Sync-Switch: Hybrid Parameter Synchronization for Distributed Deep Learning

Shijian Li, Oren Mangoubi, Lijie Xu† and Tian Guo
Worcester Polytechnic Institute
†State Key Lab of Computer Science, Institute of Software, Chinese Academy of Sciences

2021 IEEE 41st International Conference on Distributed Computing Systems (ICDCS) | 978-1-6654-4513-9/21/$31.00 ©2021 IEEE | DOI: 10.1109/ICDCS51616.2021.00057

***Abstract*—Stochastic Gradient Descent (SGD) has become the de facto way to train deep neural networks in distributed clusters. A critical factor in determining the training throughput and model accuracy is the choice of the parameter synchronization protocol. For example, while Bulk Synchronous Parallel (BSP) often achieves better converged accuracy, the corresponding training throughput can be negatively impacted by stragglers. In contrast, Asynchronous Parallel (ASP) can have higher throughput, but its convergence and accuracy can be impacted by stale gradients. To improve the performance of synchronization protocol, recent work often focuses on designing new protocols with a heavy reliance on hard-to-tune hyper-parameters.**

In this paper, we design a hybrid synchronization approach that exploits the benefits of both BSP and ASP, i.e., reducing training time while simultaneously maintaining the converged accuracy. Based on extensive empirical profiling, we devise a collection of adaptive policies that determine how and when to switch between synchronization protocols. Our policies include both offline ones that target recurring jobs and online ones for handling transient stragglers. We implement the proposed policies in a prototype system, called *Sync-Switch*, on top of TensorFlow, and evaluate the training performance with popular deep learning models and datasets. Our experiments show that *Sync-Switch* can achieve ASP level training speedup while maintaining similar converged accuracy when comparing to BSP. Moreover, *Sync-Switch*'s elastic-based policy can adequately mitigate the impact from transient stragglers.

***Index Terms*—Distributed deep learning, synchronization policy design, empirical performance optimization**

I. Introduction

We are witnessing the increasingly widespread adoption of deep learning in a plethora of application domains. The unprecedented success of deep learning is, in large part, powered by rapid model innovations, which in turn critically depend on algorithms and systems support for training. One of these innovations, *distributed deep learning*–training deep neural networks on a cluster of GPU servers–is increasingly leveraged to train complex models on larger datasets. In particular, SGD-based optimization has emerged as the de facto way to perform distributed training and provides the basis for parallelizing training jobs, allowing deep learning practitioners to evaluate different model variants quickly.

However, it is more difficult to achieve good performance and high-quality training with SGD-based distributed training, compared to traditional single-node training [1], [2]. A large number of factors, such as slow servers and network communication links, can all impact the distributed training performance [3]–[5]. Of particular importance is how each cluster node communicates and synchronizes their respective progress during training, governed by *parameter synchronization protocols*, which has a profound impact on both the model converged accuracy and training time. Bulk synchronous parallel (BSP) [6], a default option for popular frameworks including TensorFlow, requires each node to synchronize every iteration. In contrast, asynchronous parallel (ASP) allows nodes to work at their own pace [7]. However, both distributed training protocols have their respective limitations, e.g., BSP is prone to slow down due to workers need to wait for synchronization while ASP suffers from decreased accuracy due to stale gradients.

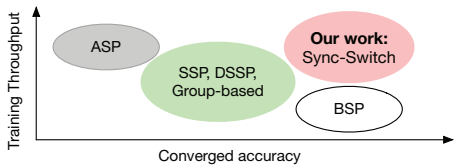
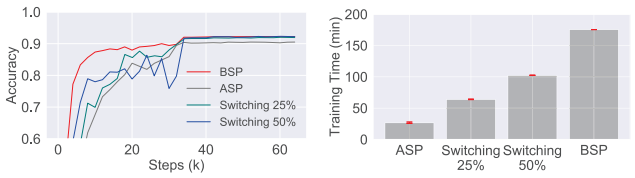


Fig. 1: **Ours vs. prior work on synchronization protocols.** Our work looks to improve both the training time and accuracy simultaneously, compared to prior work that trades-off these two metrics.

In this work, we explore ways to exploit the benefits of both BSP and ASP and design a hybrid synchronization approach *Sync-Switch*. In contrast to prior work [8]–[11], which often needs to sacrifice either training throughput or converged accuracy, we set out to reduce training time while simultaneously maintaining the converged accuracy as illustrated in Fig. 1. Specifically, we propose an empirically-driven methodology—which we also use to generate a set of policies—that determine how and when to switch the synchronization protocol. Our policies, the offline ones that target jobs under normal training circumstances and the online ones that react to cluster runtime status, are designed with the key insight of maximizing the time the GPU servers spend on training asynchronously without sacrificing the trained model's accuracy.

Through extensive empirical profiling of distributed training workload, we demonstrate that our key idea of *hybrid synchronization* can lead to converged accuracy and training time benefit compared to using BSP and using ASP, respectively. In particular, we empirically observe that, while one may need to perform synchronous training throughout all epochs to achieve the highest possible *training* accuracy, only a minimal amount of synchronous training is needed to achieve a high-quality *test* accuracy. Indeed, as shown in Figure 2(a), starting the

2575-8411/21/$31.00 ©2021 IEEE
DOI 10.1109/ICDCS51616.2021.00057



(a) Test accuracy. (b) Total training time.

Fig. 2: **Benefits of synchronization switching.** Training ResNet32 on Cifar-10 using 8 machines, with BSP then ASP, reduces the training time by up to 63.5% while achieving similar converged accuracy, compared to training with BSP.

training synchronously with BSP and then switching to an asynchronous protocol achieves almost the same converged test accuracy compared to training exclusively with BSP. Furthermore, by spending less time training with BSP (50% of training workload vs. 25%), the total training time is reduced by 37.5% (see Section VI-A for detailed methodology).

To determine *how and when to use* BSP and ASP synchronizations, we develop an empirical-driven methodology that allows us to derive policies for a given distributed training workload. Specifically, by comparing the training performance under different settings, one can derive the first *protocol policy* that describes the relative execution order of the synchronization protocols. We found that training with BSP (a more precise computation method) first and then switching to ASP (a less precise one) leads to improved training throughput and similar converged accuracy, compared to training with BSP. To determine when to use BSP and when to use ASP, we introduce both online and offline *timing policies*. Based on our empirical observation that training longer with BSP does not improve converged accuracy beyond a knee point, we use a binary search-based approach to find the optimal switch timing. Furthermore, to account for transient stragglers, we devise an online policy that works in tandem with the optimal timing policy obtained offline. Finally, to properly adjust hyper-parameters when switching to a new synchronization protocol, we devise the *configuration policies* by adapting prior work and by taking into account factors including the cluster size and the training stage.

We implement the proposed policies in a prototype system called *Sync-Switch* on top of a popular distributed training framework TensorFlow and evaluate the training performance under three distributed training setups. Our experiments show that *Sync-Switch* achieves up to 5.13X throughput speedup and similar converged accuracy when comparing to BSP. Further, we observe that *Sync-Switch* achieves 3.8% higher converged accuracy with just 1.23X the training time compared to training with ASP. Moreover, *Sync-Switch* can be used in settings when training with ASP leads to divergence errors. *Sync-Switch* is also effective in handling transient stragglers (nodes that exhibit temporary slowdown), mitigating the potential performance degradation. Furthermore, we quantify the overhead of using *Sync-Switch* in terms of offline search cost and runtime overhead. Specifically, we show that the upfront search cost can be quickly amortized for jobs with more than ten recurrences, a very likely scenario given the trial-and-error nature of deep learning training. Additionally, *Sync-Switch* incurs as low as 36 seconds, or about 1.7% of the total training time, overhead in switching between BSP and ASP protocols.

