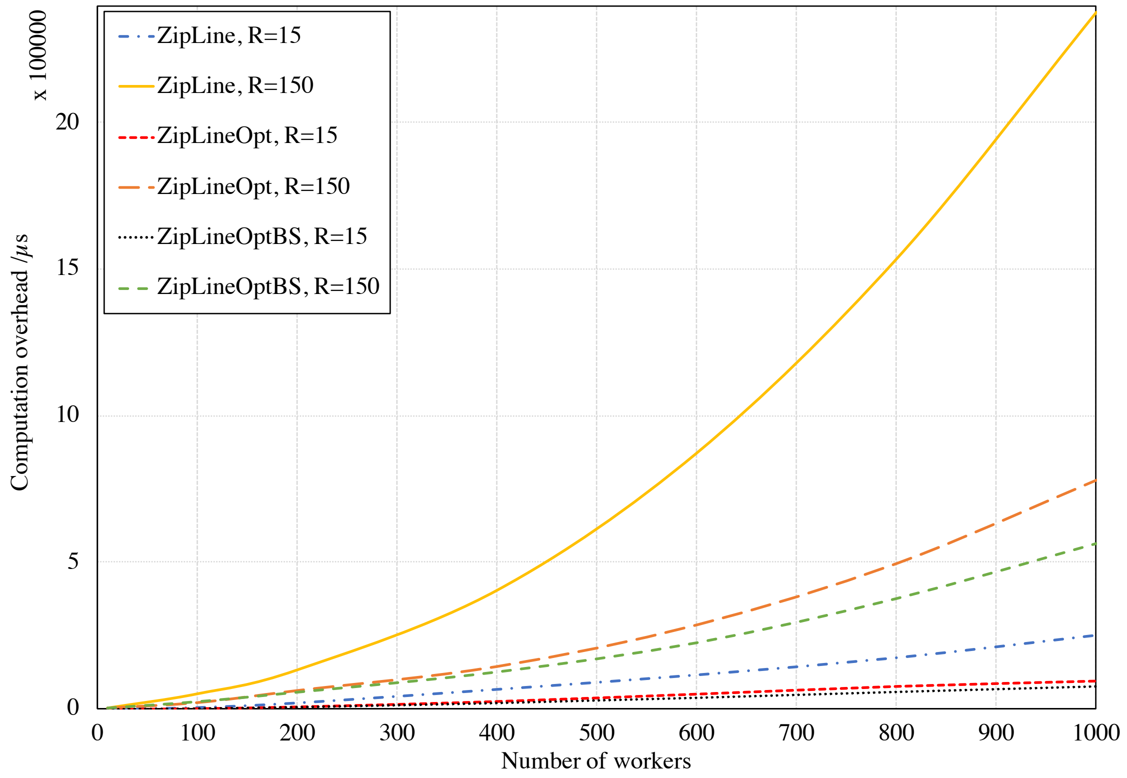


Figure 6.9: Computation time cost comparison of ZipLine and its variants. The cost of ZipLine and its variants increases as the number of workers n and the value of parameter R increases. Both *ZipLineOpt* and *ZipLineOptBS* outperform the basic *ZipLine*. For larger values of R ($R \geq 100$), *ZipLineOptBS* outperforms *ZipLineOpt*.

Table 6.4: Computation time of algorithms in microseconds/μs.

Algorithm	10 Workers		100 Workers		1000 Workers	
	R=15	R=150	R=15	R=150	R=15	R=150
ZipLine	1.49e2	1.32e3	6.37e3	4.99e4	2.53e5	2.38e6
ZipLineOpt	**0.90e2**	**8.08e2**	**2.46e3**	**1.93e4**	9.39e4	7.78e5
ZipLineOptBS	1.24e2	1.15e3	2.65e3	2.30e4	**7.68e4**	**5.66e5**
FullGridScan	1.54e3	4.67e4	8.13e4	2.15e6	4.04e6	2.07e8
GridScan	1.68e2	5.50e3	***1.11e3***	4.38e4	***7.45e3***	***2.57e5***

6.4 that as the number of workers n increases, the computation time of *FullGridScan* increases much faster than that of the ZipLine family of algorithms. We have also discussed that for a fixed number of workers, as R increases the computation time of *FullGridScan* increases much faster. To further elaborate on the behavior of the ZipLine variants, in Figure 6.9 we provide an illustration of their run time comparison (we don't plot *FullGridScan* to improve the comparative analysis of the ZipLine variants). It can be observed that when n is small (e.g., below 100), *ZipLineOpt* is faster than *ZipLineOptBS*. But, as n increases (e.g., above 200), *ZipLineOptBS* outperforms *ZipLineOpt*. That is because *ZipLineOptBS* has to maintain two auxiliary data structures and accessing the second one (the table $\mathcal{M}$) introduces extra cost, compared to *ZipLineOpt*. However, this extra cost is amortized into many scan iterations when the number of elements of Ω is large (i.e.,Rn) (i.e., most iterations are very fast and only few of them are expensive). Observe in Figure 6.9 that for R=150, *ZipLineOptBS* grows significantly slower than *ZipLineOpt*. Same trend is depicted for R=15 as well – as n increases, *ZipLineOptBS* outperforms *ZipLineOpt*. Therefore, we recommend to employ *ZipLineOpt* when n is small (e.g., $n \leq 100$) and employ *ZipLineOptBS* otherwise. Most research and industrial labs can typically afford a deep learning cluster with 4 to 8 nodes (workers), in which scenario *ZipLineOpt* is preferred. The *GridScan* heuristic can serve as an alternative when one is willing to sacrifice accuracy (i.e., finding optimal solution) to gain in scalability.

6.6.2 Distributed Deep Learning using ElasticBSP

We compare the performance of ElasticBSP with BSP, SSP and ASP [1] by training DNN models from scratch on a parameter server setting. For SSP, we set its threshold parameter to s=3 to ensure it convergences and achieves higher accuracy, as suggested by Ho et al. [7]. For ElasticBSP, we set $R = \{15, 30, 60, 120, 240\}$. In case of training large-sized DNN models on large-scale dataset of higher image resolution, we additionally consider $R = \{480, 960\}$. We run each experiment three times and report the medium result of the overall test accuracy.

Platform: We implement ElasticBSP with *ZipLineOpt* into MXNet [22] which supports the BSP and ASP models. When $n \leq 10$, *ZipLineOpt* performs the best in terms of run time.

