dSyncPS: Delayed Synchronization for Dynamic Deployment of Distributed Machine Learning

Yibo Guo
Case Western Reserve University
yibo.guo@case.edu

An Wang
Case Western Reserve University
an.wang@case.edu

Abstract

The increasing demand of applying machine learning technologies in various domains has driven the involvement of complex machine learning models. To fulfill this demand, distributed machine learning has become the de facto standard computing paradigm for model training. Machine-Learning-as-a-Service (MLaaS) has also emerged as a solution provided by cloud service providers to address this need. With MLaaS, customers can submit their models and training datasets to the service providers, and leverage the existing cloud infrastructure for model training and inference. However, we find that, for end users who require complex and accurate machine learning models but only obtains moderate amount of data, existing solutions are insufficient. The main issue is the lack of support for dynamic deployment of distributed machine learning tasks. To address this issue, we propose a parameter server based framework, called *dSyncPS*, that allows worker nodes to participate in training dynamically. The key idea is that it separates parameter synchronization from aggregation function in the parameter server nodes, thus resulting in a delayed synchrony.

CCS Concepts: • **Computer systems organization → Distributed architectures**.

Keywords: distributed machine learning, edge computing, synchronous parallel model

ACM Reference Format:
Yibo Guo and An Wang. 2022. *dSyncPS*: Delayed Synchronization for Dynamic Deployment of Distributed Machine Learning. In *2nd European Workshop on Machine Learning and Systems (EuroMLSys '22), April 5–8, 2022, RENNES, France.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3517207.3526981

1 Introduction

The past decade has witnessed the transformation of artificial intelligence (AI) and machine learning (ML). To deal with the ever increasing amount of data collected from various devices and sensors, more complex ML models have been proposed and applied for this purpose. The Moore's law, the engine that powered the machine learning revolution, has shown to be running out of its steam. Training a complex model with a single machine has become increasingly impossible. Thus, distributed computing has evolved into the de facto standard for machine learning. Driven by this demand, Machine-Learning-as-a-Service (MLaaS) emerges as an imperative solution.

MLaaS is a collection of various cloud-based platforms that use machine learning tools to provide machine learning solutions. These solutions typically include pre-built ML algorithms, such as natural language processing (NLP) and computer vision algorithms, ML management tools, ML training, and ML deployment at scale. Particularly for ML training, customers can submit their models and training datasets to service providers and leverage the existing cloud infrastructure for model training and inference. However, we argue that the training should be deployed and executed closed to the edge since most of the training data is collected from end users and devices. In this work, we propose a framework that extends distributed ML to the edge so that edge devices with moderate data size may contribute to complex model training.

There are two main motivations for this work. First, the current design of MLaaS can only benefit big data stakeholders, but not individual end users. For example, an end user may collect a moderate amount of data, and would like to use the data to train a more complex and accurate model to perform a specific task. In this case, the existing MLaaS can only provide limited help due to the insufficient data. If there are other end users with a similar demand, their data can be pooled together to perform this training task. Second, many edge servers have adequate bandwidth for training ML tasks. Given that most of their currently deployed applications are time-sensitive, it is unlikely that they are computation-intensive at the same time. On the other hand, it is expensive for end users to upload their data to a centralized cloud. Based on the above discussion, we argue that these non-critical ML training tasks should be executed at the edge.

To achieve this goal, however, three keys challenges need to be addressed. First, edge computing environment is more

EuroMLSys '22, April 5–8, 2022, RENNES, France
© 2022 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-9254-9/22/04.
https://doi.org/10.1145/3517207.3526981

dynamic and unreliable compared to data center infrastructures. For example, unexpected traffic bursts can happen, causing intermittent network congestion and training task failures. Second, modern distributed ML platforms, such as MXNet [5], Pytorch [13] and Tensorflow [3], do not support dynamic participation of worker and server nodes during training. Third, existing synchronization paradigms, bulk synchronous parallel (BSP) [8] and Stale Synchronous Parallel (SSP) [9], cannot satisfy the running of a distributed machine learning task at large scale. To overcome these challenges, a peer-to-peer (P2P) based framework, such as Ako [15], seems to be promising at first glance. However, P2P based frameworks may fall short in several aspects. First of all, training is performed in an asynchronous fashion in these frameworks, which means that there is no synchronization barrier of trained parameters. This would adversely affect the global model convergence so that convergence is either delayed or prevented. Secondly, system management, such as tracking involved peer nodes, is expensive and also error-prone. Such complex system management can also lead to training failures. Finally, since scaling up the worker nodes increases computation capacity, while scaling up the server nodes increases system management complexity. Independent scaling of worker nodes and server nodes might be necessary at the edge.

