Elastic Parameter Server: Accelerating ML Training with Scalable Resource Scheduling

Shaoqi Wang, *Student Member, IEEE,* Aidi Pi, *Student Member, IEEE,*
Xiaobo Zhou, *Senior Member, IEEE,*

Abstract—Parameter server (PS) based on worker-server communication is designed for distributed machine learning (ML) training in clusters. In feedback-driven exploration of ML model training, users exploit early feedback from each job to decide whether to kill the job or keep it running so as to find the optimal model configuration. However, PS does not support adjusting the number of workers and servers of a job at runtime. It becomes the bottleneck of scalable distributed ML training because the cluster resources cannot be dynamically allocated or deallocated to jobs, resulting in significant early feedback latency and resource under-utilization. This paper rethinks the principle of PS architecture. We present Elastic Parameter Server (EPS), a lightweight and user-transparent PS that accelerates feedback-driven exploration for distributed ML training. EPS allows to remove a subset of workers and servers from running jobs and allocate the released resources to an incoming job at runtime so as to reduce its early feedback latency. It can also use the released resources from a killed job to add workers and servers to running jobs to improve resource utilization and the training speed. We develop a heuristic scheduler that leverages EPS and offers scalable resource scheduling for multiple ML jobs. We implement EPS in Tencent Angel and the scheduler in Apache Yarn, and conduct evaluations with various ML models. Experimental results show that EPS achieves up to 1.5x improvement on the ML training speed compared to PS.

Index Terms—Parameter Server, Feedback-driven Exploration, ML Model Training, Resource Scheduling, Elasticity.

1 Introduction

Distributed ML frameworks such as Tencent Angel [1], [2], TensorFlow [3], Petuum Bosen [4], [5], and MXNet [6] have emerged to support distributed ML training in clusters. In these frameworks, parameter server (PS) [7], which is a common architecture that coordinates the distributed storage and access of ML model parameters, has been regarded as the key component for distributed training of ML jobs with a large dataset and a high dimensional model. In PS, there are two types of resource units, workers and servers. The servers serve as a distributed storage of model parameters, and the workers update the model in parallel with their partitions of the training data. The ML frameworks employ cluster schedulers, such as Mesos [8] and Apache Yarn [9], to allocate a fixed amount of resources (i.e., a number of workers and servers) to a training job upon its submission according to the resource requirements specified by the user.

ML model training is a feedback-driven exploration process [10], [11], [12], [13] where a user trains a ML model repeatedly to tune its hyperparameters and adjust the model structure. In specific, the user submits multiple training jobs with different model configurations in a certain distribution. The job execution is divided into two phases: *early feedback phase* (e.g., the first 100 iterations) and *training phase* (the rest iterations). The user exploits early feedback of the model accuracy from each job to decide whether to kill the job or keep it running so as to find the optimal configuration of the model. The larger number of jobs that can provide early feedback in time, the faster the feedback-driven exploration could be. The user also sets a pre-defined threshold that limits the number of running jobs in the training phase.

However, the PS architecture has a fundamental limitation, that is, there is no sufficient support for adjusting the number of workers and servers at runtime. It leads to two major drawbacks in ML model training. First, when all cluster resources are allocated to running jobs in the training phase, they hold resources for hours or even days until the completion since the number of allocated workers and servers are unadjustable. Incoming jobs with potential better model configurations will be queued and thus model training suffers significant early feedback latency and prolonged training time. A straightforward approach is to kill running jobs and allocate their resources to incoming jobs, but it incurs high overhead. Second, when jobs with poor early feedbacks are killed by the user and new jobs are not submitted yet, the resources freed from the killed jobs cannot be timely utilized to allocate new workers and servers to the rest running jobs, leading to significant resource under-utilization.

Intuitively, one may propose to reduce early feedback latency through reserving a resource pool dedicated to running jobs in the early feedback phase. However, when the user decides to keep a job running after its early feedback phase, moving the job out of the resource pool incurs non-trivial overhead since the job has to be killed first and restarted using different resources. Also, it is difficult to decide the size of the resource pool. A small resource pool prolongs the early feedback phase, while a large one leads to resource under-utilization when there is no incoming job.

Recent years, innovative approaches have been proposed to improve the scalability of distributed ML training

- *S. Wang, A. Pi and X. Zhou are with the Department of Computer Science, University of Colorado, Colorado Springs, CO 80918, USA. E-mail: {swang, epi, xzhou}@uccs.edu.*

This research was supported in part by U.S. NSF grant CCF-1816850.

through optimizing PS architecture [14], [15], [16], [17], [18], [19] and cluster schedulers [13], [20], [21]. There are also research focused on developing parameter server based distributed machine learning recommendation systems [22], [23], [24]. However, the fundamental limitation of PS architecture has not been addressed. Moreover, There are a few recent efforts that aim to speedup feedback-driven exploration, such as SLAQ [25], Hyperdrive [12], Gandiva [13], and Pytorch [26]. These approaches do not support PS architecture.

Interestingly, Tencent Angel introduces fault tolerance support in PS [1]. In specific, when a server or worker is crashed, a new server or worker will be allocated and initialized to resume job training. However, the fault tolerance mechanism cannot be used to adjust the number of workers and servers at runtime.

In this paper, we tackle the fundamental limitation of the PS architecture. We design and develop Elastic Parameter Server (EPS), an augmented and lightweight PS that allows adjusting the number of workers and servers for a ML job at runtime and significantly accelerates feedback-driven exploration and distributed ML model training. EPS addresses several key design challenges. First, it supports runtime update of the dependency between workers and servers. Second, it enables server-to-server and worker-to-worker communications to support dynamic data and parameter transmissions between the removed or added workers and servers and the existing workers and servers. Finally, it supports parameters or input data repartitioning in order to determine the amount of parameters or data to be transferred among servers or workers.

To leverage EPS, we develop a heuristic scheduler that adaptively specifies the number of workers and servers to be added or removed. When no resource is available for an incoming job, EPS removes a specified number of workers and servers from the running jobs while maintaining their execution, and allocate the released resources to the incoming job to reduce its early feedback latency. This heuristic is based on the insight that memory in each worker is over-provisioned to avoid Out-Of-Memory errors [27]. Thus, the input data on the removed workers can be transmitted to and stored in the other workers, and so do the parameters on the removed servers. When a job is killed by the user, the freed resources are utilized by EPS to add workers and servers to the running jobs so as to improve resource utilization and accelerate ML model training.

The design of EPS consists of three core components: dependency update, data and parameter transmission, and global decision making. With the dependency update component, a job can keep running without the removed workers and servers because the component removes the dependency between the removed workers and servers and the remaining workers and servers. A job can also keep running with the added workers and servers since the component adds the dependency between the added workers and servers and the existing workers and servers. The transmission component enables server-to-server parameter transmission and worker-to-worker data transmission. The global decision making component makes data and parameter transmission decisions by repartitioning input data and parameters for runtime adjustment. For example, the component partitions the input data on a removed worker and specifies the workers to which the input data is transmitted.