Cluster Environment: The experiments are run on 4 IBM POWER8 machines. Each machine has 4 NVIDIA P100 GPUs, 512 GB ram and 2×10 cores. The machines are connected with Infiniband EDR. Each server connects directly to a switch with 100 Gbps bandwidth.

Datasets & DNN models: We first train the downsized AlexNet [13], ResNet-50 and ResNet-100 [31] on two image classification datasets CIFAR-10 and CIFAR-100 [58] with 28×28 pixels resolution. Then, we move on to train the large-sized DNN model (VGG-16 [12]) on the large dataset ImageNet 1K [71] with 256×256

[1]BSP is generally practised in industry and is supported by Pytorch, TensorFlow and MXNet. The latter two also support ASP. SSP is available in Petuum and we implement it into MXNet. Other state-of-the-art methods are only available in the research field and are non-trivial to be implemented into MXNet due to different architectures.

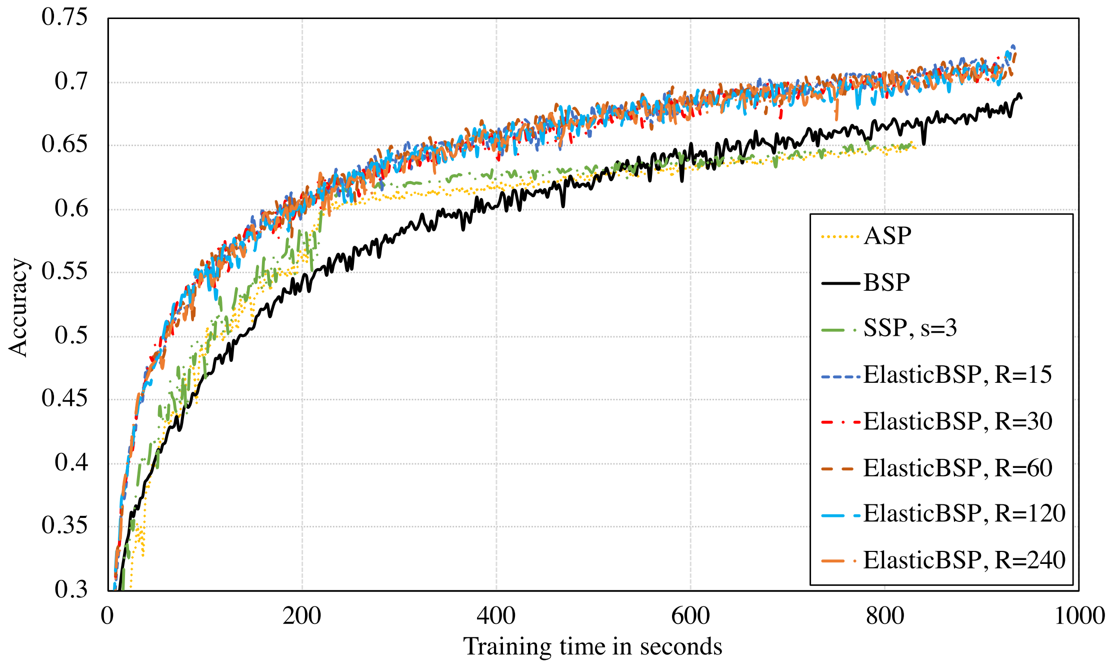


Figure 6.10: Downsized AlexNet on CIFAR-10 dataset ($n = 4$)

pixels resolution to see if ElasticBSP can also support industry ready applications.

Downsized-AlexNet on CIFAR-10

To train this DNN model, we set mini-batch size to 128, epoch to 400, learning rate to 0.001 and weight decay to 0.0005. As can be observed in Figure 6.10, ElasticBSP converges faster and to higher accuracy than the rest of the synchronization models. BSP converges to a higher accuracy than ASP and SSP, but is slower. Regarding the effect of the parameter R of ElasticBSP, as R increases, *ZipLineOpt* requires more computation time to determine the optimal synchronization time, for each synchronization barrier imposed. As a result, ElasticBSP with larger R has larger computation cost. This is done without any significant benefit to the accuracy,

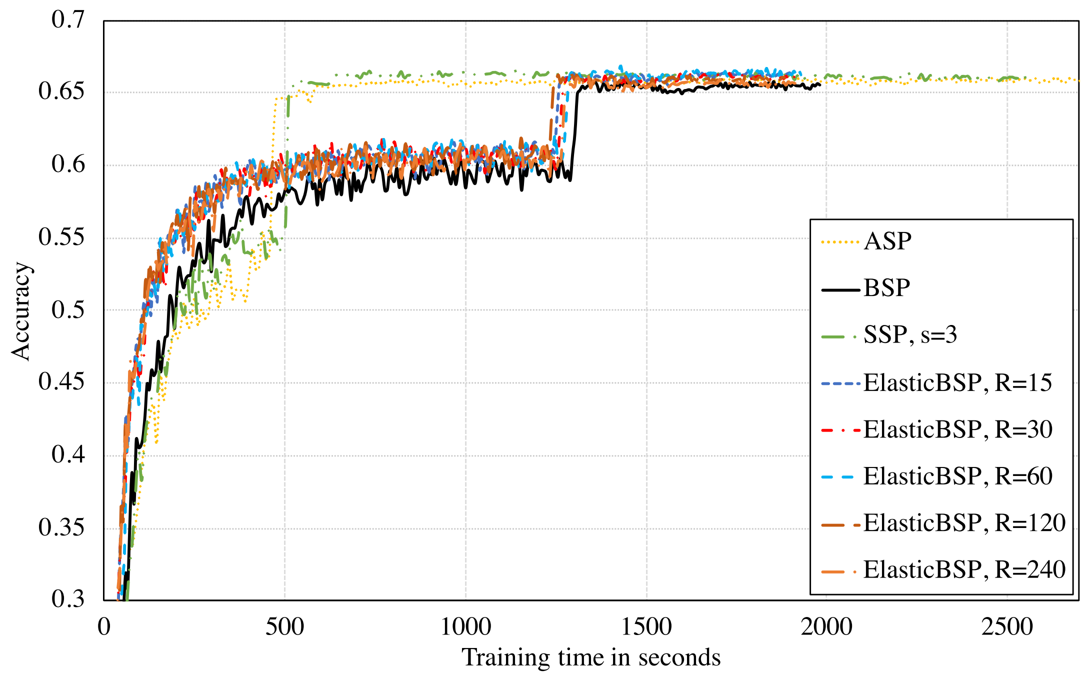


Figure 6.11: ResNet-50 on CIFAR-100 dataset ($n = 4$)

therefore smaller R values are preferred for smaller models. Irrelevant to accuracy, the faster model to finish the 400 epochs is SSP, followed by ASP, ElasticBSP (R={15, 30}) and BSP.