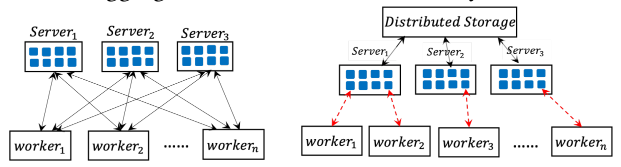
Motivated by these observations, we propose a novel framework, called *dSyncPS*, that is designed based on the parameter server architecture [10]. The **key idea** of this framework is that it separates parameter synchronization from aggregation function in the parameter server nodes, thus resulting in a delayed synchrony. Specifically, we separate the synchronization process into two independent phases: local and global synchronization. Local synchronization is performed among a small group of worker nodes that share geolocation affinity. Global synchronization is performed among all the participating parameter servers in a delayed fashion. In our framework, local synchronization is dictated by local parameter servers, and global synchronization is achieved by utilizing a distributed storage system, such as Amazon S3 [1]. More details will be discussed in Section 2 and Section 3.1.4. Additionally, we also develop a TCP-based application protocol to facilitate this mechanism. The proposed framework helps achieve two goals: (1) End users are provided with the flexibility to determine when to join or leave a ML training task, and (2) Non-critical distributed ML tasks can be deployed and executed at a large scale.

Through evaluations, we show that *dSyncPS* can effectively manage distributed training. It can also scale up with the increasing number of participating worker nodes efficiently. The rest of the paper is organized as following. We provide an overview of the proposed system in Section 2. The detailed of the system design are discussed in Section 3. Specifically, we discuss the design of parameter servers in Section 3.1, and the design of worker nodes in Section 3.2. The communication protocol formats and functions are specified in Section 3.3. The evaluation results are shown and analyzed in Section 4. We finally conclude our work in Section 6.

2 System Overview

System Components. There are three typical system architectures supporting the execution of distributed ML, parameter server (PS), peer-to-peer (P2P), and ring-Allreduce [4]. Our proposed framework, called *dSyncPS*, is built upon the PS architecture for two reasons: 1. The parameter server nodes of the PS architecture attributes to a central role in coordination of participating worker nodes. 2. Management is easier and less error-prone in PS architecture based systems. Additionally, such an architecture aligns with the current Edge computing paradigm.

The architecture of *dSyncPS* is shown in Figure 1b. *dSyncPS* consists of four major components: the worker nodes, the parameter server nodes, a custom-designed TCP-based protocol, and a distributed storage system. The worker and parameter server nodes work in similar ways to those in the current PS architectures (shown in Figure 1a), where worker nodes perform iterative training and parameter servers perform parameter aggregations. But there are **two key differences** that separate two architectures.



(a) Current PS architecture **(b)** Architecture of *dSyncPS*

Figure 1. System architectures

First, in the current PS architecture design, parameters are split by the key range among multiple server groups. Thus a worker node needs to maintain connections with each server in a group for synchronization. In the proposed *dSyncPS*, we split the keys vertically along the worker dimension so that each worker node only maintains one connection with an individual server node. This connection can be dynamically migrated to other server nodes in the same group or a different group if needed, such as when a server node goes offline. To facilitate dynamic migration, we develop a TCP-based application protocol that is discussed in details in Section 3.3. Second, the server nodes in the current PS architecture integrate the two key functions, aggregation and synchronization, with the Bulk-Synchronous Parallel (BSP) mechanism. In BSP, aggregation cannot be done until the server node receives updated parameters from all the worker nodes. Alternatively, we propose to separate global synchronization from aggregation in *dSyncPS*. Specifically, in local aggregation period, a server node only performs synchronized aggregation over a set of workers that are connected to this server node. The size of this set is expected to be small. While for the global aggregation, it is performed in a delayed fashion among different server nodes via a distributed storage system. Such a mechanism reduces the amount of time needed for synchronization.