In summary, we make the following main contributions:

- We propose a methodology and derive policies that govern hybrid parameter synchronization to improve the training throughput while *simultaneously* maintaining high-quality converged accuracy. The offline policies can be derived for a given distributed training workload and are particularly useful for recurring jobs[1], while the online policies are effective in dealing with stragglers that are temporary.
- We implement a prototype system called *Sync-Switch*, on top of a popular deep learning framework TensorFlow, that encapsulates all the adaptive policies. Deep learning practitioners can directly leverage *Sync-Switch* without modifying the distributed training scripts. Our code and experiment data are available in the project GitHub repository: https://github.com/cake-lab/Sync-Switch.
- We demonstrate the efficacy of these policies through experiments using popular convolutional neural networks and datasets for image classification. We show that *Sync-Switch* can improve the total training time by up to 5X while achieving similar test accuracy, compared to training with BSP. *Sync-Switch* also achieves 4X improvement on the time-to-accuracy metric, outperforming prior work that reported a speedup of 1.1X-2X when using protocols SSP and DSSP [8]. Further, *Sync-Switch* can effectively circumvent the performance degradation caused by transient stragglers under moderate slowdown scenarios.

II. Background

A. Distributed Deep Learning

In this work, we target *distributed deep learning* where multiple GPU-equipped computing nodes work together to train a deep learning model by communicating gradients and parameters over network. We focus on the more popular approach *data parallelism*, where the training data are partitioned and offloaded to the workers, instead of model parallelism where models themselves are distributed. When training with data parallelism, each worker trains on the complete neural network in *steps*, which each step corresponds to going through one mini-batch of data, to update the model parameters.

Further, we focus on *parameter server (PS)* based architecture that consists of two logical entities: PS and worker. We choose to collocate PSs and workers, to better exploit computational resources and to reduce network communication, and configure the training cluster with equal numbers of PSs and workers based on prior work [12]. To train a neural network, a worker will first pull model parameters from all PSs, then perform the forward and backward propagation computation on a mini-batch and the current model parameters, and finally

[1]Recurring jobs refer to training jobs with the same deep learning model, but can be trained with different hyper-parameters, such as hyper-parameter tuning, or different datasets, such as online learning.

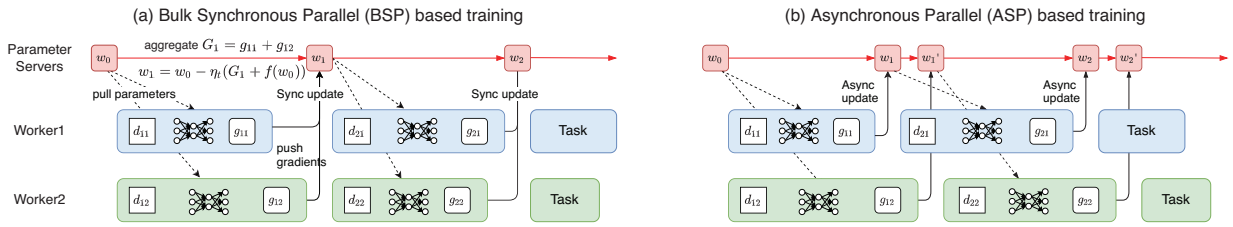


Fig. 3: **Popular distributed parameter synchronization protocols.** For BSP, parameter servers use barriers for gradient collection and only start to update parameters w once all gradients g are aggregated. In contrast, ASP workers push/pull gradients/parameters at their own pace.

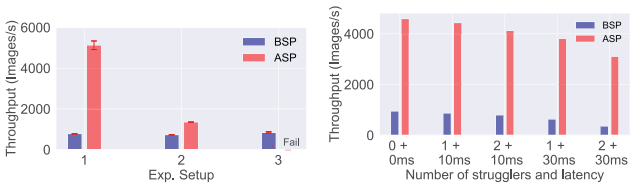
push the computed gradients to all PSs. Depending on the parameter synchronization protocol in use, the PSs will immediately update the model parameters or wait until receiving all gradients from all workers.

B. Distributed Parameter Synchronization Protocols

In distributed deep learning, synchronization and coordination of worker progress (i.e., gradient computation) is achieved by *distributed parameter synchronization protocols* [6]–[8], [11], [13]. There are two popular protocols: *Bulk Synchronous Parallel* (BSP) and *Asynchronous Parallel* (ASP) that differ in their horizontal scalability, sensitivity to stragglers, and converged test accuracy. Via an empirical measurement with different training workloads and cluster sizes, we quantify the training throughput difference between BSP and ASP as shown in Figure 4. Given the same experiment setup, training with ASP can be up to 6.59X faster than with BSP, especially when stragglers are presented. Our observations align with recent work [12], [14] and suggest the promise of leveraging ASP to improve training time.

BSP, as shown in Figure 3(a), is a deterministic scheme where workers perform a parallel computation phase followed by a synchronization phase where the gradients (e.g., g_{11} and g_{12}) are aggregated at a barrier. The model parameters are then updated according to this aggregated set of gradients. This method ensures that all workers work on the same model parameters and prevents any workers from proceeding to the next mini-batch. This synchronous update guarantees the method to be equivalent to a true mini-batch stochastic gradient descent algorithm, which can ensure the correctness of the parallel computation [15]. Since BSP workers need to wait for all updates to the model parameters at the barrier, the faster workers all remain idle while waiting for the stragglers. This drastically limits the training performance of the whole training cluster. Generally speaking, BSP offers high converged accuracy but suffers from computation inefficiency, especially in unfavorable environments.

ASP, as shown in Figure 3(b), allows computations to execute in parallel as fast as possible by running workers completely asynchronously. Each worker individually pulls parameters and pushes new gradients for updates at the parameter server. For example, once Worker1 pushes the computed gradient g_{11} to the PSs, w_0 is updated as $w_1 = w_0 - \eta_t(g_{11} + f(w_0))$ and used by the subsequent task in Worker1. Similarly, the PSs will update w_1 as $w_1' = w_1 - \eta_t(g_{12} + f(w_1))$ when receiving



(a) Without straggler. (b) Exp. Setup 1: with straggler.

Fig. 4: **Training throughput comparison between BSP and ASP.** We observe that training with ASP has higher throughput than with BSP, even without stragglers. See Section VI-A for a detailed methodology.

g_{12} from Worker2. Consequently, ASP can often achieve better speedups than BSP. However, ASP can be impacted by the stale gradients: the tasks may use old model parameters (e.g., the second tasks of Worker1 and Worker2 use different model parameters w_1 and w_1') for training, which introduces noise and error into the computation. As such, the model trained with ASP often converges to lower training and test accuracy when compared to BSP [11], [15].

III. Problem Statement and Solution Overview

In this paper, we investigate *how to improve the training throughput while simultaneously maintaining the converged accuracy* for distributed deep learning. Our study is motivated by the inherent limitations of existing synchronization protocols, as previously discussed in Section II. Our key insight is that by adaptively switching between the synchronization protocols based on both internal training status and runtime factors, we can avoid their respective drawbacks such as low speedup and low converged accuracy as much as possible.

System model. We consider the scenario of training a deep learning model in a dedicated cloud-based GPU cluster. We focus on the popular parameter-server-based distributed training architecture adopted by TensorFlow and prior work [5], [12], and collocate the PS and worker on the same physical server. Further, we target the training of deep convolutional neural networks on widely used image datasets with data parallelism, a commonly used approach for models that can fit into the memory of a discrete GPU. We assume deep learning practitioners will provide a training script that specifies the initial training configuration, describing the training cluster and the deep learning workload, as well as the hyper-parameters. This assumption is reasonable as distributed training is often an

iterative and recurring process, making such training scripts readily accessible.