ResNet-50 and ResNet-110 on CIFAR-100

To train this DNN model, we set mini-batch size to 128, epoch to 300, learning rate to 0.5 and decay to 0.1 at epoch 200 for both ResNet-50 and ResNet-110. The results are shown in Figure 6.11 and Figure 6.12.

For ResNet-50 in Figure 6.11, ElasticBSP converges faster and to a slightly higher accuracy than BSP. Besides, ElasticBSP converges to a slightly higher accuracy than ASP and SSP. ResNet-110 has a similar model size to ResNet-50, but

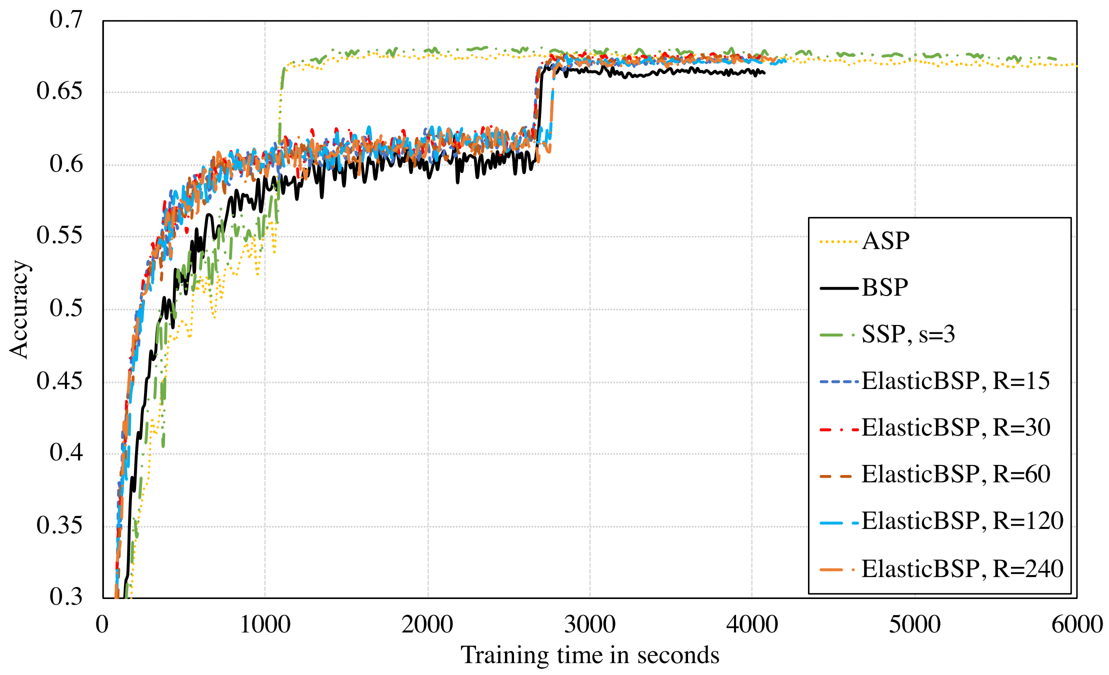


Figure 6.12: ResNet-110 on CIFAR-100 dataset ($n = 4$)

takes much more computing time due to its deeper convolutional layers. Thus, when computation time is long and communication time is relatively short, there is little opportunity to save on communication time during training. As is shown in Figure 6.12, ElasticBSP converges at a similar rate to BSP, but reaches to slightly higher accuracy. Regarding the effect of the parameter R of ElasticBSP, as R increases to 120, its training time becomes slightly larger than that of BSP due to extra computation time required by *ZipLineOpt* to compute the time to impose a synchronization barrier. ASP and SSP converge faster, but require more training time than ElasticBSP and BSP. Recall that ASP and SSP have no bulk synchronization barriers therefore have larger iteration throughput causing faster convergence than ElasticBSP and BSP. But larger iteration throughput, introduces more frequent communication between workers and server and leads to increased number of weight updates. Meanwhile, weight updates have to be computed in order (as mentioned in Section 6.2). Thus, their tasks are queued on the server, which introduces extra delay. We further elaborate on this issue in Section 6.6.2.

On ResNet models, the faster model to finish the 300 epochs is ElasticBSP, followed by BSP, SSP and ASP. An exception is the ElasticBSP (R={120, 240}), which are slower than BSP on ResNet-110.

VGG-16 on ImageNet 1K

To train this DNN model, we set mini-batch size to 256, epoch to 19[2], learning rate to 0.01, weight decay to 0.0005 and decay 0.2 at epoch 10, 15, 18. The results are shown in Figure 6.13.

Compared to distributed paradigms with zero staled gradient updates, such as BSP, ElasticBSP converges 1.77× faster and achieves 12.6% higher final accuracy. Compared to distributed paradigms with zero staled gradient updates, such as BSP, ElasticBSP converges 1.77× faster and achieves 12.6% higher final accuracy.Compared to distributed paradigms with staled gradients updates, ElasticBSP is learning faster than SSP and ASP despite the fact that it also has staled gradient updates during training. We also observe that while both SSP and ASP complete the fixed 19 epochs faster than ElasticBSP, they fail to learn. This is

[2]The testing policy restricts a job's running time to at most 24 hours, so the most epochs VGG-16 can complete on ImageNet 1K is set to 19.