System Workflow. With these components, the workflow of *dSyncPS* is shown in Figure 2. When a worker node first

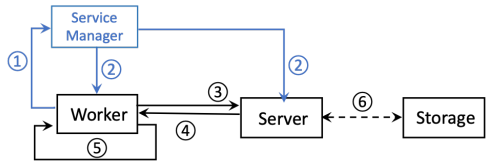


Figure 2. Workflow of *dSyncPS*

joins the network, it initiates a service request to the service manager (step ①), which is a component that handles distributed machine learning services as part of the MLaaS platform. Based on the location of the worker node, a nearby server node is assigned as the primary parameter server (PPS) that provides distributed training service for the worker (step ②). This can be done through DNS localization mechanisms. For example, RFC 7871 proposes EDNS0-Client-Subnet (ECS) which passes end-user subnet information through recursive resolvers to authoritative DNS servers for localization [6]. For the training, worker first sends a request to the server to obtain the current parameters of the model (step ③ and ④) since we assume that not all the worker nodes start the training at the same time. Once the worker gets the initial parameters, local computation begins (step ⑤). At the end of each iteration, the worker synchronizes its parameter values with its PPS in a similar way as step ③ and ④. In step ③, updated parameters are pushed to the server and the server sends back aggregated parameters in step ④. For the global synchronization among different server nodes, it is done in a delayed synchronous fashion via the distributed storage system (step ⑥). Specifically, each server performs local aggregation first and then the locally aggregated parameters are written back to the storage. Such updates are exclusive by utilizing locks. Detailed designs of each component are discussed next.

3 Design and Implementation

3.1 Parameter Server

3.1.1 Server Architecture. The architecture of a parameter server node is shown in Figure 3. There are four main

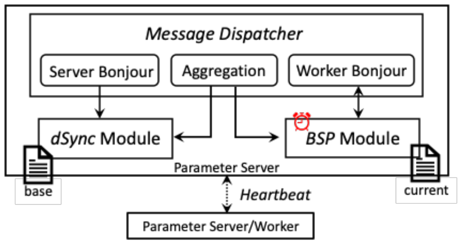


Figure 3. Architecture of parameter server

modules in each parameter server node: Dispatcher, *dSync* Module, *BSP* module, and a Heartbeat Module. The dispatcher module is responsible for receiving various types of messages from other nodes, and invoking the corresponding handler functions. The details of the proposed protocol and message types are discussed in Section 3.3. The *dSync* module initiates the delayed synchronization to a distributed storage system. The *BSP* module handles the synchronization among the local worker nodes. The heartbeat module helps maintain the liveness of other parameter server nodes in the system.

3.1.2 *BSP* Module. The core of a parameter server is an *Aggregation* function. When the message dispatcher receives parameter updates from a worker node, it invokes the aggregation function to update the current parameters. The results of this function are passed to the *dSync* module and the *BSP* module for synchronization operations. Specifically, the *BSP* module works follows a similar strategy to that of the current BSP paradigm. It allows the local worker nodes to synchronize after each training iteration. But the key difference is that we introduce a timer mechanism in the proposed *BSP* module. The purpose of this timer is to prevent worker from waiting so a long time. Generally, the parameter server maintains a list of active worker nodes for synchronization. If the parameter server does not receive the updated parameters from a worker node until the timer fires off, the parameter server continues without further waiting. By default, we set this timer to be 120 seconds, but this can be adjusted by the system operator. When a worker attempts a reconnection or first joins the training, the worker sends a `Worker Bonjour` message to the parameter server, which is handled by a function called *Worker Bonjour*. This function updates the list of active worker nodes when a worker joins and leaves the training.

3.1.3 Full-mesh Server Topology. On the other hand, each parameter server also maintains a list of their peer server nodes for global synchronization and load balancing purposes. To check the liveness of their peers, a *Heartbeat* function periodically sends heartbeat messages to the other server nodes. More about heartbeat will be discussed in section 3.1.7. When a server node first joins the system, it sends a `Server Bonjour` message to one of the existing server nodes, which we call designated contacting point (DCP), and is assigned via DNS redirection service. This request is processed by the *Server Bonjour* function. This function returns a list of active peer server nodes so that the new server can further establish connections with these nodes. Eventually, all the parameter servers form a full-mesh topology. If any of the parameter servers fails, it can be detected by all the other servers via the heartbeat messages.

3.1.4 *dSync* Module. The core function that implements global synchronization is the *dSync* Module. This module operates based on a hash table, called *status table*, that maps all the server nodes to their corresponding status: *ready* or *not ready*. Note that the *status table* is also used to maintain the liveness of a server node. If a server node is not detected alive, it is removed from this table. When the global synchronization is initiated, this module copies the locally synchronized parameter to the shared storage. Upon success, the server node, for instance S_0, sends a `Server Sync` message to all other peer nodes. Upon the receipt of this message, each server node updates their *status table* by changing the server status of S_0 from *not ready* to *ready*. When the server status of all the servers in the table become *ready*, one of the servers performs the global aggregation while all others wait for the operation to be complete. The server node can then issue another request to the distributed storage system to get the updated base parameters. Meanwhile, the server statuses of all the servers in the table are reset to *not ready*. Since this global synchronization is expensive, we cannot afford to do this for every single iteration. Therefore, we define a time interval, T, that determines the frequency of the global synchronization. In other words, servers perform the global synchronization after T iterations of local aggregations. Although servers may drift apart and become out of sync, the global synchronization enforces a barrier synchronization among all the servers. For the newly joined servers, they are forced to perform the first global synchronization upon the receipt of `SRV_SYNC` message regardless of their current local aggregation iterations. The details of this message will be discussed in Section 3.3.