Challenges. There are three key challenges in designing policies for hybrid synchronization. First, given that the training itself is a stochastic process, it is challenging to leverage internal training metrics such as training loss and anytime accuracy to extrapolate and generalize observed performance benefits. Second, we have observed high accuracy variations, even using the same training workload and cluster setup (e.g., Figure 5(a)). As such, it necessitates the consideration of this inherent performance fluctuation when designing any policies. Additionally, it also makes designing empirical-based policies costly at the very least because each configuration needs to be evaluated multiple times. Third, distributed training can be prone to runtime performance variations such as network bandwidth fluctuations, and if left undealt with, the stragglers can lead to degraded training performance (e.g., Figure 4).

***Sync-Switch* Overview.** We address the above-mentioned challenges of hybrid synchronization with an guided empirical exploration and introduce a new prototype system called *Sync-Switch*. Specifically, we devise a set of policies that regulate the protocol, the timing, and the configuration to use for distributed training. Our policies include the offline ones that are generated by a binary search-based algorithm and the online ones for handling transient stragglers with temporary slowdown[2]. These policies can be used in tandem to improve the training throughput while *simultaneously* maintaining high-quality converged accuracy for distributed training jobs, as we will demonstrate empirically in Section VI. To support adaptive synchronization, *Sync-Switch* is built upon the existing framework functionalities such as saving the training progress and restarting from the checkpoint as well as mechanisms to monitor and collect internal training metrics. More details on implementation will be described in Section V.

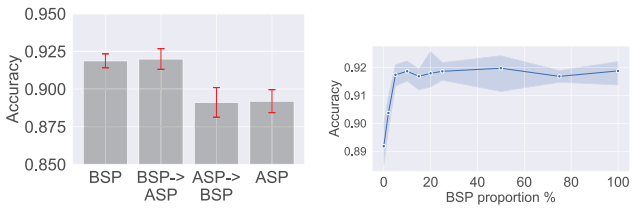
IV. *Sync-Switch* Policy Design

In this section, we introduce a set of policies that determine *how and when to switch to a different synchronization*. This includes offline policies that target recurring jobs (Sections IV-A and IV-B1), online policies that react to training status (Section IV-B2), and policies for adjusting hyper-parameters.

A. Protocol Policy: Which to Use and in What Order?

In this work, we focus on selecting between two synchronization protocols, i.e., fully synchronous (BSP) and fully asynchronous (ASP). As described previously in Section I, there are a plethora of other protocols that provide different trade-offs between training throughput and converged accuracy. Mixing in these in-between training protocols might produce a slightly better outcome due to larger decision spaces; however, these in-between protocols can be complicated to use due to lack of framework support and the use of extra hyper-parameters [9]–[11].

[2]This differs from a longer-term slowdown, which can be most effectively handled by replacing the slow worker [14].



(a) Order of synchronicity. (b) Percentage of synchronicity.

Fig. 5: **Impact of synchronicity.** We observe that *(i)* switching from BSP to ASP can maintain the same level of converged accuracy and *(ii)* training longer with BSP does not further improve accuracy beyond the knee point.

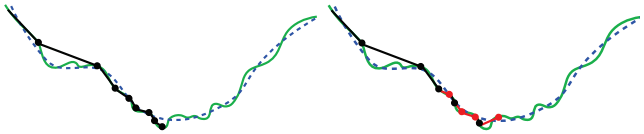
Our first policy states that we should start the training with BSP and then switch to ASP when the following conditions are satisfied: *(i)* When switching to ASP yields a similar converged *test* accuracy (but perhaps a lower *training* accuracy) as continuing with BSP; *(ii)* when transient stragglers arise in the training cluster. We will describe the policies that verify both conditions to determine the switching timing in Section IV-B.

1) Empirical Analysis: To evaluate our hypothesis that running BSP at earlier epochs and switching to ASP at later epochs allows one to obtain a test error which is as low as using BSP during the entire training, we conducted two experiments. The experiments train the ResNet32 model on the CIFAR-10 dataset with an 8-worker cluster, with each configuration repeated five times (additional experiment details can be found in Section VI-A). First, we evaluate the converged accuracy when training with different combinations of BSP and ASP protocols, as shown in Figure 5(a). We observe that training with BSP for 50% of the workload and then switching to ASP outperformed its counterpart, i.e., ASP $\rightarrow$ BSP. Second, we examine the relationship between converged accuracy and the percentage of BSP training and find that training more with BSP does not necessarily lead to higher converged accuracy. As shown in Figure 5(b), the converged accuracy first increases with the percentage of BSP training and then stays on par with training entirely with BSP (i.e., 100%). This observation also provides the basis for deriving the timing policy as described in Section IV-B.

2) Theoretical Explanations: Next we explain why using BSP in the earlier epochs and then switching to ASP can allow us to simultaneously accomplish both our goals of minimizing the *population loss* (that is, the expected value of the testing loss) as effectively as when using *static BSP*, and improving the per-epoch computation time. Here, by static BSP (respectively, ASP), we mean the protocol where one uses only BSP (respectively, ASP) from start to finish.

First, we note that the steps taken by SGD earlier in the training are much larger than those taken later in the training. This is because, *(i)* at earlier epochs the gradient tends to have a much larger magnitude, and *(ii)* we use a decaying learning rate schedule, where the learning rate is much larger at earlier epochs than at later epochs, a common practice when training deep learning models [16], [17].

Second, we note that at coarser scales corresponding to



(a) Static BSP. (b) Switching, BSP to ASP.

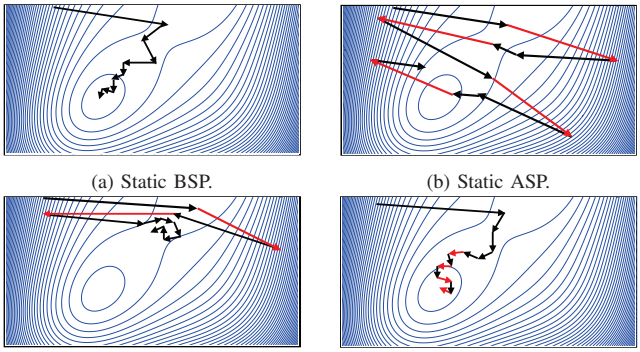
Fig. 6: The training loss (green solid curve) is not as smooth as the population loss (blue dashed curve). Thus, the steps taken by *Sync-Switch* (b) with stale gradients (red lines) prevent it from finding the minimum of the training loss as effectively as static BSP (a). However, it is still able to find a point which minimizes the *population* loss (and hence the *test* loss) as effectively as static BSP.

the large steps taken by the algorithm earlier in the training, the landscape of the *population* loss resembles that of the training loss (see Remark A.1 in the extended version for more details [18]). On the other hand, past empirical work [19]–[21] suggests that at finer scales corresponding to the smaller steps taken later in the training, the landscape of the population loss is much smoother than that of the training loss.

These two facts imply that while stale gradients may be ineffective very early in the training when larger steps are taken by the algorithm, stale gradients can be used to effectively minimize the population loss later in the training, since the landscape of the population loss is much smoother on a finer scale corresponding to the smaller steps taken by the algorithm later in the training (Figure 6). This is because, at later epochs, the gradient of the population loss does not change as quickly each time the algorithm takes a step, which means that stale gradients may still be able to provide useful information about the gradient of the population loss at the current point even though the stale gradients were computed at previous (but nearby) points. This suggests that using ASP at later epochs (Figure 6(b)) can allow one to minimize the *population* loss (and hence the testing loss) as effectively as static BSP (Figure 6(a)), despite the fact that static BSP can achieve a lower *training* loss value (see Remark A.2 in our extended version for more details [18]).