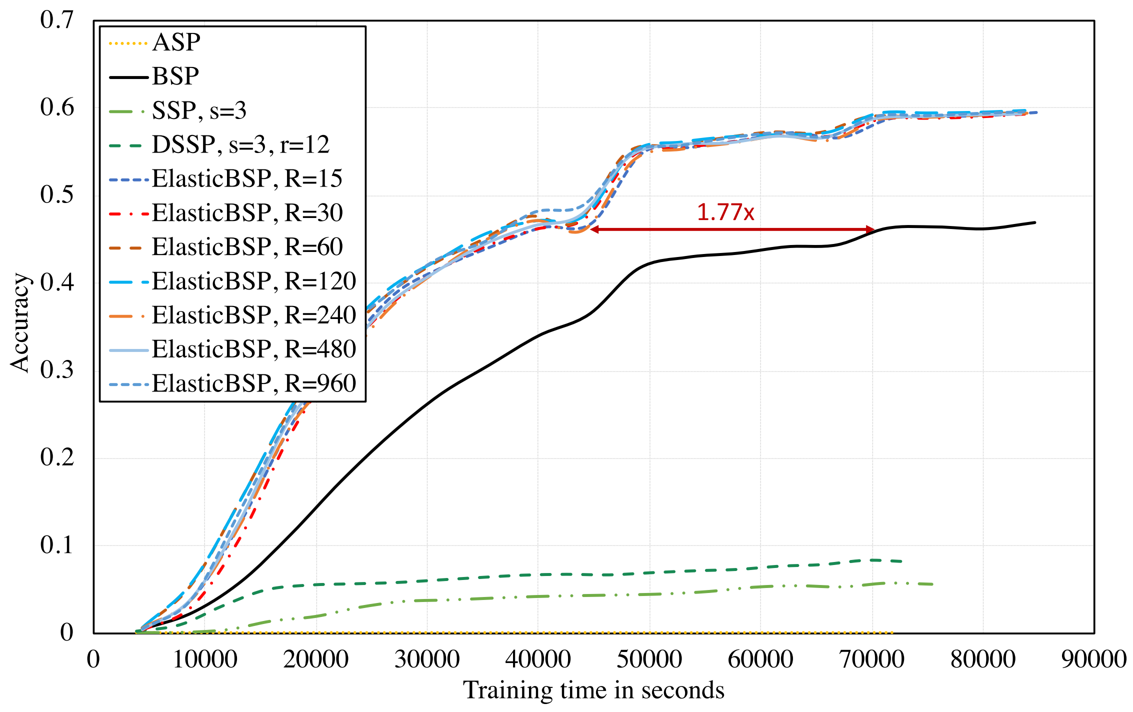


Figure 6.13: VGG-16 on ImageNet 1K dataset ($n = 4$)

due to staled gradients that are constantly present in the training process. Note that for a fixed training time or fixed number of epochs (e.g., 19), ElasticBSP always converges to a higher accuracy than the other synchronization models.

VGG-16 is the largest model size in our analysis, containing 3 fully connected layers (FCLs) that are very sensitive to the staled gradient updates, similarly to AlexNet containing 2 FCLs in Figure 6.10 (both have similar learning curves). ASP is unable to learn due to the large number of staled gradients allowed during training. SSP can hardly learn despite it restricts the staleness of its gradients using within a small fixed staleness threshold (e.g., 3).

Since VGG is the largest convolutional neural network model, we experimented with additional two larger values of R for ElasticBSP (i.e., $R = \{480, 960\}$). We wanted to see if increasing the computation time of ZipLine and possibly introducing larger c-staled gradients (introduced by a larger R) might harm the convergence speed and accuracy of ElasticBSP. The results in Figure 6.13 show there is only a small variance introduced by the different Rs, meaning that increasing the value of R does not significantly harm the performance of ElasticBSP. This is due to the fact that the computation time of ZipLine is rather negligible compared to the data transmission time of the parameters.

On VGG-16 model, the fastest model to finish the 19 epochs is ASP, followed by SSP, ElasticBSP and BSP. An exception is ElasticBSP (R=15), which is slightly slower than BSP.

Lastly, we add DSSP to the comparison. Figure 6.13 shows DSSP with the threshold range $s \in [3, 15]$ is better than SSP but is inferior to ElasticBSP.

Discussion

The results above, run on different DNN size and model architectures, provide empirical evidence that ElasticBSP converges to higher accuracy than BSP and can require less training time. That is when R is not too large for the case of small-sized DNNs, and not too small for the case of large-sized DNNs. As the size of the DNNs with FCLs grows larger, ElasticBSP shows its superior performance with respect to both converge speed and accuracy. Note that the variation in the performance of ElasticBSP, BSP, SSP and ASP on different DNNs is expected. The downsized AlexNet and the VGG-16 are similar to each other, but different to ResNets; AlexNet contains 2 FCLs and VGG-16 has 3 FCLs, whereas ResNets has no FCLs and therefore fewer model parameters (i.e., smaller size). Training FCLs requires much less computation time compared to convolutional layers (CVLs), while their representation consists of many more parameters than CVLs leading to larger model sizes. On the other hand, training convolutional neural networks without FCLs, such as ResNets, requires much more computing time, but consumes less communication time due to the smaller model size (compared to FCL networks). When the ratio of communication time to computation time is small, there is less room to save on the training time. More detailed analysis of the different behavior on DNNs with different ratio of computation to communication time can be found in [41]. In addition, FCLs are sensitive to the staled gradient updates since their representation consists of a large number of parameters (many more than those of CVL representations that consist of shared parameters [14]). Zhao et al. [14] provide a thorough rationale on the different performances of ASP, BSP and SSP on distributed training of various DNN models, with or without FCLs.

Another important observation that derives from Figure 6.13 is that ElasticBSP is able to take advantage of the restricted staled gradient updates (via elastically imposing synchronization barriers defined by ZipLine). Due to elasticity, it is able to skip poor local optima and saddle points of other synchronization methods. Poor local optima and saddle points are well-known obstacles to the model learning using serial SGD. In effect, executing parallel SGD using ElasticBSP can be considered as a hybrid model alternating behaviors of an ASP-like and a BSP-like mode: An ASP-like mode occurs during a synchronization period (a superstep τ) and a BSP-like mode occurs across different synchronization barriers b. During the ASP-like mode, staled gradients are generated by the stragglers that harm the convergence, which is traded off for large iteration throughput. At the end of each τ, a BSP-like mode is switched on which eliminates all the staled gradients by imposing a weight synchronization to all workers. Such a periodic clearing of the staled gradients mitigates any significant harm to the convergence. The length of τ is limited by ZipLine following Corollary 3.1. Therefore, staled gradients with small staleness value (due to small length of τ) functions as a regularizer preventing the learning model being overfit to the training set.

6.7 Related work

Several important works closely related to our research have already been cited throughout the manuscript. Here, we extend more on work that tries to mitigate the slow down caused by the straggler problem of the BSP which is generally practised in industry. These work use techniques from different areas, for instances, one uses reinforcement learning to find the optimal scheme, one uses generative model to predict the best synchronization time and one uses adaptive learning rate to mitigate the harm of the staled gradients.

Preempting straggler jobs. A-BSP [52] handles the straggler problem by terminating the iteration job corresponding to the slowest worker once the fastest workers have completed their jobs. That way, the waiting time is eliminated. The remaining data of the terminated job of the slowest worker is prioritized in the next iteration. This design is limited to the CPU cluster where samples are processed one after another. But in a GPU cluster, a batch of samples are processed all at once in parallel – GPU takes a batch of samples per iteration and computes the gradients. Decreasing the data of a batch (iteration) does not reduce the computation time of GPU. Therefore, GPU does not support preempting jobs [33]. Terminating a job means losing all the computed result on that batch of data.