3.1.5 Asynchronous Joining of Server Nodes.

A key challenge with this pre-defined synchronization interval is to handle the asynchronous joining of new server nodes. For instance, a new server node might join 2 iterations after the global synchronization. In this case, it would require all the other server nodes to wait for 2 additional iterations in order to achieve global synchronization. To solve this issue, we introduce a mechanism that allows the global synchronization to either advance or defer. Specifically, assume that all the existing server nodes are about to perform the i^{th} global synchronization, and receive the `Server Bonjour` message from a new server node. Then one of the two things happens. Either the i^{th} global synchronization is deferred to the next iteration, if it had not began. Or the $(i+1)^{th}$ global synchronization is advanced to the next iteration, if the i^{th} synchronization were going on or had finished.

3.1.6 Parameters Initialization.

A server manages two types of configuration files, which are shared *current* file and local configuration file. The *current* file is shared among all servers through the distributed storage, which contains the up-to-date parameters obtained from the last global synchronization. When a training network is created and initialized, the *current* file can be fetched from a previous training. The local configuration file is stored locally on each server, which contains the latest parameters from the last local synchronization. Upon global synchronization or new server joining the network, the local configuration file is overwritten by the *current* file. The local configuration file is shared between each server and their worker nodes, while the *current* file is shared among all the servers via the shared storage.

3.1.7 Heartbeat.

As mentioned in section 3.1.4, a heartbeat module is designed to detect peer aliveness. Such a module also exists in worker nodes. The heartbeat module contains two critical parts: an initiator and a health monitor. The heartbeat initiator broadcasts heartbeat message to all connect peers periodically, while the monitor checks peer connectivity. If one node, either server or worker node, is not heard over within a predetermined time interval, it is considered disconnected. As a result, the disconnected node is removed from the *status table* maintained by its peers.

3.2 Worker Node

3.2.1 Worker Architecture.

The design of a worker node is mainly centered around its interactions with the parameter server node and executions of model parameter updates. The architecture of a worker node is shown in Figure 4. Each worker has three main components: a *BSP* Module, a *Connection Manager* Module and a *Computation* Module. The *BSP* Module connects with that of the parameter server *BSP* Module to synchronize local model parameters after each iteration. The *Computation* Module implements the core functions for the training purpose. The *Connection Manager* is responsible for initiating connection with the parameter server and load balancing.

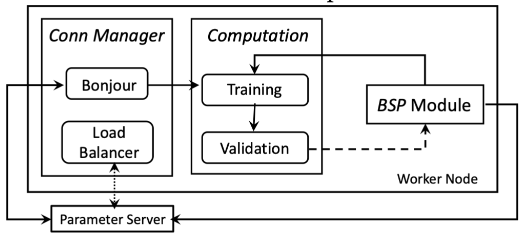


Figure 4. Architecture of worker node

3.2.2 Connection Manager.

A worker node joins the training by initiating a `Worker Bonjour` message to a parameter server that is obtained from the DNS services. In the response of this message, the worker can obtain a complete list of active parameter servers and a complete training package. This package includes a training model, a copy of the PPS's local configuration file, training code, and a validation code. The detailed discussions of this package are in Section 3.3. During the operation of a worker node, if it loses connection with its current PPS, the *Connection Manager* provides a mechanism to automatically reconnect the worker with a different parameter server. The new PPS is selected from the complete list of parameter servers based on some load balancing metrics. After a new connection is established, the worker node can resume its training based on the new parameter file obtained from its new PPS. Given a list of active parameter servers, a worker can also decide to switch to a different server as their new PPS proactively. The details of this load balancing mechanism are discussed later.

3.2.3 Computation Module. This module includes two key functions, the *Training* function and the *Validation* function. The *Training* function performs the typical forward and backward propagation calculations in DNN models. Generally, a *Validation* function is used to validate the model on test data, which is a subset of the data that does not overlap with the training data, at each training epoch's end by calculating a validation score or accuracy. In our design, this function is invoked at the end of each iteration for two purposes. First, "bad" parameters that do not contribute to the learning of the model significantly can be excluded temporarily from the aggregation. For example, we can set a threshold, V_τ, on the minimum validation score a worker node needs to achieve after each training iteration. If the score is below V_τ, then the *BSP* module does not send anything to the server and the worker continues training. As the training progresses, the validation score may increase and the accumulated training parameters can be included in the local aggregation after that.