On the other hand, at very early epochs, when the algorithm is very far from a minimum point, the gradient of the loss function may have a very large magnitude and may change very quickly at each step [3] Thus, in protocol policies where ASP is used early in the training, its stale gradients may not provide much information about the value of the gradient at the current point, and may cause ASP to exhibit unstable behavior early in the training (Figures 7(b) and 7(c)), preventing ASP from effectively decreasing the loss value. To avoid this unstable behavior, one should use BSP in the very early stages of the training, and only switch to ASP at a later epoch once the learning rate is lower and the gradients are changing more slowly (Figure 7(d)) (See Remark A.3 in our extended version for additional discussions [18].)

[3]For instance, consider the simple 4'th-order polynomial $f(x) = x^4$. For this function, the gradient $f'(x) = x^3$ changes more quickly when x is further from the minimum point $x = 0$. Roughly speaking, for loss functions defined by deep neural networks there can be many (multivariate) high-order terms, with the order of the terms growing with the depth of the network.



(a) Static BSP. (b) Static ASP.

(c) ASP to BSP. (d) BSP to ASP.

Fig. 7: **Illustrations of various combinations of BSP and ASP on a simple smooth loss function (blue level sets).** In earlier epochs, the gradient changes quickly at each step and stale gradients (red) are much less reliable than non-stale gradients (black). Thus algorithms which use ASP earlier in the training may have a difficult time settling into a local minimum region where the gradient is small (b, c). This is true even if one switches from ASP to BSP (c) as the algorithm may get stuck in a saddle point once the learning rate has decayed, causing it to take a long time to escape the saddle point. In contrast, if one only uses stale gradients at later epochs once the algorithm is closer to a minimum point, the stale gradients will be low-bias estimates for the true gradient and will still allow the algorithm to reach the minimum point (d).

B. Timing Policy: When to Switch?

Next, we introduce *Sync-Switch*'s timing policy which determines when to switch between BSP and ASP. Deciding the proper timing to switch is an important problem as it not only impacts the converged accuracy but also the training time. Furthermore, it is a challenging problem as both model-specific factors such as training progress and runtime factors such as slow nodes can impact the timing. Specifically, we develop both offline and online policies that are suitable to use under different training conditions.

1) Offline Policy via Binary Search: Based on our empirical observation that BSP is strictly slower than ASP, with or without stragglers, and that the converge accuracy increases monotonically with the amount of BSP training, we formulate the searching process as a binary search problem. Specifically, for a given training workload, our goal is to find a switching point s that yields a converged accuracy $\alpha(s)$ that satisfies $\alpha(s_{min}) \leq \alpha(s) \pm \alpha_{threshold} \leq \alpha(s_{max})$, where s_{min} and s_{max} represents training with ASP and BSP, respectively. Further, the corresponding training time $T(s)$ should be as close to $T(s_{min})$ as possible, i.e., switching as early from BSP to ASP as possible. Lastly, we want to find s in as fewer trial training sessions as possible to reduce search overhead.

The pseudo-code for our binary search algorithm can be found in our extended technical report [18]. In summary, for each distributed training workload, we can use a binary search algorithm to find the best switching point s at which point *Sync-Switch* will switch from training with BSP to ASP. We analyze the cost and performance tradeoff in Section VI-C1.

2) Online Policies for Handling Stragglers: The offline policy described above provides us a good basis for speeding up distributed training while achieving high-quality test accuracy. However, it does not account for runtime factors including stragglers and also requires upfront search cost. In this section, we introduce two types of policies that are designed to react to the training status.

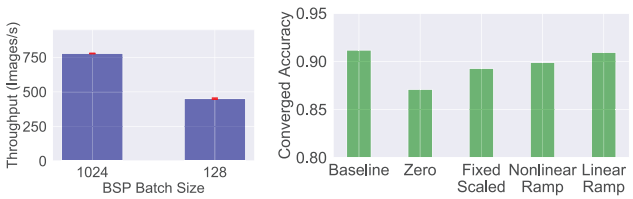
We target transient stragglers, e.g., nodes that exhibit temporary slowdown due to datacenter network or server resource contention, a non-rare occurrence when using public cloud [22], [23]. Note that permanent stragglers are best dealt with by requesting replacement [12], [14]—simply switching to a more straggler-tolerant protocol like ASP might mitigate but not eradicate the performance impact. More explicitly, we consider the policies that deal with permanent stragglers as complementary work and therefore skip the discussion and evaluation of such policies. For ease of exposition, we assume that each occurrence of the slowness lasts at most the time to provision a new cloud server—we use 100 seconds based on empirical measurement reported by prior work [5]. We further assume that *(i)* each node can become a straggler at any time during the training; *(ii)* the number of unique straggler nodes is less than the cluster size. Our goal is to design policies that adequately deal with the potential impact on the training time which also work *in tandem* with the other policies of *Sync-Switch* described above.

We introduce two policies, both centering on the key insight that any transient straggler-oriented policies only need to react before the switch timing for a given workload. This is because once a training session is switched to ASP, we consider it immune from the impact of transient stragglers. The first *greedy* policy simply switches to ASP (if it is not already using ASP, which can happen if two stragglers overlap) when a straggler is detected; once the cluster is free of any stragglers and the aggregate BSP training has not been satisfied, it will switch back to training with BSP. This policy therefore might result in multiple synchronization switches, with overhead in the order of tens seconds as we will show in Section VI-C2.

To circumvent the switching overhead, we design an elastic-based policy that removes any detected stragglers from the current cluster so as to complete the specified amount of BSP training free of stragglers. Once the designated BSP workload is fulfilled, it will then restore the cluster size and train the remaining workload with ASP. As such, this elastic-based policy is more resistant to the frequency of straggler occurrences. For both policies, we leverage the historical average training throughput to detect the stragglers, a common technique used in various application domains [5], [24]. Specifically, a worker k is identified as a straggler if its training throughput over a sliding window S_k is lower than the difference between the cluster average and standard deviation $S - \sigma$, for a number of consecutive detection windows.

C. Configuration Policy: How to Adjust Hyper-parameters?

Hyper-parameters are commonly considered important for training performance [25], [26]. In our work, we also observe non-negligible differences with different hyper-parameters *after switching* from BSP to ASP. For example, in Figure 8(a), we show that the difference in training throughput can be up to 2X with different batch sizes; in Figure 8(b), we observe up to 5% converged accuracy differences using different momentum scaling techniques. As such, it is important to choose suitable hyper-parameters after synchronization switching so as to retain the training benefits.



(a) Batch size scaling. (b) Momentum Scaling.

Fig. 8: **Comparison of different hyper-parameter configurations: Exp. Setup 1.** Please see our extended report [18] for techniques used to scale momentum.

Through empirical investigations, we find that adjusting the following two hyper-parameters: *mini-batch size* and *learning rate* sufficient. Note, since our work is not about hyper-parameter tuning, we do not focus on finding the optimal hyper-parameters set for a specific training setting; rather, we are aiming to automatically change the value of these hyper-parameters based on the synchronization protocol in use. We assume the deep learning practitioners will provide an initial set of hyper-parameters, e.g., a mini-batch size of B and a learning rate of η, given the training workload and a GPU cluster of n nodes.