Speculative execution to avoid stragglers jobs. Chen et al. [1] deal with the straggler problem by adding k extra backup workers to the distributed training with n workers. In this approach, $k+n$ workers are running to train the model. At each iteration, the server only accepts the gradient updates of the n workers that arrived faster and moves on to the next iteration. The gradients of the k slower workers are dropped. While this approach saves on waiting time (as the n faster workers are needed per iteration), the computing resources allocated to the k slower workers are wasted.

Bayesian prediction for synchronization. Tend and Wood [53] use a large complex generative model, Bayesian Distributed SGD (BDSGD), to predict the optimal synchronization time or barrier for workers via first predicting the current run-time (iteration interval excluding communication time) of workers based on their historical (run-time) data distributions, and then determine the optimal synchronization barrier according to the predicted workers' run-times. Yet it follows Chen et al.'s approach [1] of dropping the gradients of the slower workers that arrive after the predicted optimal synchronization barrier. BDSGD assumes there is a correlation between run-times of workers in each synchronization. Thus, it predicts the current run-time of each worker based on the posterior distributions (i.e., historical run-times of workers). BDSGD requires pre-training on its generative model for the run-time prediction and costs more computing resources per predication in the actual DNN model training. Consequently, it increases the training time due to its prediction time cost exceeds the workers' run-time per synchronization. In practice, it has to reuse the result of the synchronization prediction in every few contiguous synchronizations to reduce the frequency of running the learned generative prediction model. BDSGD converges to a similar accuracy as BSP with its complex prediction model. Our ElasticBSP converges to a significantly higher accuracy than BSP on large DNNs despite it uses a greedy algorithm to solve the optimization problem.

Sparse synchronizations. EASGD [72] first proposed to reduce the communication cost by allowing the workers to update weights locally per iteration and to synchronize with the server only at a fixed communication period. However, with fewer synchronizations, the divergence among local models can result in an error-convergence [41]. ADACOMM [41] uses periodic-averaging SGD (PASGD) for bulk synchronization in which workers are doing local updates for τ iterations before a weight synchronization. That way, the communication cost of both uploading gradients and downloading weights from the server occurs only once every τ iterations. In practice, ADACOMM estimates the optimal τ for a bulk synchronization of local weights based on the training loss, but does not address the straggler problem. In contrary to ADACOMM's approach of assigning the same τ to all workers, our ElasticBSP predicts the optimal synchronization time for all workers, where each worker can have a different τ.

Auto-synchronization. Zhu et al. [73] proposed to address the straggler problem using reinforcement learning (RL) in full automation by formulating the synchronization policy as a RL problem. By using deep Q-learning algorithm [74], the learned RL policies were able to speedup the training on shallow DNNs and small datasets. Yet the authors admit the work has its limitations and it is not ready for the real-world scenario. Our ElasticBSP uses the lightweight but effective greedy algorithm to minimize the waiting time caused by the stragglers. Therefore, we save the time cost of a few episodes required by RL on training the RL model to learn the policies before it can be deployed to the actual DNNs training. Besides, ElasticBSP demonstrates outstanding performance on large DNNs and large datasets.

Dynamic soft synchronization. Other than BSP, the straggler problem also exists in SSP despite that SSP was devised to solve the straggler problem in BSP. DSSP [14] introduces the dynamic staleness threshold to minimising the waiting time for SSP which uses a fixed staleness threshold. By predicting the future iterations for the fastest and the slowest workers based on their most recent iterations, DSSP finds the optimal staleness threshold between pre-defined lower and upper thresholds of SSP per iteration for the fastest workers at run time to minimize the waiting time of the fastest workers. The prediction of the future R iterations of workers based on their most recent iteration history is also practised in our ElasticBSP.

Adjusting the learning rate of staled gradients. Dutta et al. [56] promote asynchrony (i.e., ASP) and provide a thorough analysis on the parallel SGD with and without synchronization. They proposed an adaptive learning rate scheme to accelerate the convergence speed in wall-clock time for ASP. Similar to ElasticBSP, they measure iteration intervals (processing time) by using the consecutive `push` timestamps of each worker. In [56], the harm that the staled gradients bring to the convergence was diminished by tuning down the learning rate according to the staleness value of the weight parameters at run time on the parameter server side. However, the tuning process is expensive on the computing resources for large DNNs and not scalable when the number of workers increases, since it requires the server to always keep a copy of the recently read weight parameters for every worker. The authors also confirm that the synchronization is critical to the convergence speed and aim to increase the synchronization frequency in the future.

6.8 Epilogue

In this chapter, we introduce ElasticBSP for distributed DNN model training, using the parameter server framework. Our model is orthogonal to other types of DNN training optimizations. ElasticBSP is relaxing the bulk synchronization requirement of BSP and allows asynchronous gradient updates to a certain extent to ensure the quality of convergence and achieve higher accuracy. As a result, it increases the iteration throughput of the workers, limits the staled gradients to a small amount and their staleness values to a small number, which brings less harm to the convergence [7]. ElasticBSP operates in two phases. First R future iterations of each worker are predicted. Then, ZipLine (or any of its variants) is applied to determine the time to impose the next synchronization barrier that minimizes the overall workers' waiting time overhead. ZipLine is a one-pass algorithm with linearithmic complexity $\mathcal{O}(Rn\log n)$ and adds a minimal overhead on the server, so it can be easily ported on popular distributed machine learning frameworks. The experimental results show that ElasticBSP provides faster convergence than BSP for a number of DNNs while achieving higher (or comparable) accuracy than other state-of-the-art synchronization models, including ASP, SSP and BSP, on different datasets. Overall, our work provides theoretical and empirical evidence that ElasticBSP offers better generalization.