The second purpose is to exclude poorly performed worker nodes from the training network. Due to the device heterogeneity at the edge, different worker nodes can perform diversely. If a worker node continuously (for at least N_τ iterations) calculates low validation scores, then it may benefit the entire training network by excluding such nodes. In this case, its PPS sends a termination message to close the connection with this worker node. By default, N_τ is set to be 10. It is difficult to set a default value for V_τ since the score increases as the training progresses. Alternatively, each worker node keeps the current best score locally, and compares it with the new validation score. If the new score wins, the current best score is updated to the new score; Otherwise, the worker continues to train.

3.2.4 Load Balancing. In most existing work, load balancing is performed on the server side and is transparent to the clients. In our design, the worker nodes can initiate the load balancing process by terminating its connection with the current PPS and connecting to a new PPS. The rationales behind this design are threefold. First, it is difficult to deploy a centralized load balancer that operates on the server side given the distributed deployment of parameter servers. Second, load balancing might cause interruption to training services if it is scheduled in the middle of worker nodes updating gradients to their PPS. Third, workers should be given the flexibility to choose a better performed link. Existing load balancing mechanisms make decisions mainly based on the server workload. In the distributed training system, however, communication efficiency is the key. Thus, workers should circumvent the congested links dynamically and proactively.

Node	evtType	Description
server	SRV_BONJOUR	Server bonjour messages
	ALIVE	Heartbeat messages
	TRAINMF	Transmit training and validation model to worker nodes in response to *Worker Bonjour* message
	BONJOUR_MDES	Send initial training parameters to a worker node
	SRV_SYNC	Global synchronization ready messages
worker	WRK_BONJOUR	Worker bonjour messages
	WRK_REQ_SRVS	Worker request parameter server information
	WRK_UPDATE	Worker send messages to update parameters

Table 1. Descriptions of different events

To implement this specific mechanism, the worker nodes need to obtain workload and latency related information on all the active parameter servers. To this end, the proposed framework include both passive and proactive measurements. For the passive measurements, parameter servers share their *load average* values with their peer server nodes by carrying this information in their heartbeat messages. On Linux, *load average* represents the overall system demand, including CPU, disk, uninterruptible locks. Then, these values are shared with all the participating worker nodes upon request. A worker node fetches the complete list of parameter servers and their information from its PPS by issuing requests periodically after each worker *BSP* update by using a `WRK_REQ_SRVS` message. PPS then respond with the load average information of all active servers. For the proactive measurements, a worker node sends probing packets to collect Round-Trip Time (RTT) values from a set of parameter servers whose load averages are below certain threshold. Then, the worker node computes a metric, what we call *Workload-Latency Product*, that is the product of the load average and the RTT values. If the calculated metric value is 20% lower than that of the current PPS, the worker node switches to this new parameter server; Otherwise, it stays with the current PPS. The intention is to prevent frequent reconnections caused by network traffic load and instance workload fluctuations.

3.3 Communication Protocol

3.3.1 Message Format. The format of the proposed communication messages is shown in Figure 5.

```
struct protocol_t {
    unsigned char preamble[PREAMBLE_SIZE];
    unsigned char evtType;
    String msgBody;
    u_int32_t msgBodyLen;
    unsigned char hashVal[SHA_DIGEST_LENGTH];
    u_int64_t ts;
}
```

Figure 5. Communication message format

In this data structure, the **`preamble`** specifies the message delimiter, which is a fixed string of size `PREAMBLE_SIZE` (8 bytes

by default). Since our communication protocol is built on top of TCP, and given that TCP is a stream-oriented protocol, a delimiter is necessary for a receiving node to determine the end of message in a data stream. The **evtType** represents the type of a message. The specific types of message are shown in Table 1. The **msgBody** contains the message data of variable length and **msgBodyLen** shows the length of this message. To guarantee message integrity, a 64-byte long SHA-1 hash value is included in **hashVal**, where SHA_DIGEST_LENGTH is 64 by default, and can be adjusted. The **ts** represents the timestamp when this message is generated.
These messages can be classified into two groups based on their audience groups. The *server* type of events are used for the coordination of different parameter servers, and the *worker* type of events are to dictate the interactions between worker nodes and their parameter servers. Most of these messages have been discussed in Section 3.1 and 3.2. Among all, SRV_SYNC is the message that a parameter server sends to all other server peers when it is ready for global synchronization and when the synchronization is done. This message includes an ID of the parameter server that performs the update. Upon the receipt of this message, a server node updates the entry that corresponds to this ID in its local *status table*. The WRK_REQ_SRVS message is used by worker nodes to initiate the load balancing process. Although alternative synchronization algorithms, such as ring all-reduce, can be applied to our system, it aims to optimize communication bandwidth rather than delay. Therefore, it may not deliver better performance than our proposed synchronization algorithm.