Upon the three core components, EPS provides four functions with interfaces to the heuristic scheduler. The four functions are adding workers, adding servers, removing workers, and removing servers, all at runtime.

In a nutshell, we make the following contributions.

1) We design EPS, which enables lightweight and user-transparent resource adjustment to a running job by augmenting PS with three core components and four functions.
2) We develop a heuristic scheduler that leverages EPS and offers scalable resource scheduling to speed up feedback-driven exploration and accelerate distributed ML training in clusters.
3) We implement EPS in open-source Tencent Angel framework and the scheduler in Apache Yarn. We conduct evaluations with various ML models. Experimental results show that EPS achieves up to 1.5x and 50% improvement on the ML training speed compared to PS and the kill-based approach deployed in Optimus [21], respectively. EPS is lightweight in runtime resource adjustment, compared to the kill-based approach.

In the following, Section 2 gives motivation on EPS. Section 3 describes the design of EPS. Section 4 presents the design of the heuristic scheduler. Section 5 describes the system implementation. Sections 6 and 7 present the experimental setup and evaluation results. Section 8 reviews related work, and Section 9 concludes the paper.

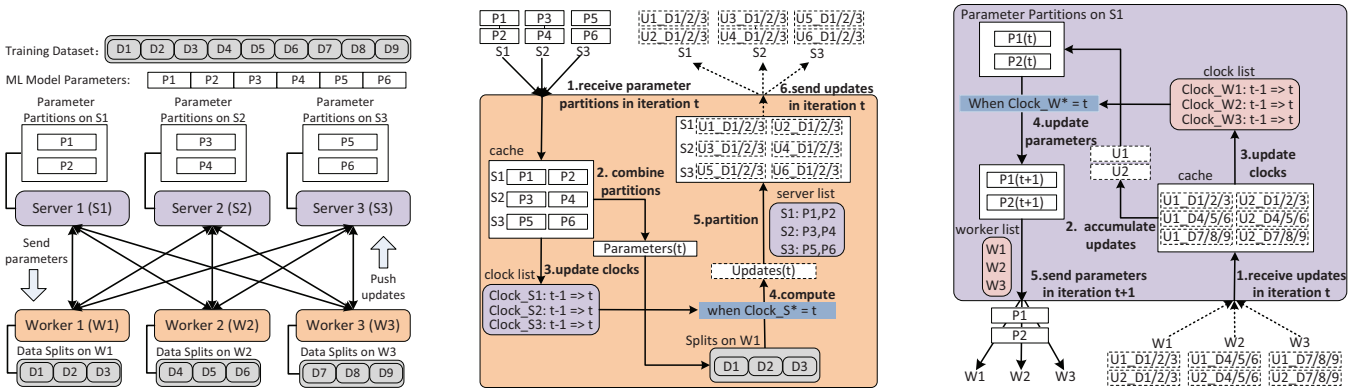
2 Background and Motivation

In this section, we first describe the feedback-driven exploration in ML model training. We then introduce PS architecture for distributed ML training in clusters. Further, we present a case study to show that resource allocation with the current PS results in significant early feedback latency, and show the advantage of scalable resource allocation at runtime with EPS.

2.1 Multi-job in Feedback-driven Exploration

One key characteristic of ML model training is the feedback-driven exploration due to the inherent iterative trial-and-error methodology of ML. A user often trains a model on the same dataset multiple times for the exploratory purpose, by continuously submitting jobs with different configurations such as model structures and hyperparameters. The execution of each job is divided into two phases: the early feedback phase (e.g., the first 100 iterations) and the training phase (the rest iterations). The first phase provides early feedback of model accuracy to users so as to help guide the optimization process of model configurations.

A ML model contains many hyperparameters that need to be specified, for example, the learning rate of model parameters, the batch size in each iteration, the number of hidden layers in a neural network, etc. Ideal values of these hyperparameters usually cannot be calculated based on the training data. The current approaches empirically explore different combinations of hyperparameter values [10], [11].



(a) PS architecture. (b) The execution mechanism inside a worker. (c) The execution mechanism inside a server.

Fig. 1. PS architecture and internal execution mechanisms in workers and servers. P*m*(*t*) refers to the *m*th partition in iteration *t*. The updates of partition P*m* is notated as U*m*. U*m*_D*i*/*j*/*k* refers to U*m* generated from splits D*i*, D*j*, and D*k*.

In order to tune hyperparameters in a ML model, a user submits multiple ML jobs, in which each job trains the model using one combination of hyperparameter values. The user also uses early feedback from each job to decide whether to kill the job or keep it running. When all jobs are finished, the ML model is configured with the hyperparameters of the job that gives the best training result, e.g., the highest ML model accuracy.

2.2 Parameter Server Architecture

Prevalent distributed ML frameworks (e.g., TensorFlow, Tencent Angel, Petuum Bosen) employ PS to train ML models iteratively in clusters. The PS architecture is shown in Figure 1(a). There are two types of resource units: workers and servers. The training dataset is partitioned into multiple data splits (i.e., D1 to D9) that are distributed to workers. ML model parameters are divided into multiple parameter partitions (i.e., P1 to P6) that are distributed to servers. In one iteration, each server sends the latest parameters to workers. Then each worker computes parameter updates (e.g., gradients) locally using its data splits and pushes the updates to the servers that store the corresponding parameters. After receiving parameter updates, servers update the model parameters using a pre-defined optimization algorithm, such as Stochastic Gradient Descent. There are explicit dependencies between workers and servers in conducting computation and model parameter updates. For example, in Figure 1(a), worker $W1$ depends on servers $S1$, $S2$, and $S3$ since the worker has to push the updates to the three servers. Server $S1$ depends on workers $W1$, $W2$, and $W3$ since the server has to send the updated parameters to the three workers.

Figure 1(b) illustrates the execution mechanism in worker $W1$. The execution at each iteration involves six steps. 1) $W1$ receives parameter partitions and iteration number from each server. 2) $W1$ combines the partitions into ML model parameters when all partitions are received. 3) $W1$ updates a clock list. The clock list stores the clock of each server, which is the iteration number of partitions in the server. 4) When the clocks of all servers in the list increase by one, $W1$ conducts computation using its data splits and ML model parameters to generate parameter updates. 5) $W1$ partitions the generated updates based on the server list that stores the metadata of partitions in each server. 6) $W1$ sends the partitioned updates to servers in the list. According to the metadata in the list, $W1$ is able to partition and send updates in a way that the updates are sent to the servers that store the corresponding parameters. Note that the clock list and the server list maintain the dependency between $W1$ and the three servers. The worker implementation varies in different ML frameworks. In some frameworks [1], [3], [5], the first three steps are overlapped.