To start the training with BSP, we will configure the mini-batch to be nB. This configuration is based on both the implementation of BSP batch size in the TensorFlow framework (i.e., as a global value distributed to each worker) and prior work's suggestion of setting BSP batch size proportionally to the cluster size [27]. Once switching from BSP to ASP, we will reduce the mini-batch to be B as in ASP training the batch size is treated as a local value specific to each worker. We use the *linear scaling rule* for setting the learning rate for training with BSP as $\eta_{BSP} = n\eta$. This is based on prior work that demonstrated the effectiveness of scaling learning rate based on mini-batch size [27]. In contrast to prior work that adjusts momentum based on mini-batch size [28], we find that using the *same* momentum value for both BSP and ASP allows *Sync-Switch* achieve comparable accuracy to training with BSP (i.e., leftmost bar in Figure 8(b)).

V. *Sync-Switch* Implementation

We implemented a *Sync-Switch* prototype, as shown in Figure 9, based on TensorFlow v1.10 and Tensor2Tensor v1.9 [29]. The prototype includes the parameter synchronization policies described in Section IV for distributed training jobs. *Sync-Switch* users can manage their distributed training jobs via the command line. *Sync-Switch*'s implementation consists of two logical components: a standalone cluster manager

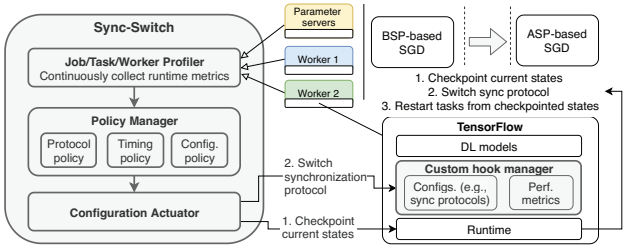


Fig. 9: ***Sync-Switch* architecture and implementation.** We implement *Sync-Switch* on top of the popular TensorFlow framework as two logical parts: a standalone entity and a per-node part. Grey-shaded components are our modifications. Hollow and solid arrows represent the profiling and actuation workflow, respectively.

that interfaces with Google Cloud Platform and a custom hook manager embedded in TensorFlow that collects training status and adjusts per-node configurations.

The *cluster manager* first takes the user input, including the training job script and cluster size, to initialize protocol and configuration policies. If the job is a recurring one, the cluster manager initializes the timing policy based on the prior binary search-based result. Otherwise, the cluster manager launches a pre-specified number of pilot jobs per the search algorithm described in Section IV-B1 to obtain the timing policy. Afterward, *Sync-Switch* creates the training cluster consisting of nodes running with TensorFlow and sets up the *profiler* for continuously collecting runtime metrics.

Sync-Switch's custom hook manager is written as a core Python component to interact with TensorFlow runtime to collect internal metrics such as training throughput and training loss and to change hyper-parameters like learning rates. The collected metrics are sent back to the profiler, which, in conjunction with the policy manager, decides whether to trigger a synchronization protocol switch. The switch mechanism is implemented by having each custom hook manager listen at a pre-specified port for incoming commands and by leveraging TensorFlow's built-in model checkpoint/restore functions for persisting the training progress. In *Sync-Switch*, once all custom hook managers finish checkpointing, the cluster manager propagates the updated training job and configurations to all nodes. Once notified, custom hook managers relaunch the training tasks to resume the training from the last model checkpoint but with a different synchronization protocol.

VI. Evaluation

We conducted our experiments by training popular deep learning models on Google Cloud Platform (GCP) to quantify *Sync-Switch*'s performance over training exclusively with BSP and with ASP, two commonly chosen baselines [9], [11]. Note, since the performance of existing synchronization protocols all fall in between that of BSP and ASP, we believe evaluating using BSP and ASP provide us a good foundation for understanding *Sync-Switch*'s performance. Furthermore, semi-synchronous protocols, such as SSP and DSSP, can also be utilized in *Sync-Switch* (for example switching from SSP to ASP)—*Sync-Switch* is agnostic to the underlying synchronization protocols. Our evaluation includes systems experiments using our *Sync-Switch* prototype to evaluate the efficacy of timing policies and framework overhead with real TensorFlow jobs, as well as simulation experiments to analyze the performance and cost of our binary search-based algorithm under realistic workload conditions. Table I summarizes our experiment setups and result highlights.

A. Evaluation Setup and Methodology

Distributed Training Workloads. We choose two different workloads, *(i)* ResNet50 on the CIFAR-100 dataset and *(ii)* ResNet32 on the CIFAR-10 dataset. Both models are part of the ResNet model family, one of the widely used CNNs for image recognition tasks. We use the ResNet implementations from Tensor2Tensor [29]. ResNet50 has more layers than ResNet32, leading to different model parameter size and floating-point operations, and therefore has longer per-batch training time with the same cluster. The datasets, each containing 60K images of size 32X32 pixels, are widely used in deep learning research [30]. The key difference between the two datasets is the number of the classification classes (i.e., CIFAR-100 contains 100 classes vs. CIFAR-10 contains 10 classes); therefore, it often takes more epochs to train on the CIFAR-100. As such, these two workloads allow us to evaluate *Sync-Switch*'s performance under different computation and learning requirements.

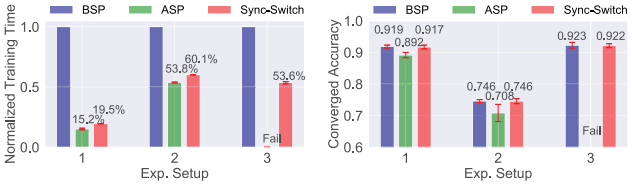
Cluster Setup and Configuration. We run all experiments on cloud-based GPU clusters in GCP's *us-west1* region; we choose two commonly used cluster sizes of eight and sixteen[4] to evaluate *Sync-Switch*'s performance [11], [14]. Each server, running Ubuntu 18.04 LTS, has 8 vCPUs, 30 GB of main memory, 100GB local HDD storage, and is equipped with one Nvidia K80 GPU card. To account for the inherent accuracy variations in SGD-based deep learning training, we repeat each experiment setup five time using the same model parameter initialization algorithm. We report both the average performance with standard deviation and the runs with the best performance, measured by the highest achieved test accuracy.

Evaluation Metrics. We use two groups of metrics for evaluating *Sync-Switch*'s efficiency in parameter synchronization (first group) and its associated overhead (second group). The first group includes training loss, test accuracy, total training time, and time-to-accuracy. *Training loss* is calculated based on the cross-entropy loss function per mini-batch. We report the average training loss collected every 100 ASP steps to avoid incurring excessive measurement overhead [5]. *Test accuracy* refers to the top-1 accuracy of the trained model when evaluating on the test dataset. We measure the test accuracy every 2000 ASP steps on the standalone cluster manager to avoid impacting the training performance. A model is said to be converged if its test accuracy has not changed for more than 0.1% for five evaluations and we report the corresponding value as the *converged accuracy*. *Total training time* is the

[4]Smaller cluster size has less impact on ASP's converged accuracy [5].

Experiment Setup	Workload (model, dataset)	Cluster (size, GPU)	*Sync-Switch* policy (protocol, timing)	Throughput Speedup vs. ASP	Throughput Speedup vs. BSP	TTA Speedup vs. ASP	TTA Speedup vs. BSP
1	ResNet32, CIFAR-10	8, K80	P1: ([BSP, ASP], 6.25%)	0.78X	5.13X	N/A	3.99X
2	ResNet50, CIFAR-100	8, K80	P2: ([BSP, ASP], 12.5%)	0.89X	1.66X	N/A	1.60X
3	ResNet32, CIFAR-10	16, K80	P3: ([BSP, ASP], 50%)	failed	1.87X	N/A	1.08X

TABLE I: **Summary of our experiment setups, timing policies, and performance.** We observe that *Sync-Switch* achieves up to 5X (4X) speedup in training throughput (TTA). P_i represents the set of policies for setup i. Note ASP-based training failed in exp. setup 3.



(a) Total training time. (b) Converged accuracy.