3.3.2 Training Package.

As mentioned in Section 3.2, a worker node receives a training package from the parameter server for initialization. The transmission of this package is achieved with two subsequent messages,TRAINMF and BONJOUR_MDES. The initial parameters of the model are transmitted with BONJOUR_MDES, and TRAINMF carries the serialized training and validation codes. Since all the participating worker nodes seek to train the same model, it is essential that they get the model structure from the parameter servers. This model can be a custom model or a pre-trained model. It should be selected by the worker node who initiated the training task. To share this model, we send the code snippet that constructs the model structure with uniform APIs, such as `buildModelArchitecture()`, to the the worker node. In this way, the uniform APIs can be updated easily without changing the other components of the execution.
Our framework is not specific to MXNet, it can be integrated with Pytorch and Tensorflow as well. Considering that different worker nodes may support different learning libraries, the worker nodes, when they first join the training network, needs to send their capabilities to the parameter server. If Pytorch is supported instead of MXNet, a code snippet that is implemented with Pytorch library is sent to the worker node. Note that the `buildModelArchitecture()` API remains the same regardless of the machine learning library. Similarly, the training code and validation code also need to be shared with the worker nodes so that the data processing and calculation is consistent across all the worker nodes. In this case, `validate()` is the uniform API implemented by the parameter server to calculate validation scores. The generation of these code snippets should be simple given that most of these computations are using standard APIs. All the code snippets are serialized into byte streams for message transmission, and deserialized upon receipt.

4 Evaluation

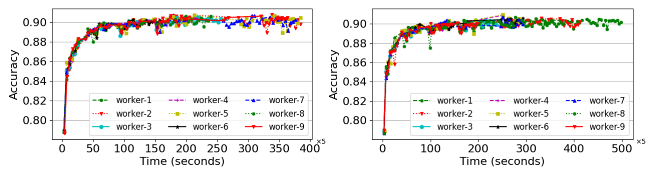
We evaluate *dSyncPS* from three aspects. First, we evaluate the accuracy of converged models of nine workers, which are managed by three parameter servers. Secondly, we inspect the effectiveness of the load balancing mechanisms. Finally, we also evaluate the scalability of *dSyncPS* by increasing the number of participating worker nodes.

4.1 Experiment Setup

We evaluate our work with 15 bare metal machines of type c220g5 on CloudLab [7]. They are equipped with two Intel Xeon Silver 10-core CPUs, 192GB ECC DDR4-2666 memory, and 10Gb NICs. Among these machines, three of them are used to execute parameter servers, while the remaining nodes are used to run worker processes. All the tests are conducted on Ubuntu 20.04 and MXNet 1.6.0. During the execution of *dSyncPS*, a logger is activated to record the post-aggregation and post-synchronization scores and their timestamps. It also keeps track of all the load-balancing events. To enable the synchronization of parameter servers, we also leverage AWS S3 to maintain the synchronized parameters. In all the following experiments, *dSyncPS* trains a LeNet model with the FashionMNIST dataset [2] with a learning rate of 0.1, optimizer *SGD*, and loss function of *Softmax Cross Entropy Loss*. Additionally, all the worker nodes are trained on the same data from the FashionMNIST dataset. The frequency of global synchronization is set to three, which means that the global synchronization happens every three local aggregations.

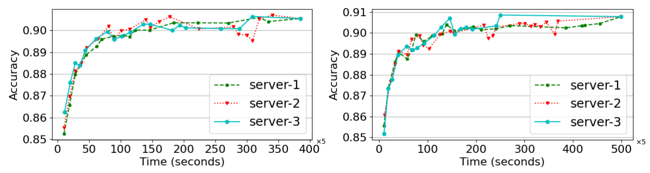
4.2 Accuracy of Converged Model

In this experiment, we measure the post-aggregation scores on the worker nodes and the post-synchronization scores on the server nodes as the training accuracy. Post-aggregation scores are calculated by validating the current model over a test dataset after each local aggregation. Post-synchornization scores are calculated in a similar way on the server side after each global synchronization. The results are shown in Figure 6 and Figure 7. Figure 6 shows the results of post-aggregation scores on the worker nodes, and Figure 7 shows the post-synchronization scores on the server nodes.