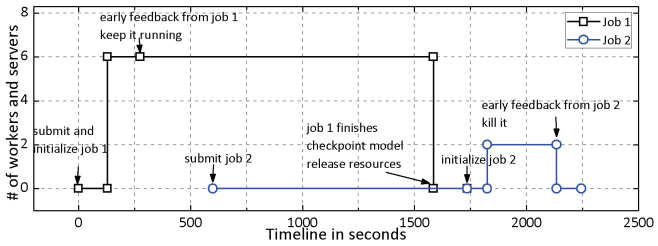
Figure 1(c) shows the execution mechanism in server $S1$. The execution at each iteration involves five steps. 1) $S1$ receives parameter updates from each worker. 2) $S1$ accumulates the updates of the same partition when all updates are received. 3) $S1$ updates a clock list. The clock list stores the clock of each worker, which is the iteration number of updates pushed by the worker. 4) When the clocks of all workers in the list increase by one, $S1$ updates its partitions and iteration number. 5) $S1$ sends the updated partitions to the workers in the worker list and starts the next iteration. The clock list and the worker list maintain the dependency between $S1$ and the three workers.

For synchronization between servers and workers, there are several popular models, such as BSP [28], A-BSP [29], SSP [4], [5], and ASP [30].

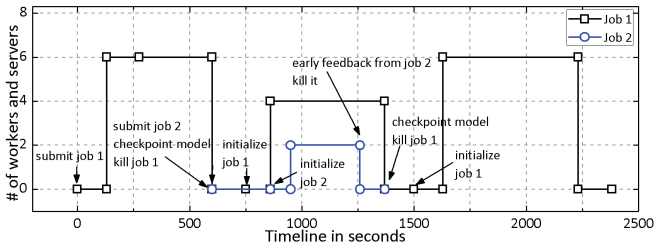
2.3 A Case Study

We created a small 4-node cluster to demonstrate that resource allocation upon to the current PS could result in significant early feedback latency and the potential of EPS reducing the latency. Note that EPS adopts a simple heuristic scheduler in this case study. The outcome of the study motivates us to design and develop a more sophisticated heuristic scheduler (i.e., H-scheduler) in a large cluster.

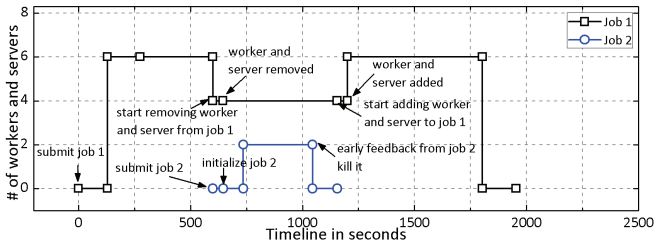
The distributed ML framework deployed in the cluster is Tencent Angel. Its underlying scheduling framework is Apache Yarn that allocates resource units in workers and servers to ML training jobs. Workers and servers run in Yarn containers. The cluster is configured with one master and three slave nodes. Each slave node can run one worker and one server. BSP model is used for synchronization between servers and workers.



(a) Job execution due to the default PS.



(b) Job execution due to the kill-based approach.



(c) Job execution due to EPS.

Fig. 2. Microscopic views of the execution of two jobs.

Two Logistic Regression jobs with different hyperparameter values are submitted at different time by a user. Job 1 is configured with correct hyperparameters and it converges quickly. Job 2 is configured with wrong hyperparameters and it converges slowly. Table 1 gives the job details. The performance metrics include the average queuing delay, the average early feedback latency, and the model training time. Note that for a training job, the queuing delay is a part of the early feedback latency. Three approaches are examined, the default PS, a kill-based approach used in Optimus [21], and EPS. The kill-based approach terminates a job and restarts it to adjust its number of workers and servers.

TABLE 1
Two Logistic Regression Jobs in the Case Study.

Job id	Submission time	Resources	Hyperparameters
1	0 second	three workers and three servers	correct hyperparameters
2	600_{th} second	one worker and one server	wrong hyperparameters

Figure 2 shows the microscopic views of the execution of two ML jobs due to three approaches. Figure 2(a) shows the number of workers and servers used by the two jobs due to the default PS, respectively. In specific, when job 1 is submitted at time 0, Yarn initializes the job and allocates three workers and three servers to start model training. At the 276_{th} second, job 1 finishes its first 100 iterations. The user obtains the early feedback of model accuracy and decides to keep the job running since the obtained model converges quickly. When job 2 is submitted at the 600_{th} second, all cluster resources have been allocated to job 1 so that job 2 is queued. At the 1735_{th} second, job 1 finishes all its iterations. Yarn releases its allocated resources, initializes job 2, and allocates one worker and one server to the job to start model training. At the 2134_{th} second, job 2 finishes its first 100 iterations. The user decides to kill it due to poor early feedback of model accuracy.

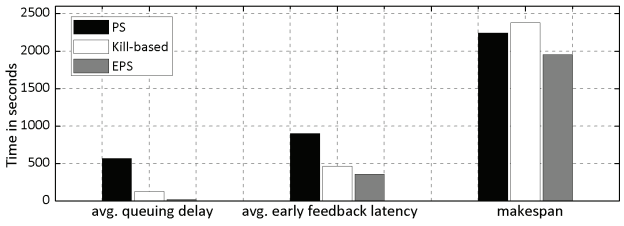


Fig. 3. Performance due to three approaches.

Figure 2(b) shows the execution process due to the kill-based approach. When job 2 is submitted at the 600_{th} second, the approach removes one worker and one server from job 1 through three steps. Firstly, Angel checkpoints the ML model in job 1. Secondly, Yarn terminates job 1 and releases its allocated resources. Finally, Yarn re-initializes job 1 and allocates two workers and two servers to resume its model training. Then, Yarn initializes job 2 and allocates one worker and one server to start model training. To utilize the resources released by job 2 at the 1369_{th} second, the approach adds one worker and one server to job 1 through three steps. Firstly, Angel checkpoints the ML model in job 1. Secondly, Yarn terminates job 1 and releases its allocated resources. Finally, Yarn re-initializes job 1 and allocates three workers and three servers to resume its model training.

Figure 2(c) shows the execution process due to EPS that allows to adjust the number of workers and servers at runtime. When job 2 is submitted at the 600_{th} second, EPS removes one worker and one server from job 1 through three steps. Firstly, the data on the removed worker and the parameters on the removed server are transmitted to the other workers and servers, respectively. Secondly, EPS removes the dependency between the removed worker and server and the remaining workers and servers. Finally, Yarn releases the resources that were previously allocated to the removed worker and server. Yarn initializes job 2 and allocates one worker and one server to start its model training.