Chapter 7

Conclusion

7.1 Summary of Contributions

In this thesis, we aim to improve the convergence speed (in time unit instead of iteration unit) without sacrificing the accuracy for distributed deep learning in the parameter server setting via dynamic deterministic optimization on the synchronizations of weights of workers partially and entirely which increases the efficiency of the distributed training. We do not merely consider developing the efficient communication strategies but also include the convergence quality of DNN models such as the final test accuracy a model converges to and the convergence speed. First, we introduce the foundations of distributed deep learning in the parameter server setting. Then, we introduce the optimization properties and the evaluation metrics in the distributed training setting. In Chapter 5 we present the partial synchronizations determined by dynamic online adaptation on the staleness threshold within a given range, named DynamicSSP (DSSP) which improves the convergence speed and the accuracy of SSP in average. In particular, DSSP significantly outperforms SSP in the heterogeneous environment with respect to the convergence speed and the accuracy where workers have different computational capacities. Then, in Chapter 6, having observed the important role of the bulk synchronizations is playing in training large DNN models (with fully connected layers) based on numerous and diversified experiments of SSP, DSSP and BSP in Chapter 5, we further develop a novel distributed model inspired by BSP and DSSP, named ElasticBSP which dynamically determines the optimal time point in an online setting for the bulk synchronizations by considering the run-time information of the computational capacities of workers. Empirically, ElasticBSP converges faster to a higher accuracy than BSP, SSP, DSSP and ASP by running diversified DNNs on different datasets which also suggests ElasticBSP offers better generalization. In particular, ElasticBSP outperforms the BSP on training large DNNs on ImageNet 1K with respect to both convergence speed ($\times 1.77$ faster) and overall accuracy (12.6% higher). Finally, we also observe a small amount of not-too-staled gradients participating in the weight updates in training serves as a regularizer adding gradient noise to the steps (weight updates) which drives the DNN model exploring new area by random steps (staled gradients) and skipping the poor local optima and the saddle point. However, we leave the concrete theoretical analysis of this observation as an open question for the future work since it is specific to the optimization problem and a little off the topic we discuss in the thesis.

7.2 Future Directions

For the contemporary distributed Deep Learning training, the optimization of finding the optimal balancing point between the accuracy and the convergence speed remains an open question. There are many potential directions to finding a good balance between improving the efficiency and maintaining the efficacy of the distributed training for DNNs.

Reinforcement Learning: Zhu et al. [73] proposed to address the straggler problem using reinforcement learning (RL) in full automation by formulating the synchronization policy as a RL problem despite it is limited to running shallow neural networks on the low dimensional dataset. In Chapter 5, DSSP requires hyperparameter s_L and s_U to be specified for the range of the staleness threshold. ElasticBSP from Chapter 6 has hyperparameter R as the number of estimated future iterations per worker. Such hyperparamters can be learned by RL in training via RL model interacting with the environment such as rewarding the action which produces the high accuracy and or the fast convergence speed. To learn an optimal online scheduling policy to enabling a fully automatic online scheduling distributed paradigm can be a promising future direction as the RL technique advances to capable of predicting good or even optimal hyperparamters without supervision (no human guidance).

Periodic Communication: Federated averaging learning (FEDAVG) [75] allows workers independently run a fixed number of epochs before they communicate to the parameter server for weight synchronization. The DNN model converges and achieves a high accuracy with the weighted average updates approach of FEDAVG. In Chapter 6, ElasticBSP follows the convention of the parameter server setting that workers, as scalable distributed computing resources for heavy computing (e.g., convolutional computing and backpropagation), compute the gradients whereas the parameter server is only in charge of the weight updates (light computing). Nonetheless, ElasticBSP is developed for dynamic distributed environment with elasticity. It can be adapted to the periodic synchronization framework where each worker can run different number of epochs per communication that is determined dynamically based on workers' computational capacities at run time. However, in FEDAVG each worker has to run the same fixed number of epochs per communication uniformly over the entire training. To combine ElasticBSP and FEDAVG could be an interesting future work to speed up the distributed deep learning as well as the federated learning via balancing the number of communication and the waiting time of workers.

Edge Computing for Distributed Online Deep Learning: Sahoo et al. [76] attempt to improve the online deep learning (ODL) by proposing Hedge Backpropagation (HBP) which allows learning DNNs on the fly in an online setting. However, HBP requires a lot of auxiliary parameters for any given ordinary DNN structure. Thus, it costs extra memory and computing time (on both feedforward and back-propagation computing). This drawback cancels the advantages of HBP brings, and hinders HBP to be deployed in practice. A feasible solution is to integrate HBP to the distributed computing environment. Meanwhile, edge computing is being known as the ideal environment for online learning by offering low latency and data privacy [77]. A small DNN model can run on an edge device such as a smart phone carried by an user. External computing help can come from nearby "central" edge servers if required. It will be interesting to develop a distributed edge computing platform to support distributed ODL so that we can have fast convergence supported by massive distributed edge computing as well as high accuracy achieved by ODL technique. Furthermore, joint learning can be helpful to enhance the learning model prediction (accuracy), for instance, in federated learning each DNN model running on an edge device exchanges the information (i.e., feature representations) with one another on some randomly selected central (edge) servers.

Bibliography

[1] J. Chen, X. Pan, R. Monga, S. Bengio, and R. Jozefowicz, "Revisiting distributed synchronous sgd," *arXiv preprint arXiv:1604.00981*, 2016.

[2] D. Das, S. Avancha, D. Mudigere, K. Vaidynathan, S. Sridharan, D. Kalamkar, B. Kaul, and P. Dubey, "Distributed deep learning using synchronous stochastic gradient descent," *arXiv preprint arXiv:1602.06709*, 2016.

[3] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, M. Mao, A. Senior, P. Tucker, K. Yang, Q. V. Le, *et al.*, "Large scale distributed deep networks," in *NIPS*, pp. 1223–1231, 2012.

[4] S. Zhang, C. Zhang, Z. You, R. Zheng, and B. Xu, "Asynchronous stochastic gradient descent for dnn training," in *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, pp. 6660–6663, IEEE, 2013.

[5] B. Recht, C. Re, S. Wright, and F. Niu, "Hogwild: A lock-free approach to parallelizing stochastic gradient descent," in *NIPS*, pp. 693–701, 2011.

[6] A. V. Gerbessiotis and L. G. Valiant, "Direct bulk-synchronous parallel algorithms," *Par. and Distr. Comput.*, vol. 22, no. 2, pp. 251–267, 1994.

[7] Q. Ho, J. Cipar, H. Cui, S. Lee, J. K. Kim, P. B. Gibbons, G. A. Gibson, G. Ganger, and E. P. Xing, "More effective distributed ml via a stale synchronous parallel parameter server," in *Advances in neural information processing systems*, pp. 1223–1231, 2013.

[8] W. Zhang, S. Gupta, X. Lian, and J. Liu, "Staleness-aware async-sgd for distributed deep learning," *arXiv preprint arXiv:1511.05950*, 2015.