Fig. 10: **End-to-end performance comparison.** Using *Sync-Switch*, we observe on average 1.66X-5.13X speedup and similar converged accuracy compared to training with BSP. *Sync-Switch* achieved up to 3.8% higher converged accuracy compared to training with ASP.

time, including computation and networking time, taken for a training cluster to complete a user-specified workload. We measure this time at the end of each training using the TensorFlow built-in logs. *Time-to-accuracy (TTA)* denotes the time to reach a specified test accuracy threshold and provides valuable insights into both the training throughput and model accuracy [31]. We use the average converged accuracy of models trained with BSP in the same setup as the threshold. For the second group, *search time* quantifies the time for *Sync-Switch* to find the near-optimal switch timing using the binary search. We calculate search time as the sum of the total training time of all sessions trained during the search. *Switch overhead* describes the total time to switch between synchronization protocols with TensorFlow.

Hyper-parameter Setting. We set the initial hyper-parameters based on the original ResNet paper [16] and use the SGD with momentum of 0.9 as the optimizer. Specifically, we configure the training workload to be 64K steps, batch size to be 128, and learning rate to be 0.1. We use a piece-wise function that decays learning rate at 32K and 48K steps, with scaling factors of 0.1 and 0.01, respectively. Further, we use the configuration policies described in Section IV-C to adjust relevant hyper-parameters based on the specific experiment setup.

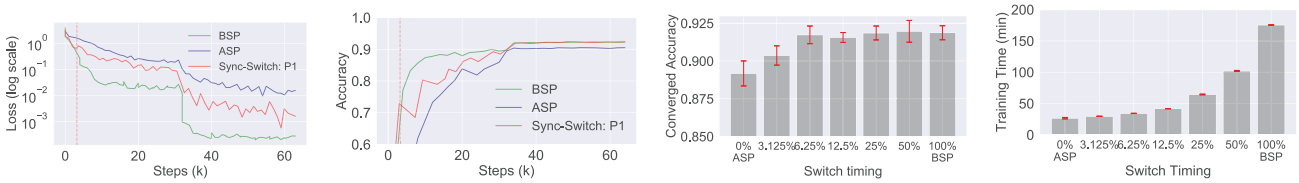
B. *Performance of* Sync-Switch

1) End-to-end Comparison: Table I and Figure 10 summarize the end-to-end training performance achieved by *Sync-Switch*. In all setups evaluated, *Sync-Switch* outperforms training exclusively with BSP and with ASP in total training time, TTA, and converged accuracy, respectively. For example, we observe that *Sync-Switch* uses only 19.5% of the training time while reaching similar converged test accuracy when compared to BSP; additionally, *Sync-Switch* improves converged accuracy by 2.5% with only 1.28X the training time when compared to ASP (exp. setup 1). The speedups achieved by *Sync-Switch* are more prominent compared to 1.2-3.7X reported in prior work [9]; the high-quality converged accuracy is significant as recent innovations on models have similar levels of improvement, e.g., less than 2% [32], [33]. The training speedup comes from *Sync-Switch* only needs to train a small portion of workload using BSP and the competitive converged accuracy comes from identifying the most appropriate switch timing.

Furthermore, Figure 11 details the performance of training a ResNet32 on the CIFAR-10 dataset using an 8-GPU cluster. We plot training loss and test accuracy of the best runs for ASP, BSP, and *Sync-Switch* in Figures 11(a) and 11(b). Interestingly, training with BSP at the first 6.25% steps allows *Sync-Switch* to decrease the training loss faster and to lower values (the lower the better), compared to training exclusively with ASP. Additionally, even though *Sync-Switch* does not decrease training loss to the same level as BSP, it still reaches the same converged accuracy. More importantly, *Sync-Switch* only needs 25% training time used by BSP to reach the same converged accuracy (i.e., a 4X TTA speedup). Further, Figures 11(c) and 11(d) draw the comparison to manual switching, i.e., switching at a static time point, and demonstrate *Sync-Switch*'s utility.

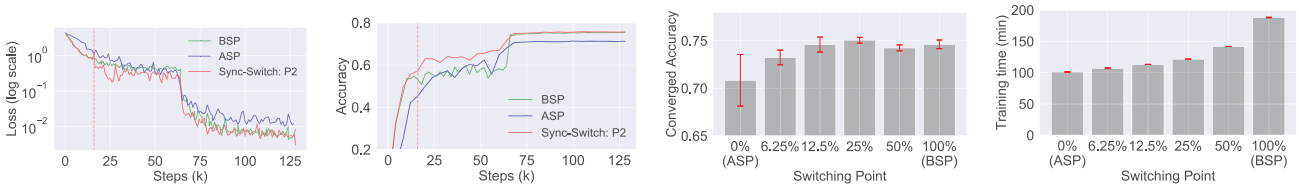
Results of similar magnitude are also observed for the other two setups as shown in Figure 12 and Figure 13. Note that training ResNet32 with ASP (and training with BSP for less than 50% steps) in a cluster of 16 GPU servers resulted in failed training due to the training loss divergence. The failed training sessions are likely caused by noisy gradients that are exacerbated with larger cluster sizes. This observation attests to an additional benefit provided by *Sync-Switch*—being able to complete training in scenarios where ASP cannot.

2) Generality Analysis of Our Observations: Figure 14 compares the performance of directly using policies found for the other two setups to the third setup. We make the following key observations. First, for similar workloads (i.e., training ResNet32 on Cifar-10 vs. ResNet50 on Cifar-100), the cluster size seems to play an important role in determining the efficiency of the policies. Concretely, training with policy 2 in the experiment setup 1 achieves almost the same converge accuracy (91.7% vs. 91.4%) and uses about 33.0% longer training time. The prolonged training time is expected as policy 2 requires more training to be done with BSP. In comparison, training with policy 3 under the same setup uses 3X of the training time with policy 1. Furthermore, by using the correct policy under experiment setup 3, the training successfully finished (without diverging) and achieved comparable test accuracy to using BSP while saving almost 46.4% training time. As shown in Figure 13, training with



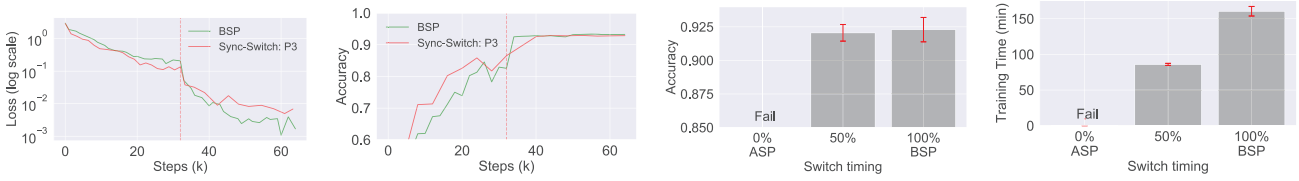
(a) Training loss. (b) Test accuracy. (c) Converged accuracy. (d) Total training time.

Fig. 11: **Performance of exp. setup 1.** Using the policy of switching to ASP after completing 6.25% workload with BSP, *Sync-Switch* achieves similar converged accuracy and 80.5% training time saving compared to training with BSP. Between the range of 6.25% to 50%, switch timing has minimal impact on the converged accuracy but noticeable impact on training time. Dashed line marks the switch timing.



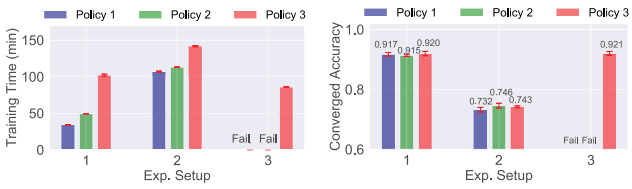
(a) Training loss. (b) Test accuracy. (c) Converged accuracy. (d) Total training time.