To utilize the resources released by job 2 at the 1045_{th} second, EPS adds one worker and one server to job 1 through three steps. Firstly, EPS requests Yarn to add one worker and one server to job 1. Secondly, EPS adds the dependency between the added worker and server and the existing workers and servers. Finally, a subset of data on the two existing workers and a subset of parameters on the two existing servers are transmitted to the added worker and server, respectively. Overall, the overhead of adding or removing workers and servers by EPS is much lower than that by the kill-based approach.

Figure 3 depicts the performance of job executions due to the three approaches. EPS achieves the smallest average queuing delay, average early feedback latency, and model training time. The default PS leads to the largest average

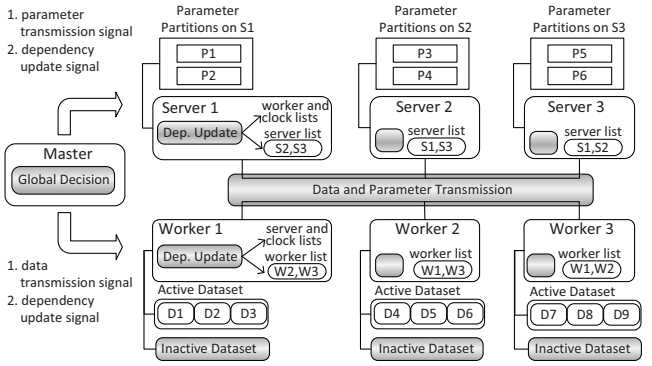


Fig. 4. The architecture of EPS.

queuing delay and average early feedback latency, because job 2 has to wait in the queue until job 1 finishes. The kill-based approach achieves a smaller queuing delay of job 2 than that due to the default PS. However, it brings in non-trivial overhead when adjusting the number of workers and servers allocated to a job. In the case study, it results in the largest model training time because two jobs are relatively small and thus the overhead becomes more severe.

3 EPS Design

In this section, we first describe the architecture of EPS with its three key components. We then introduce four functions in EPS: adding workers, adding servers, removing workers, and removing servers.

3.1 EPS Architecture

Figure 4 illustrates the architecture of EPS. The new components in EPS are shown in grey. EPS centers on three core components: dependency update, data and parameter transmission, and global decision making. EPS also includes a two-dataset component that can reduce data transmission overhead between workers.

3.1.1 Dependency Update

In PS, workers and servers depend on each other explicitly. A worker pushes updates to its dependent servers that store the parameters, and a server sends the parameters to its dependent workers that use the parameters in computation. The dependency is maintained by a server list and a clock list in workers, and a worker list and a clock list in servers. In EPS, the dependency update component is designed to update the lists to remove or add dependency. When workers and servers are removed from a running job, the component deletes the removed workers and servers from the lists in other workers and servers of the running job. When new workers and servers are added, it adds them to the lists of the current workers and servers.

3.1.2 Data and Parameter Transmission

In PS, there are worker-to-server and server-to-worker communications that push updates and send parameters. EPS enables worker-to-worker and server-to-server communications to transmit data splits and parameter partitions, respectively. With the component, each worker stores a worker list and each server stores a server list. The worker and server lists contain information such as the host IP and port number that are used to communicate with the other workers and servers. When a worker is removed, the removed worker sends its data splits to other workers in the worker list. When a new worker is added, the existing workers first update their worker lists using the dependency update component and then transmit a subset of the data splits to the new worker. When removing or adding a server, parameter transmission is conducted in a similar way.

3.1.3 Global Decision Making

EPS uses a global decision algorithm to make data and parameter transmission decisions. For example, for data splits on a removed worker, the algorithm partitions the splits and determines the worker to which each split is transmitted. When a new worker is added, it partitions data splits on the current workers and determines the number of splits to be transmitted to the new worker. Parameter transmission decisions are made similarly. The algorithm avoids the situation that parameters and data are irregularly distributed across servers and workers.

The algorithm is conducted in the master, e.g., ApplicationMaster in Yarn, which stores the metadata of the data splits on each worker and the parameter partitions on each server. After the decision is made, the master sends two signals to workers and servers. The first one is the data and parameter transmission signal that requests the execution of transmission. The second one is the dependency update signal that requests the dependency update component to update the lists in workers and servers, respectively.

3.1.4 Two datasets

In PS, each worker loads its local data splits into memory for computation. In EPS, the splits are divided into two datasets: active dataset and inactive dataset. The splits in the active dataset are loaded into memory and the splits in the inactive dataset are only stored on local disks. The rationale of the two-dataset component is that it helps to reduce data transmission overhead when workers are added or removed. For example, when a worker is added, a subset of the data splits on the other workers are transmitted to the added worker. Meanwhile, these data splits are moved to inactive dataset. When the added worker is removed later, it does not need to transmit its data splits to the other workers because they can read the splits from the inactive dataset directly.

3.2 Function Calls to Adding Servers or Workers

With EPS, when adding servers to a running job there are three steps, as shown in Figure 5. In the first step, the scheduler allocates containers and initializes the added servers. In the second step, EPS adds dependency between the added servers and the current workers and servers. In the third step, parameters on the current servers are transmitted to the added servers. Note that adding workers to a running job is done in three similar steps.

The three steps of adding servers are partially overlapped and there are synchronization barriers. In the first step, the scheduler launches the containers of the added

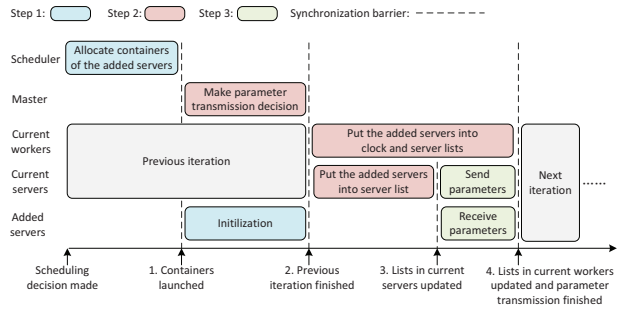


Fig. 5. Three steps for adding servers to a job.