[9] K. Lee and R. Bhattacharya, "On the relaxed synchronization for massively parallel numerical algorithms," in *2016 American Control Conference (ACC)*, pp. 3334–3339, IEEE, 2016.

[10] Z. Zhou, P. Mertikopoulos, N. Bambos, P. W. Glynn, Y. Ye, L.-J. Li, and F.-F. Li, "Distributed asynchronous optimization with unbounded delays: How slow can you go?," in *ICML*, pp. 1–10, 2018.

[11] E. P. Xing, Q. Ho, P. Xie, and D. Wei, "Strategies and principles of distributed machine learning on big data," *Engineering*, vol. 2, no. 2, pp. 179–195, 2016.

[12] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[13] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *NIPS*, pp. 1097–1105, '12.

[14] X. Zhao, A. An, J. Liu, and B. X. Chen, "Dynamic stale synchronous parallel distributed training for deep learning," in *ICDCS*, pp. 1507–1517, 2019.

[15] X. Zhao, M. Papagelis, A. An, B. X. Chen, J. Liu, and Y. Hu, "Elastic bulk synchronous parallel model for distributed deep learning," in *19th IEEE International Conference on Data Mining (ICDM)*, pp. 1504–1509, 2019.

[16] M. Li, D. G. Andersen, A. J. Smola, and K. Yu, "Communication efficient distributed machine learning with the parameter server," in *Advances in Neural Information Processing Systems*, pp. 19–27, 2014.

[17] H. Zhang, Z. Zheng, S. Xu, W. Dai, Q. Ho, X. Liang, *et al.*, "Poseidon: An efficient communication architecture for distributed deep learning on {GPU} clusters," in *USENIX*, pp. 181–193, 2017.

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

[19] "Apache MXNet." accessed 2018-08-01.

[20] X. Lian, C. Zhang, H. Zhang, C.-J. Hsieh, W. Zhang, and J. Liu, "Can decentralized algorithms outperform centralized algorithms? a case study for

decentralized parallel stochastic gradient descent," in *Advances in Neural Information Processing Systems*, pp. 5330–5340, 2017.

[21] X. Lian, W. Zhang, C. Zhang, and J. Liu, "Asynchronous decentralized parallel stochastic gradient descent," *arXiv preprint arXiv:1710.06952*, 2017.

[22] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, *et al.*, "Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems," *arXiv preprint arXiv:1512.01274*, 2015.

[23] H. Cui, H. Zhang, G. R. Ganger, P. B. Gibbons, and E. P. Xing, "Geeps: Scalable deep learning on distributed gpus with a gpu-specialized parameter server," in *Europ. Conf. on Comp. Syst.*, p. 4, ACM, 2016.

[24] J. Mao, W. Xu, Y. Yang, J. Wang, Z. Huang, and A. Yuille, "Deep captioning with multimodal recurrent neural networks (m-rnn)," *arXiv preprint arXiv:1412.6632*, 2014.

[25] J. Zhou, X. Li, P. Zhao, C. Chen, L. Li, X. Yang, Q. Cui, J. Yu, X. Chen, Y. Ding, *et al.*, "Kunpeng: Parameter server based distributed learning systems and its applications in alibaba and ant financial," in *KDD*, pp. 1693–1702, ACM, 2017.

[26] K. Al Nuaimi, N. Mohamed, M. Al Nuaimi, and J. Al-Jaroodi, "A survey of load balancing in cloud computing: Challenges and algorithms," in *2012 Second*

Symposium on Network Cloud Computing and Applications, pp. 137–142, IEEE, 2012.

[27] X.-W. Chen and X. Lin, "Big data deep learning: challenges and perspectives," *IEEE access*, vol. 2, pp. 514–525, 2014.

[28] A. Coates, B. Huval, T. Wang, D. Wu, B. Catanzaro, and N. Andrew, "Deep learning with cots hpc systems," in *ICML*, pp. 1337–1345, 2013.

[29] A. Hannun, C. Case, J. Casper, B. Catanzaro, G. Diamos, E. Elsen, R. Prenger, S. Satheesh, S. Sengupta, A. Coates, *et al.*, "Deep speech: Scaling up end-to-end speech recognition," *arXiv preprint arXiv:1412.5567*, 2014.

[30] J. Gehring, M. Auli, D. Grangier, D. Yarats, and Y. N. Dauphin, "Convolutional sequence to sequence learning," in *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pp. 1243–1252, JMLR. org, 2017.

[31] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *IEEE CVPR*, pp. 770–778, 2016.

[32] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," in *NIPS*, pp. 5998–6008, 2017.

[33] M. Bauer, H. Cook, and B. Khailany, "Cudadma: optimizing gpu memory bandwidth via warp specialization," in *SC*, p. 12, ACM, 2011.

[34] N. Dryden, T. Moon, S. A. Jacobs, and B. Van Essen, "Communication quantization for data-parallel training of deep neural networks," in *Machine Learning in HPC Environments*, pp. 1–8, IEEE, 2016.

[35] N. Strom, "Scalable distributed dnn training using commodity gpu cloud computing," in *Interspeech*, 2015.

[36] H. Li, A. Kadav, E. Kruus, and C. Ungureanu, "Malt: distributed data-parallelism for existing ml applications," in *Europ. Conf. on Comp. Sys.*, p. 3, ACM, 2015.

[37] P. Moritz, R. Nishihara, I. Stoica, and M. I. Jordan, "Sparknet: Training deep networks in spark," *arXiv preprint arXiv:1511.06051*, 2015.

[38] T.-Y. Lin, M. Maire, S. Belongie, J. Hays, P. Perona, D. Ramanan, P. Dollár, and C. L. Zitnick, "Microsoft coco: Common objects in context," in *European conference on computer vision*, pp. 740–755, Springer, 2014.

[39] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *2009 IEEE conference on computer vision and pattern recognition*, pp. 248–255, Ieee, 2009.

[40] M. Zinkevich, M. Weimer, L. Li, and A. J. Smola, "Parallelized stochastic gradient descent," in *Advances in neural information processing systems*, pp. 2595–2603, 2010.

[41] J. Wang and G. Joshi, "Adaptive communication strategies to achieve the best error-runtime trade-off in local-update sgd," *SysML Conf*, 2019.

[42] L. Luo, Y. Xiong, Y. Liu, and X. Sun, "Adaptive gradient methods with dynamic bound of learning rate," *arXiv preprint arXiv:1902.09843*, 2019.

[43] H. Cui, J. Cipar, Q. Ho, J. K. Kim, S. Lee, A. Kumar, J. Wei, W. Dai, G. R. Ganger, P. B. Gibbons, *et al.*, "Exploiting bounded staleness to speed up big data analytics," in *USENIX*, pp. 37–48, 2014.