(a) Experiment 1 **(b)** Experiment 2

Figure 6. Post-aggregation scores



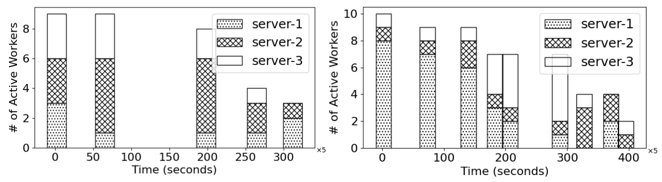
(a) Experiment 1 **(b)** Experiment 2

Figure 7. Post-synchronization scores

The two sub-figures represents two independent runs of the same experiment (out of 16 runs), respectively. They represent the best and the worst cases in our experiments. In all four figures, we linearly scale the x-axis by dividing each time point by 5. In other words, the total training time should range between 2k and 2.5k seconds. The initial validation scores are about 0.78 because the workers are provided with a reasonable set of initial parameters. We can observe that in both cases, the training accuracy increases rapidly to about 0.9 within 500 seconds. Similar observations can be obtained on the server side as well. The average accuracy of all experiments is about 0.905. We also compare this result with the scenario when the frequency of global training is set to 1. In that case, the convergence time will be approximately reduced by half. Note that the training process relies heavily on the validation function, thus a large mount of the training time is consumed by validation.

4.3 Effectiveness of Load Balancing

We also evaluate the load balancing mechanism of *dSyncPS* by examining the number of active worker nodes managed by each server during the training. The results are shown in Figure 8. In this experiment, we set up two different sce-



(a) Balanced scenario **(b)** Imbalanced scenario

Figure 8. Effectiveness of load balancing

narios. In the first scenario, all nine workers are assigned to each parameter server in a round-robin fashion so that all servers have equal workload initially. In the second scenario, we have ten worker nodes in total. Each of the first two parameter servers is assigned with one worker node, and the remaining eight worker nodes are assigned to the third server. Then we compare the results of the balanced and the imbalanced scenarios. In this figure, three different bars represent the number of active works on each server, respectively. The corresponding time values represent the points when load balancing happens, i.e., one or more workers reconnect themselves to a different server. For the balanced scenario, we can see that the workloads of each server remain almost balanced during training. For the imbalanced scenario, workers eventually also attempt to connect to servers in a load-balancing fashion. Note that the total number of workers will decrease as their training converges and they leave the network. The results suggest that our proposed load balancing mechanism works efficiently to distribute workloads to parameter servers. In figure 8, the total count of active worker is keep decreasing because workers may exit if no better model could be discovered within the given training iteration threshold.

4.4 Scalability of *dSyncPS*

In this group of experiments, we attempt to evaluate the scalability of *dSyncPS* by incrementally increasing the number of worker nodes in the system. Initially, all three parameter servers are signed with one worker node each. Then we increase the total number of worker nodes by two in each experiment. We then compare the average local aggregation time and the average global synchronization time of the parameter servers. The results are shown in Figure 9. In this

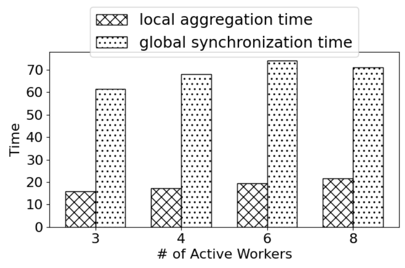


Figure 9. Scalability of *dSyncPS*

figure, each group of experiments has two bars, one representing the average local aggregation time and the other representing the average global synchronization time. We can see that both time increase slightly as more workers are added into the system. But they are still manageable by *dSyncPS*. The results suggest that *dSyncPS* can handle the increasing number of worker nodes effectively. In practice, the capacity of parameter servers should also scale up accordingly so that the servers are not overloaded.

5 Related Work

Many prior work adopt the parameter server based architecture to build distributed machine learning frameworks. Ooi *et al.* developed SINGA, a deep learning framework that supports multiple partitioning and synchronization schemes. It supports complex cluster topologies, such as one with multiple server groups [12]. Zhang *et al.* proposed, Poseidon, that implements a hybrid synchonization model: communication is performed either between the parameter server and the worker nodes or among the worker nodes in a peer-to-peer fashion, depending on the model structures and the size of the clusters [16]. Su *et al.* proposed a novel distributed training framework that performs different types of synchronization for different model layers [14]. For example, ring-based synchronization is used to update the large amount of parameters in CNN models, and All-Reduce synchronization is used to update the frequently changed parameters in the last layer. A similar approach has also been proposed by Li *et al.*, a collection of empirical policies were derived to determine how and when to use BSP and ASP synchronizations [11]. More recently, Zhou *et al.* proposed a Community-aware Synchronous Parallel (CASP) mechanism by leveraging advanced actor-critic (A3C) based algorithm to intelligently determine community configuration and fully improve the synchronization performance [17].