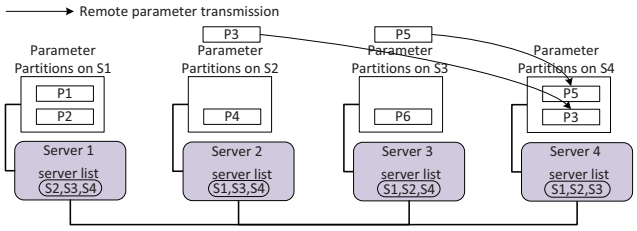


Fig. 6. Parameter transmission in adding server $S4$.

servers. After launching the containers (barrier 1), the added servers start the initialization process that is overlapped with the parameter transmission decision in the second step. The master makes the decision on the number of parameter partitions to be transmitted to each added server. The first step and the decision making in the second step are overlapped with the previous iteration executed by the current workers and servers. After making the decision, finishing the initialization, and finishing the previous iteration (barrier 2), the current workers put the added servers into their clock lists and server lists to add the dependency. Meanwhile, the current servers put the added servers into their server lists. After the server lists in the current servers are updated (barrier 3), the current servers transmit their parameter partitions to the added servers in the third step. After parameter partitions are transmitted and all lists are updated (barrier 4), the job starts the next iteration with the added servers.

When adding servers, the master makes parameter transmission decisions using a round-robin algorithm. The algorithm determines the number of partitions and which partitions to be transmitted to each added server. When adding workers, the master makes data transmission decisions using a similar algorithm. The major difference between the two algorithms is that the data splits in data transmission are not only remotely transmitted to the added workers but also locally moved to the inactive datasets. Note that the round-robin algorithm can avoid the situation where parameters and data are irregularly distributed across workers and servers.

We use two examples to illustrate parameter and data transmissions. A job is running with three workers and three servers as shown in Figure 4. When a new server ($S4$) is added to the job, the master decides to transmit two parameter partitions to $S4$. The transmission procedure is illustrated in Figure 6. First, the current servers put $S4$ into their server lists. Then, parameter partitions $P3$ and $P5$ are

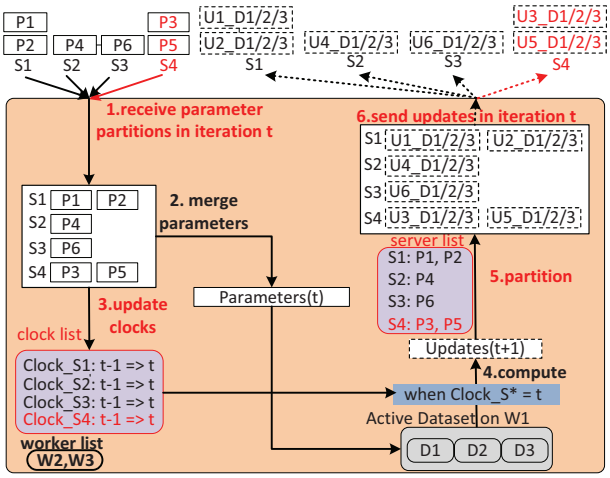


Fig. 7. Dependency between $S4$ and $W1$ is added.

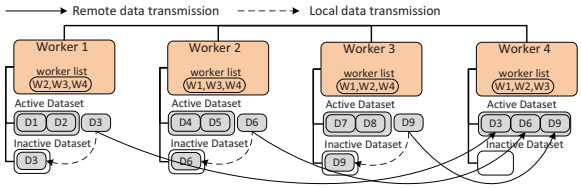


Fig. 8. Data transmissions in adding worker $W4$.

transmitted to $S4$. After the transmission, $P3$ and $P5$ are removed from $S2$ and $S3$, respectively. The server lists and clock lists in the current workers are updated to add the dependency between them and $S4$. Figure 7 illustrates the dependency update procedure inside worker $W1$ and the updates are shown in red. In specific, worker $W1$ receives $P3$ and $P5$ from $S4$ and updates the clock of $S4$. The generated updates are partitioned based on the updated server list. The updates of $P3$ and $P5$ are sent to $S4$.

When a new worker ($W4$) is added to the job, the master decides to transmitted three data splits to the added worker. The transmission procedure is illustrated in Figure 8. The

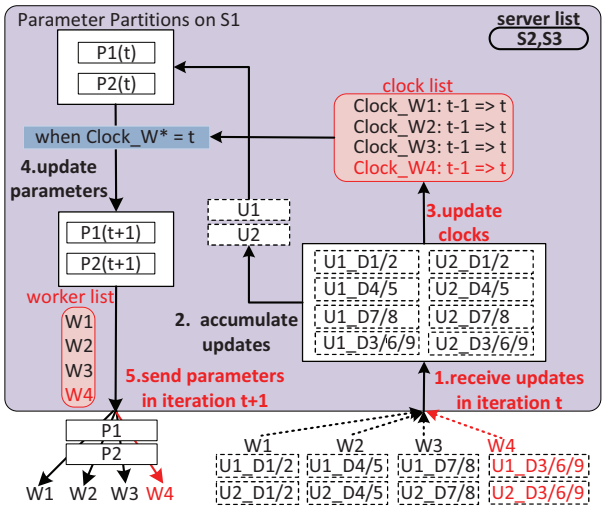


Fig. 9. Dependency between $W4$ and $S1$ is added.

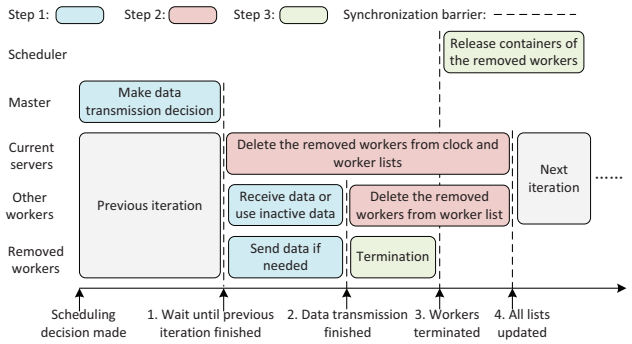


Fig. 10. Three steps for removing workers from a job.

current workers put $W4$ into their worker lists. Data splits $D3$, $D6$, and $D9$ are remotely transmitted to $W4$. Meanwhile, these splits are locally moved to the inactive datasets. Further, the worker lists and clock lists in the current servers are updated to add the dependency between $W4$ and the servers. Figure 9 illustrates the dependency update procedure inside server $S1$ and the updates are shown in red. In specific, $W4$ sends the updates generated from data splits $D3$, $D6$, and $D9$ to $S1$, following which the clock list of $W4$ is updated. The updated parameters are also sent to $W4$.

3.3 Function Calls to Removing Servers or Workers

With EPS, when removing workers from a running job, there are three steps as shown in Figure 10. In the first step, data splits of the removed workers are transmitted to the remaining workers and servers. In the second step, EPS removes dependency between the removed workers and the remaining workers and servers. In the third step, a scheduler terminates the removed workers and releases their containers. Note that removing servers from a running job is done in three similar steps.