Fig. 12: **Performance of exp. setup 2.** With the policy of switching to ASP after completing 12.5% steps using BSP, *Sync-Switch* achieves similar converged accuracy with 39.9% training time saving, compared to training with BSP.



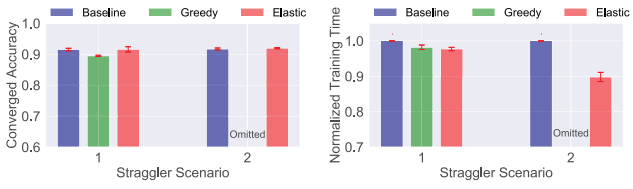
(a) Training loss. (b) Test accuracy. (c) Converged accuracy. (d) Total training time.

Fig. 13: **Performance of exp. setup 3.** Due to the highly unstable convergence of stale gradients, ASP and switching before 50% (first learning rate decay) diverged, leading to failed training.



(a) Total training time. (b) Converged test accuracy.

Fig. 14: **Cross examination of *Sync-Switch* policies in different experiment setups.** Policy i represents the set of policies found by *Sync-Switch* for experiment setup i (as summarized in Table I).



(a) Converged accuracy. (b) Normalized training time.

Fig. 15: **Comparison of *Sync-Switch*'s straggler-aware policies.** We observe that the elastic-based policy preserves the converged test accuracy and has a 1.1X speedup compared to the baseline policy.

ASP failed without producing usable models. This observation further suggests the practical utility of *Sync-Switch*.

3) With Transient Stragglers: We construct two transient straggler scenarios, mild and moderate, for experiment setup 1 to evaluate the effectiveness of the straggler-aware policies introduced in Section IV-B2. In the first scenario, we set the number of stragglers to be 1, the frequency of straggler occurrences to be 1, and inject the slowness by emulating the network latency to be 10ms. In the second scenario, we increase the number of stragglers, the frequency of straggler occurrences, and the degree of slowness to be 2, 4, and emulated with 30ms network latency.

Figure 15 compares the training performance with and without (baseline) applying our straggler-aware policies. We observe that when the straggler scenario is mild (scenario one), both straggler-aware policies adequately handle the potential performance degradation and even shorten the total training time by 2%, compared to the straggler-agnostic baseline policy. Furthermore, we find that the greedy policy leads to a 2% lower converge accuracy while policy two is able to maintain

Search Setting	Cost	Amortization	Effective (vs. BSP)	Success Probability
(Exp.1, No, 5, 5)	12.71X	15.79	1.97X	100%
(Exp.1, No, 3, 3)	7.62X	9.47	1.97X	99.2%
(Exp.1, Yes, 0, 3)	4.63X	5.75	2.59X	100%
(Exp.2, No, 5, 5)	17.86X	44.81	1.12X	100%
(Exp.2, No, 4, 4)	14.28X	35.83	1.12X	93.4%
(Exp.2, Yes, 0, 4)	9.05X	22.71	1.17X	100%
(Exp.3, No, 5, 5)	7.68X	16.54	1.30X	100%
(Exp.3, No, 3, 3)	4.61X	9.93	1.30X	100%
(Exp.3, Yes, 0, 1)	0.54X	1.16	1.87X	100%

TABLE II: **Binary search cost analysis.** We define a search setting as job recurrence, number of BSP trainings, and number of candidate policy trainings.

the high-quality converge accuracy. The accuracy degradation is most likely due to having to perform two extra switches, one to ASP and the other back to BSP, before the optimal timing. Based on our empirical observation, we conclude that the greedy policy does not work in conjunction with *Sync-Switch*'s existing baseline policies.

In contrast, the elastic-based policy is proven to be effective even under moderate levels of slowness. In particular, we observe that this policy not only achieves a similar level of converged accuracy but also leads to a 1.11X speedup. This further suggests that the better course of action is to train without the transient stragglers than to block the remaining cluster nodes from making progress when training BSP.

C. *Overhead of* Sync-Switch

1) Binary Search Cost: To quantify the overhead of our binary search-based algorithm (in search time) in different training scenarios, i.e., with recurring jobs and fewer measurement runs, we use all our training logs and simulate each search setting 1000 times with the accuracy threshold of 0.01. Table II details the cost-performance trade-off (more results are available in our extended report [18]). If the job is recurring, the search cost can be reduced by up to 5X the BSP training. However, when facing a new training job, it is best to at least repeat the BSP runs 3 times. We further observe that with too few BSP training, the search setting often ends up with significantly lower success probability (e.g., 56.8% to 82.3%) in finding the same switching timing as the baseline setting.

Additionally, we analyze the amortized cost, measured by the number of job recurrences, and the effective training ratio, measured by the multiples of BSP training sessions. The former provides further justification of the cost-effectiveness of our binary search algorithm and the latter signifies the potential information gains. As an example of the search setting (No, 3, 3) in Table II, if a job needs to be trained for more than 9 times, a very likely event given the trial-and-error nature of deep learning training, the corresponding search cost is then amortized. Moreover, the search process itself also produces almost 2X valid training sessions compared to training with BSP. In summary, our analysis shows that the search cost is reasonably low and can be further reduced by continuously using *Sync-Switch*.

Cluster	Actuator Exec.	Init. (s)	Switching (s)	Total (s)
8 K80	Sequential	157	90	247
	Parallel (Ours)	90	36	126
16 K80	Sequential	268	165	433
	Parallel (Ours)	128	53	181

TABLE III: ***Sync-Switch* Overhead.** We measure both the initialization time and the switching overhead.

2) Runtime Overhead: We quantify the overhead of using *Sync-Switch* to perform distributed training. Our measurements are based on training ResNet32 on the CIFAR-10 dataset, as the training framework overhead is largely workload-independent [5]. Table III shows the total time spent by *Sync-Switch* for initializing the cluster and switching to a different synchronization protocol. First, initializing a cluster of twice the workers takes 1.42X the time of initializing a cluster of 8 workers. Note that one can expect to have similar initialization time even with just the vanilla TensorFlow [5]. Second, by having a configuration actuator that propagates distributed training tasks in parallel, *Sync-Switch* reduces both the initialization time and switching overhead by 2X and 3.1X, respectively. In summary, *Sync-Switch* incurs low switching overhead that increases sub-linearly with the cluster size.

VII. Related Work

Distributed Synchronization Protocols. Researchers have designed many synchronization protocols that can be roughly categorized as BSP [6], ASP [7], and semi-synchronous protocols [8], [34], that trade-off the training throughput and accuracy of distributed DL training. These studies all focus on improving the synchronization protocols for distributed SGD, by exposing mechanisms and policies to control the model staleness. In contrast, our work focuses on determining the best way to utilize existing protocols and can be used in conjunction with new synchronization protocols. Compared to semi-synchronous protocols such as SSP and DSSP, our work leads to good converged accuracy and does not require users to tune extra hyper-parameters. In addition to directly modify the synchronization protocols, researchers also look at using different synchronizations for different cluster nodes to account for the heterogeneous performance caused by network and GPU servers [9], [10]. For example, Gaia used synchronous training for nodes running inside the same datacenter and fully asynchronous training for inter-datacenter communication [10]. Additionally, Dutta et al. [11] introduced a number of SGD variants where the synchronization degree is controlled by a new hyper-parameter. Our work is similar in that we will also use different synchronizations for a given training session but with the key difference of deriving the policies for hybrid synchronization.