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

[45] W. Dai, A. Kumar, J. Wei, Q. Ho, G. Gibson, and E. P. Xing, "High-performance distributed ml at scale through parameter server consistency models," in *Twenty-Ninth AAAI Conference on Artificial Intelligence*, 2015.

[46] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pp. 135–146, ACM, 2010.

[47] K. Siddique, Z. Akhtar, E. J. Yoon, Y.-S. Jeong, D. Dasgupta, and Y. Kim, "Apache hama: An emerging bulk synchronous parallel computing framework for big data applications," *IEEE Access*, vol. 4, pp. 8879–8887, 2016.

[48] H. Senger, V. Gil-Costa, L. Arantes, C. A. Marcondes, M. Marín, L. M. Sato, and F. A. Da Silva, "Bsp cost and scalability analysis for mapreduce operations," *Concurrency and Computation: Practice and Experience*, vol. 28, no. 8, pp. 2503–2527, 2016.

[49] M. F. Pace, "Bsp vs mapreduce," *Procedia Computer Science*, vol. 9, pp. 246–255, 2012.

[50] S. Landset, T. M. Khoshgoftaar, A. N. Richter, and T. Hasanin, "A survey of open source tools for machine learning with big data in the hadoop ecosystem," *Journal of Big Data*, vol. 2, no. 1, p. 24, 2015.

[51] X. Meng, J. Bradley, B. Yavuz, E. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. Tsai, M. Amde, S. Owen, *et al.*, "Mllib: Machine learning in apache spark," *The Journal of Machine Learning Research*, vol. 17, no. 1, pp. 1235–1241, 2016.

[52] S. Wang, W. Chen, A. Pi, and X. Zhou, "Aggressive synchronization with partial processing for iterative ml jobs on clusters," in *Proceedings of the 19th Int. Middleware Conference*, pp. 253–265, ACM, 2018.

[53] M. Teng and F. Wood, "Bayesian distributed stochastic gradient descent," in *NIPS*, pp. 6378–6388, 2018.

[54] X. Lian, Y. Huang, Y. Li, and J. Liu, "Asynchronous parallel stochastic gradient for nonconvex optimization," in *Advances in Neural Information Processing Systems*, pp. 2737–2745, 2015.

[55] H. Zhang, C.-J. Hsieh, and V. Akella, "Hogwild++: A new mechanism for decentralized asynchronous stochastic gradient descent," in *2016 IEEE 16th International Conference on Data Mining (ICDM)*, pp. 629–638, IEEE, 2016.

[56] S. Dutta, G. Joshi, S. Ghosh, P. Dube, and P. Nagpurkar, "Slow and stale gradients can win the race: Error-runtime trade-offs in distributed sgd," in *AISTATS*, pp. 803–812, 2018.

[57] "SOSCIP GPU." accessed 2018-08-01.

[58] A. Krizhevsky and G. Hinton, "Learning multiple layers of features from tiny images," tech. rep., Citeseer, 2009.

[59] J. Tompson, R. Goroshin, A. Jain, Y. LeCun, and C. Bregler, "Efficient object localization using convolutional networks," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 648–656, 2015.

[60] A. Krizhevsky, "One weird trick for parallelizing convolutional neural networks," *arXiv preprint arXiv:1404.5997*, 2014.

[61] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: a simple way to prevent neural networks from overfitting," *The Journal of Machine Learning Research*, vol. 15, no. 1, pp. 1929–1958, 2014.

[62] L. Perez and J. Wang, "The effectiveness of data augmentation in image classification using deep learning," *arXiv preprint arXiv:1712.04621*, 2017.

[63] Y. Xu, T. Xiao, J. Zhang, K. Yang, and Z. Zhang, "Scale-invariant convolutional neural networks," *arXiv preprint arXiv:1411.6369*, 2014.

[64] L. Kang, P. Ye, Y. Li, and D. Doermann, "Simultaneous estimation of image quality and distortion via multi-task convolutional neural networks," in *Image Processing (ICIP), 2015 IEEE International Conference on*, pp. 2791–2795, IEEE, 2015.

[65] A. Neelakantan, L. Vilnis, Q. V. Le, I. Sutskever, L. Kaiser, K. Kurach, and J. Martens, "Adding gradient noise improves learning for very deep networks," *arXiv preprint arXiv:1511.06807*, 2015.

[66] S. Hadjis, C. Zhang, I. Mitliagkas, D. Iter, and C. Ré, "Omnivore: An optimizer for multi-device deep learning on cpus and gpus," *arXiv preprint arXiv:1606.04487*, 2016.

[67] R. Zhang and J. Kwok, "Asynchronous distributed admm for consensus optimization," in *International Conference on Machine Learning*, pp. 1701–1709, 2014.

[68] F. Dunke, *Online optimization with lookahead.* PhD thesis, Karlsruhe Institute of Technology, 2014.

[69] L. Liu, H. Jiang, P. He, W. Chen, X. Liu, J. Gao, and J. Han, "On the variance of the adaptive learning rate and beyond," *arXiv preprint arXiv:1908.03265*, 2019.

[70] K. Benz and T. Bohnert, "Dependability modeling framework: A test procedure for high availability in cloud operating systems," in *VTC*, pp. 1–8, IEEE, 2013.

[71] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "ImageNet: A Large-Scale Hierarchical Image Database," in *CVPR*, 2009.

[72] S. Zhang, A. E. Choromanska, and Y. LeCun, "Deep learning with elastic averaging sgd," in *NIPS*, pp. 685–693, 2015.

[73] R. Zhu, S. Yang, A. Pfadler, Z. Qian, and J. Zhou, "Learning efficient parameter server synchronization policies for distributed {sgd}," in *ICLR*, 2020.

[74] H. Van Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double q-learning," in *AAAI*, 2016.

[75] H. B. McMahan, E. Moore, D. Ramage, S. Hampson, *et al.*, "Communication-efficient learning of deep networks from decentralized data," *arXiv preprint arXiv:1602.05629*, 2016.

[76] D. Sahoo, Q. Pham, J. Lu, and S. C. Hoi, "Online deep learning: Learning deep neural networks on the fly," in *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence (IJCAI-18)*, pp. 2660–2666, 2018.

[77] J. Chen and X. Ran, "Deep learning with edge computing: A review," *Proceedings of the IEEE*, vol. 107, no. 8, pp. 1655–1674, 2019.