The three steps in removing workers are partially overlapped and there are synchronization barriers. At the first step, the master makes data transmission decisions on how to transmit the data splits from the removed workers. The decision-making process is overlapped with the previous iteration executed by all workers and servers. After making the decision and finishing the previous iteration (barrier 1), the removed workers transmit their data splits to the remaining workers. At the second step, the servers delete the removed workers from their clock lists and worker lists to remove the dependency. After the splits are transmitted (barrier 2), the remaining workers delete the removed workers from their worker lists and the removed workers are terminated. At termination (barrier 3), the scheduler releases the containers of the removed workers. When all lists are updated (barrier 4), the job starts the next iteration.

When removing workers, the master makes data transmission decisions using a round-robin algorithm. The algorithm determines the worker to which each data split of the removed servers is transmitted. In specific, it scans through the workers. In each scan, the algorithm checks whether the inactive dataset on the scanned worker contains the data splits of the removed workers. If it does, the removed workers do not transmit the split to the scanned worker. Otherwise, they do. When removing servers, the master

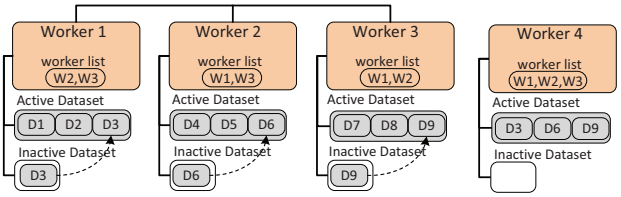


Fig. 11. Data transmission in removing worker $W4$.

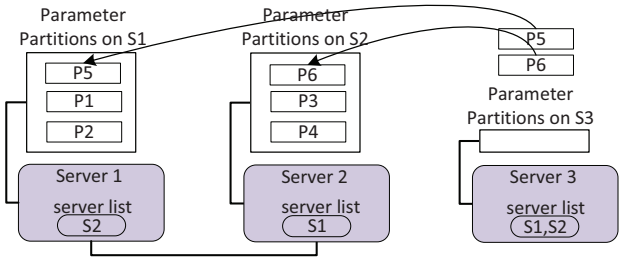


Fig. 12. Parameter transmission in removing server $S3$.

makes parameter transmission decisions using a similar algorithm without the check procedure.

We use two examples to illustrate data and parameter transmissions. Consider the job running with four workers in Figure 8. When a worker ($W4$) is removed from the job, the data transmission procedure is illustrated in Figure 11. Note that $W4$ does not transmit data splits $D3$, $D6$, and $D9$ to workers $W1$, $W2$, and $W3$ because the splits already exist in the inactive datasets of the workers. The workers only need to move the data splits from their inactive datasets to active datasets. After the transmission, workers $W1$, $W2$, and $W3$ delete $W4$ from their worker lists. Besides, the worker lists and clock lists in the servers are updated to remove the dependency between $W4$ and the servers.

When server ($S3$) is removed from the job, Figure 12 shows the parameter transmission where partitions $P5$ and $P6$ are transmitted to servers $S1$ and $S2$, respectively. After the transmission, $S1$ and $S2$ delete $S3$ from their server lists. The server lists and clock lists in the workers are updated to remove the dependency between the workers and $S3$.

4 Scalable Resource Scheduling

The heuristic scheduler contains two scheduling modes: *incoming job scheduling* and *running job scheduling*. Incoming job scheduling is triggered when a new job is submitted. The scheduler calls the removing workers and servers functions in EPS when the available resources cannot meet the resource requirements of the new job. Running job scheduling is triggered when there are available resources but no submitted jobs.

4.1 Incoming Job Scheduling

Algorithm 1 describes the incoming job scheduling mode (lines 1 to 27). In specific, the scheduler first checks the available resources in the cluster (lines 2 to 6). When the available resources cannot meet the resource requirements of the submitted job, it identifies running jobs where workers and servers are removable and determines the number

of workers and servers to be removed (lines 7 to 27). It calls two EPS functions to remove workers and servers. It then allocates resources to the submitted job.

To determine the number of workers (N_w_r) to be removed, the scheduler calculates the ratio (R_submit) of the number of workers to the number of servers in the resource requirements (line 2). It divides available resources into two parts (line 3). The first part is allocated as workers and the second is allocated as servers. The ratio between the two equals R_submit. N_w_r is the number of workers in the resource requirements minus the number of workers that can be allocated using available resources (line 7).

The scheduler selects running jobs where workers and servers are removable based on two rules (lines 10 to 16), 1) the running jobs that have not provided early feedback are not removable, and 2) the running jobs with only one worker and one server are not removable. It determines the number of workers to be removed from each removable job in a round-robin way (lines 17 to 26). The number of servers to be removed is determined similarly.

4.2 Running Job Scheduling

Algorithm 1 describes the running job scheduling mode (lines 29 to 40). The scheduler selects running jobs that new workers and servers can be added (line 33). It iteratively adds one worker and one server to each job in a round-robin way (lines 34 to 40). It monitors the training speed of the job. If the speed drops due to the additional communication overhead, it removes the added worker and server from the running job.

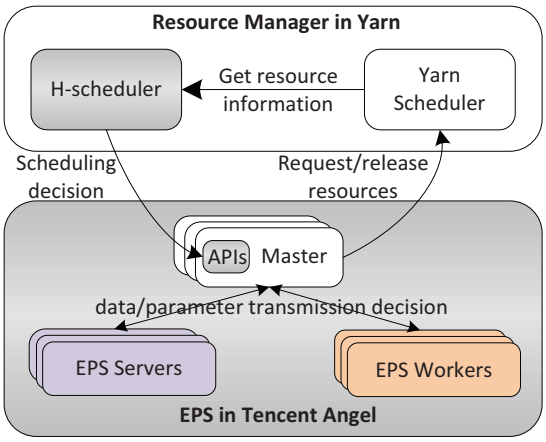


Fig. 13. Implementation of EPS and H-scheduler.

5 Implementation

We implement EPS in Tencent Angel (version 2.0.1) by modifying source files in package `angel.angel-ps.core`. The heuristic scheduler is implemented in Apache Yarn (version 2.8.3) by adding a new component called `H-scheduler`. Figure 13 shows the implementation of EPS and H-scheduler. This project is open-sourced at Github: https://github.com/ibingoogle/EPS-angel-2.0.1.