6 Conclusion

In this paper, we propose to develop a parameter-server based framework for dynamic deployment of distributed machine learning tasks. To achieve this goal, we separate the parameter synchronization from aggregation function in the parameter server nodes so that synchronization is performed in a delayed fashion. Specifically, we separate the synchronization process into two independent phases: local and global synchronization. In our framework, local synchronization is dictated by local parameter servers, and global synchronization is achieved via a distributed storage system. Additionally, we also develop a TCP-based application protocol to facilitate this mechanism. From our evaluation results, our proposed framework can perform training effectively and efficiently.

7 Acknowledgments

We appreciate the constructive comments from the reviewers. This work has been supported in part by CNS-2008468 and Google Faculty Research Award.

References

[1] 2022. Cloud Object Storage – Amazon S3 – Amazon Web Services. https://aws.amazon.com/s3

[2] 2022. Fashion MNIST. https://www.kaggle.com/zalando-research/fashionmnist

[3] Martín Abadi, Paul Barham, Jianmin Chen, et al. 2016. Tensorflow: A system for large-scale machine learning. In *Procs of USENIX OSDI*.

[4] Salem Alqahtani and Murat Demirbas. 2019. Performance analysis and comparison of distributed machine learning systems. *arXiv:1909.02061* (2019).

[5] Tianqi Chen, Mu Li, Yutian Li, Min Lin, et al. 2015. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv:1512.01274*.

[6] Carlo Contavalli, Wilmer van der Gaast, David C Lawrence, and Warren "Ace" Kumari. 2016. Client Subnet in DNS Queries. https://doi.org/10.17487/RFC7871

[7] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, et al. 2019. The Design and Operation of CloudLab. In *Procs of USENIX ATC*.

[8] Alexandros V Gerbessiotis and Leslie G Valiant. 1994. Direct bulk-synchronous parallel algorithms. *Journal of parallel and distributed computing* (1994).

[9] Qirong Ho, James Cipar, Henggang Cui, Seunghak Lee, Jin Kyu Kim, Phillip B Gibbons, Garth A Gibson, Greg Ganger, and Eric P Xing. 2013. More effective distributed ml via a stale synchronous parallel parameter server. In *Advances in neural information processing systems*.

[10] Mu Li, David G Andersen, Jun Woo Park, Alexander J Smola, Amr Ahmed, Vanja Josifovski, et al. 2014. Scaling distributed machine learning with the parameter server. In *Procs of USENIX OSDI*.

[11] Shijian Li, Oren Mangoubi, Lijie Xu, and Tian Guo. 2021. Sync-Switch: Hybrid Parameter Synchronization for Distributed Deep Learning. *arXiv:2104.08364*.

[12] Beng Chin Ooi, Kian-Lee Tan, Sheng Wang, Wei Wang, Qingchao Cai, Gang Chen, Jinyang Gao, Zhaojing Luo, Anthony KH Tung, Yuan Wang, et al. 2015. SINGA: A distributed deep learning platform. In *Procs of ACM Multimedia*.

[13] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems* (2019).

[14] Yuxin Su, Michael Lyu, et al. 2018. Communication-efficient distributed deep metric learning with hybrid synchronization. In *Procs of ACM CIKM*.

[15] Pijika Watcharapichat, Victoria Lopez Morales, Raul Castro Fernandez, and Peter Pietzuch. 2016. Ako: Decentralised deep learning with partial gradient exchange. In *Procs of ACM SoCC*.

[16] Hao Zhang, Zeyu Zheng, Shizhen Xu, Wei Dai, Qirong Ho, Xiaodan Liang, Zhiting Hu, Jinliang Wei, Pengtao Xie, and Eric P Xing. 2017. Poseidon: An efficient communication architecture for distributed deep learning on {GPU} clusters. In *Procs of USENIX ATC*.

[17] Qihua Zhou, Song Guo, Zhihao Qu, Peng Li, Li Li, Minyi Guo, and Kun Wang. 2020. Petrel: Heterogeneity-aware distributed deep learning via hybrid synchronization. *Procs of IEEE TPDS* (2020).