Network Optimization for Distributed Training. The iterative process of deep learning makes network an important bottleneck as not only the model parameters but also the gradients need to be transferred periodically. To combat the

impact on training performance without impacting the model quality, prior work explored various techniques that aim to reduce the communication costs via gradient sparsification or compression. For example, by only sending the large gradients, Aji et al. used a heuristic sparsification scheme and showed a speed gain of 22% [35]. Terngrad and QSGD improved the network communication efficiency by reducing the gradients to a few numerical levels [36], [37]. These efforts are orthogonal to our work but might be combined with *Sync-Switch* to achieve further training speedup.

VIII. Conclusion

Using the right distributed synchronization protocol at the right time can significantly improve the training throughput and produce models with good test accuracy. In this paper, we devised the first set of adaptive policies, including offline and online ones, that make such decisions and evaluated their effectiveness in a prototype system called *Sync-Switch* in Google Cloud. We found that training with BSP for only a small portion of time and then switching to ASP delivers models of comparable converged accuracy using much shorter time, compared to training with BSP. For recurring training jobs, a prevalent scenario in deep learning due to its trial-and-error nature, we used an offline approach to find the optimal switch timing. To ensure that switching from BSP to ASP does not lead to undesirable side effects, we additionally specified a configuration policy that describes how to adjust critical hyper-parameters. We showed that *Sync-Switch* improved the total training time and the time-to-accuracy by up to 5X and 4X while achieving similar test accuracy through real-world experiments, compared to training exclusively with BSP. Further, with the elastic-based policy, *Sync-Switch* can effectively circumvent the performance degradation caused by transient stragglers and instead leads to a 1.1X speedup under moderate slowdown scenarios. Additionally, the benefits brought by *Sync-Switch* come with reasonably low overhead, e.g., a search overhead that can be amortized with jobs recurring a few times and a switching overhead in the order of tens seconds.

IX. Acknowledgements

This work is supported in part by National Science Foundation grants #1755659 and #1815619, National Natural Science Foundation of China (61802377) and Youth Innovation Promotion Association at CAS, and Google Cloud Platform Research credits,

References

[1] S. Shi *et al.*, "Performance modeling and evaluation of distributed deep learning frameworks on gpus," *DASC/PiCom/DataCom/CyberSciTech*, 2018.
[2] S. Li *et al.*, "Speeding up Deep Learning with Transient Servers," *ICAC*, 2019.
[3] F. Yan *et al.*, "Performance Modeling and Scalability Optimization of Distributed Deep Learning Systems," *SIGKDD*, 2015.
[4] T. Ben-Nun *et al.*, "Demystifying Parallel and Distributed Deep Learning: An In-depth Concurrency Analysis," *ACM Comput. Surv.*, 2019.
[5] S. Li *et al.*, "Characterizing and Modeling Distributed Training with Transient Cloud GPU Servers," *ICDCS*, 2020.
[6] A. V. Gerbessiotis *et al.*, "Direct Bulk-Synchronous Parallel Algorithms," *J. Parallel Distrib. Comput.*, 1994.
[7] J. Dean *et al.*, "Large scale distributed deep networks," *NeurIPS*, 2012.
[8] X. Zhao *et al.*, "Dynamic stale synchronous parallel distributed training for deep learning," *ICDCS*, 2019.
[9] W. Jiang *et al.*, "A novel stochastic gradient descent algorithm based on grouping over heterogeneous cluster systems for distributed deep learning," *CCGRID*, 2019.
[10] K. Hsieh *et al.*, "Gaia: Geo-distributed machine learning approaching lan speeds," *NSDI*, 2017.
[11] S. Dutta *et al.*, "Slow and stale gradients can win the race," *arXiv preprint arXiv:2003.10579*, 2020.
[12] Y. Peng *et al.*, "Optimus: An Efficient Dynamic Resource Scheduler for Deep Learning Clusters," *EuroSys*, 2018.
[13] B. Recht *et al.*, "Hogwild: A Lock-Free Approach to Parallelizing Stochastic Gradient Descent," *NeurIPS*, 2011.
[14] A. Or *et al.*, "Resource Elasticity in Distributed Deep Learning," *Proceedings of Machine Learning and Systems*, 2020.
[15] J. Chen *et al.*, "Revisiting distributed synchronous sgd," *arXiv preprint arXiv:1604.00981*, 2016.
[16] K. He *et al.*, "Deep residual learning for image recognition," *CVPR*, 2016.
[17] G. Huang *et al.*, "Densely connected convolutional networks," *CVPR*, 2017.
[18] S. Li *et al.*, "Sync-switch: Extended report," *arXiv preprint arXiv:2104.08364*, 2021.
[19] B. Kleinberg *et al.*, "An alternative view: When does sgd escape local minima?" *ICML*, 2018.
[20] S. Hochreiter *et al.*, "Simplifying neural nets by discovering flat minima," in *NeurIPS*, 1995.
[21] N. S. Keskar *et al.*, "On large-batch training for deep learning: Generalization gap and sharp minima," *ICLR*, 2017.
[22] Q. Duan, "Cloud service performance evaluation: status, challenges, and opportunities–a survey from the system modeling perspective," *Digital Communications and Networks*, 2017.
[23] A. Li *et al.*, "CloudCmp: comparing public cloud providers," *ACM IMC*, 2010.
[24] J. Xie *et al.*, "Improving mapreduce performance through data placement in heterogeneous hadoop clusters," *IPDPSW*, 2010.
[25] A. Senior *et al.*, "An empirical study of learning rates in deep neural networks for speech recognition," *ICASSP*, 2013.
[26] S. L. Smith *et al.*, "Don't decay the learning rate, increase the batch size," *arXiv preprint arXiv:1711.00489*, 2017.
[27] P. Goyal *et al.*, "Accurate, large minibatch sgd: Training imagenet in 1 hour," *arXiv preprint arXiv:1706.02677*, 2017.
[28] H. Lin *et al.*, "Dynamic mini-batch sgd for elastic distributed training: learning in the limbo of resources," *arXiv preprint arXiv:1904.12043*, 2019.
[29] A. Vaswani *et al.*, "Tensor2tensor for neural machine translation," *CoRR*, 2018.
[30] A. Krizhevsky *et al.*, "Cifar-10," http://www.cs.toronto.edu/~kriz/cifar.html, 2017.
[31] C. Coleman *et al.*, "Analysis of dawnbench, a time-to-accuracy machine learning performance benchmark," *SIGOPS Operating Systems Review*, 2019.
[32] X. Gastaldi, "Shake-shake regularization," *arXiv preprint arXiv:1705.07485*, 2017.
[33] F. Chollet, "Xception: Deep learning with depthwise separable convolutions," *CVPR*, 2017.
[34] Q. Ho *et al.*, "More effective distributed ml via a stale synchronous parallel parameter server," *NeurIPS*, 2013.
[35] A. F. Aji *et al.*, "Sparse Communication for Distributed Gradient Descent," *arXiv preprint arXiv:1704.05021*, Apr. 2017.
[36] W. Wen *et al.*, "TernGrad: Ternary Gradients to Reduce Communication in Distributed Deep Learning," *NeurIPS*, 2017.
[37] D. Alistarh *et al.*, "QSGD: Communication-Efficient SGD via Gradient Quantization and Encoding," *NeurIPS*, 2017.
[38] S. Hochreiter *et al.*, "Flat minima," *Neural Computation*, 1997.
[39] S. L. Smith *et al.*, "A bayesian perspective on generalization and stochastic gradient descent," *ICLR*, 2018.
[40] P. Chaudhari *et al.*, "Entropy-sgd: Biasing gradient descent into wide valleys," *Journal of Statistical Mechanics: Theory and Experiment*, 2019.
[41] L. N. Smith, "Cyclical learning rates for training neural networks," *WACV*, 2017.