Algorithm 1 Scalable resource scheduling

1: **/* Incoming Job Scheduling */**
2: Number of workers and servers required by the submitted job: N_w_submit, N_s_submit; Ratio $R\_submit = N\_w\_submit/N\_s\_submit$;
3: Divide available resource into two parts; The ratio of the two parts = R_submit;
4: Number of workers that can be allocated using part one: N_w_ava; The number of servers that can be allocated using part two: N_s_ava;
5: **if** $N\_w\_ava \geqslant N\_w\_submit$ and $N\_s\_ava \geqslant N\_s\_submit$
6: **return**
7: Number of workers to be removed: $N\_w\_r = N\_w\_submit - N\_w\_ava$; Number of servers to be removed: $N\_s\_r = N\_s\_submit - N\_s\_ava$;
8: /* select running jobs where workers and servers are removable */
9: Initialize two lists: $l1 = []$ and $l2 = []$; $l1$ contains jobs whose workers are removable; $l2$ contains jobs whose servers are removable;
10: **for** job J_i in running jobs
11: **if** J_i has not provided early feedback
12: **continue**
13: **if** the number of workers in $J\_i > 1$
14: Put J_i into list $l1$;
15: **if** the number of servers in $J\_i > 1$
16: Put J_i into list $l2$;
17: /* decide the number of workers to be removed from each job */
18: Sort jobs in $l1$ in decreasing order based on the number of workers in each job; Initialize $count = 0$;
19: **while** true
20: **for** removable job J_k in list $l1$
21: increase the number of workers to be removed from J_k;
22: **if** the number of remaining workers in $J\_k == 1$
23: Remove J_k from list $l1$;
24: $count = count + 1$;
25: **if** $count == N\_w\_r$
26: **break** while loop
27: Decide the number of servers to be removed from each job in a similar way;
28:
29: **/* Running Job Scheduling */**
30: The number of workers and servers in cluster $N\_w\_cluster$, $N\_w\_cluster$; Ratio $R\_cluster = N\_w\_cluster/N\_s\_cluster$;
31: Calculate the number of workers and servers that can be allocated using available resources: N_w_add, N_s_add; Their ratio equals $R\_cluster$;
32: /* select running jobs that can are able to add workers/servers */
33: Initialize two lists: $l1$ and $l2$; $l1$ contains jobs that can add workers; $l2$ contains jobs that can add servers; Put running jobs in training phase into $l1$ and $l2$;
34: /* add one worker to each job in a round-robin way */
35: Sort jobs in $l1$ in increasing order based on the number of workers in each job;
36: **for** job J_k in list $l1$
37: Add one worker to J_k; $N\_w\_add = N\_w\_add$ - 1;
38: **if** $N\_w\_add == 0$
39: **break** for loop
40: Add one server to each job similarly;

5.1 Heuristic Scheduler

In package `yarn.server.resourcemanager.scheduler` of Yarn, we add a new class `H-scheduler`. It obtains the amount of available CPU and memory resources in clusters through calling function `getClusterResource` in class `YarnScheduler`. It obtains the amount of CPU and memory required by one worker and server by reading job configuration file `angel-site.xml`. It calculates the number of workers and servers that can be allocated to a submitted job using available resources. It obtains the information of the running jobs from class `RMAppManager` so that it can call APIs in the application master of each running job.

5.2 EPS

In Tencent Angel, each ML job has an application master that runs class `AngelApplicationMaster` in package `core.mast er.yarn.until`. The class communicates with Yarn to request or release resources. We add four APIs in the class: `addServers`, `addWorkers`, `removeServers`, and `removeWorkers`, which take the number of removed

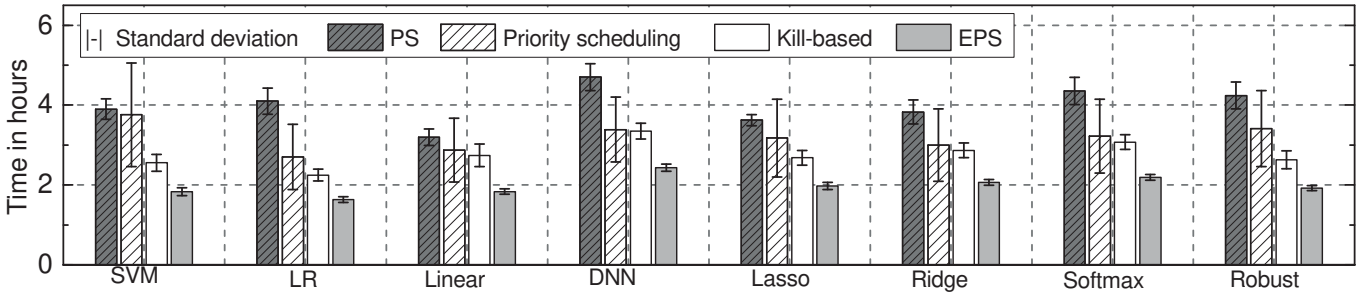


Fig. 14. The training time of eight representative ML models due to four approaches.

or added workers and servers as the input. `H-scheduler` calls APIs to make scheduling decisions.

The master makes parameter transmission decisions using the modified class `AMMatrixMetaManager` in package `core.master.matrixmeta`. The class contains the metadata of parameter partitions in servers. It makes data transmission decisions using the modified class `DaraSpliter` in package `core.master.data`. The class contains the metadata of data splits in workers.

To add or remove servers, we add a new class `PSList` in class `ParameterServer` in servers. `PSList` contains `RPC server` for parameter transmissions. In workers, we modify two classes, `PSAgentMatrixMetaManager` and `ClockCache` in package `core.psagent` so that workers can update the dependency with the added or removed servers. To add or remove workers, we add a new class `WorkerList` in class `Worker` in workers. The class contains `RPC server` for data transmissions. In servers, we modify two classes `PSMatrixMetaManager` and `ClockVectorManager` in package `core.ps` so that servers can update the dependency with the added or removed workers.

6 Evaluation Setup

Testbed and Performance Metrics. We build a multi-job cluster to evaluate the performance of EPS and the heuristic scheduler. The cluster is deployed in a university private cloud with 37 virtual machines, i.e., one master node and 36 slave nodes. Each node is configured with four vCPUs and 16GB memory. All nodes run Ubuntu Server 14.04 with Linux kernel 4.4.0-64. The primary performance metric is ML model training time. We also measure the average early feedback latency and resource utilization.

Workloads. We adopt eight representative ML models used in Tencent Angel: Support Vector Machine (SVM), Logistic Regression (LR), Linear Regression (Linear), Deep Neural Network (DNN), Lasso, Ridge Regression (Ridge), Softmax Regression (Softmax), and Robust Regression (Robust). The dataset used is kdd2012 from LIBSVM datasets [31]. The size kdd2012 dataset is 9.9 GB. It contains $1.4 * 10^9$ non-zero features. Each data point is a training instance derived from search session log messages in Tencent.

For each of the eight models, we use library Hyperopt [10] to generate 75 jobs with different hyper-parameter configurations. The jobs are submitted in an exponential distribution. The resource requirements of each job are eight workers and four servers. The requirements are homogeneous because each job trains the same model. Each worker or server is over-provisioned with 4GB memory and one vCPU. For each job, the model accuracy at the 100th iteration (same as Gandiva [13] and HyperBand [11]) is used as the early feedback. The threshold number of running jobs in the training phase is 11 because of the job resource requirements and the cluster resource availability.

Approaches. We mainly evaluate performance of three approaches in the multi-job setting : EPS with `H-scheduler` (namely EPS), the kill-based approach [21] with `H-scheduler` (namely kill-based), and the default PS (namely PS). Each evaluation runs ten times and we report average results as well as standard deviation.

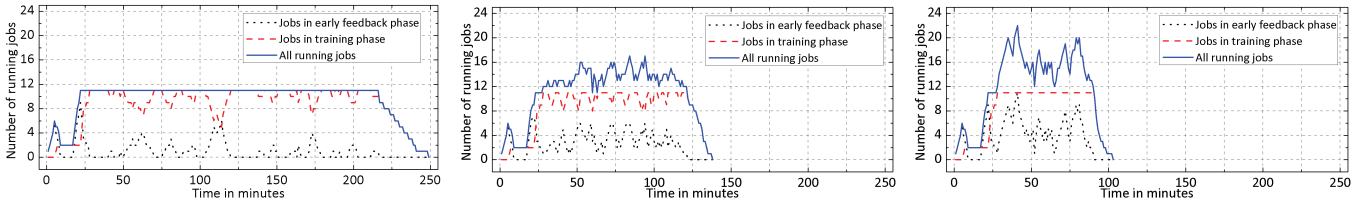
In the default PS, when the threshold is met, all cluster resources are allocated to the running jobs in the training phase and an incoming job with potential better model configurations will be queued. In the kill-based approach, when the threshold is met and there is an incoming job, it will checkpoint and kill the job in the training phase with the worst early feedback, distribute the released resources to the killed job and the incoming job, and run both jobs.

In EPS, when the above scenario occurs, it will leverage the elasticity to remove a subset of workers and servers from the running jobs at runtime and allocate the released resources to the incoming job so as to reduce its early feedback latency. In EPS and the kill-based approaches, when the number of running jobs is larger than the threashold, the one with the worst early feedback is killed.

Further, we extend the evaluation to compare the performance of two popular scheduling approaches: `H-scheduler` and priority scheduling in Litz [32]. We compare the performance of two PS architectures with elasticity: EPS and PS in Litz (namely Litz-PS).

For the comparison between `H-scheduler` and priority scheduling, we measure the training time of eight representative ML models due to EPS with priority scheduling in the multi-job setting, and compare the training time with that due to EPS with `H-scheduler` (namely EPS).

The priority scheduling in Litz simply requests a low-priority ML job to release resources when a high-priority job arrives. In the context of feedback-driven exploration, EPS with priority scheduling removes workers from running jobs in order to provide resources to incoming jobs. Note that the number of workers to be removed is random and the running jobs are randomly selected. Litz leaves the detailed design of the priority scheduling for future work.



(a) Default PS. (b) Kill-based. (c) EPS.

Fig. 15. The number of running jobs during LR training due to three approaches.

For the comparison between EPS and Litz-PS, we submit one RL job to a cluster and measure the average iteration time of the job when the number of nodes available to the job ranges from 1 to 36. The initial number of nodes and workers is set to 6 and 12, respectively.

7 Evaluation

7.1 ML Model Training Time

Figure 14 shows the ML model training time due to the four approaches. EPS significantly outperforms the other three approach. Moreover, the results of the standard deviation identify that EPS has the best robustness among them.

Compared to the default PS, EPS accelerates the model training speed by 1.13x, 1.5x, 74%, 93% 84%, 85%, 98%, and 1.2x for the eight ML models, respectively. The reasons are twofold. First, when the cluster resources are not fully utilized by the running jobs and there are no new jobs submitted yet, the default PS cannot utilize the idle resources. In contrast, EPS can add workers and servers to the running jobs at runtime so as to utilize the idle resources and improve the training speed.

Second, in the default PS, when all cluster resources are allocated to running jobs at the training phase, a new job has to wait in a queue until one running job finishes training and releases allocated resources. If the hyperparameters of the new job are worse than those of all running jobs, the queuing does not really affect ML model training because better ML models will be generated from the running jobs. However, if the hyperparameters of the new job are better than those of some running jobs, the queuing will significantly delay ML model training because the running jobs with worse hyperparameters occupy cluster resources until finished but they will not generate ML models that are better than the one to be generated by the new job. The new job has a long queueing delay. In EPS, a new job can start running shortly after its submission because of the elasticity of EPS and scalable resource scheduling. Thus, if the hyperparameters of the new job are better than those of some running jobs, it will generate a better model ML without queueing delay.

Compared to the kill-based approach, EPS improves the training speed by 39%, 37%, 50%, 38%, 36%, 39%, 40% and 37% for the eight ML models, respectively. The kill-based approach incurs significant overhead in removing all workers and servers from the killed job, adding some of the removed workers and servers back to the job, and adding the left workers and servers to the new job. Its checkpointing and model reloading further prolong the model training.

Compared to the priority scheduling approach, EPS improves the training speed by 1.05x, 65%, 57%, 39%, 61%, 45%, 47% and 78% for the eight ML models, respectively. The reasons are trifold. First, when the cluster resources are not fully utilized by the running jobs and there are no new jobs submitted yet, the priority scheduling cannot utilize the idle resources. Second, the priority scheduling randomly chooses the number of workers to be removed. When the number is too small, the released resources are not sufficient to launch the incoming job so that the priority scheduler has to make an extra call of the removing workers function in EPS, leading to high overhead in resource adjustment. When the number is too large, there would be idle resources after the incoming job is launched. Third, priority scheduling cannot remove servers. This could cause an inappropriate ratio of the number of workers to the number of servers, slowing down the per-iteration execution of a job.

Figure 15 plots the number of running jobs in LR model training. In the default PS, the maximum number of running jobs is capped to 11. When the number of jobs in training phase reaches 11 at the 28_{th} minute (t_a), there is no job running in early feedback phase since submitted jobs are queued. When a job finished at the 41_{st} minute (t_b), a queued job starts running in early feedback phase. Both the kill-based approach and EPS allow to run jobs in the early feedback phase when the number of jobs in the training phase reached 11, by adjusting resource allocations at runtime. For example, jobs are running in early feedback phase between the 28_{th} minute and the 41_{st} minute. However, the number of running jobs in training phase due to the kill-based approach often drops below 11 since the approach kills running jobs to adjust resource allocation. Moreover, the number of running jobs due to EPS is larger than that due to the kill-based approach in some time slots (e.g., from t_a to t_b). EPS is more efficient than the kill-based approach in resource utilization, which leads to faster training speed. Overall, EPS can remove more resources from running jobs in those time slots and use the resources to run more jobs in early feedback phase.

Figure 16 depicts the number of queued jobs in LR model training. The kill-based approach and EPS offer resources to a new job by removing a subset of workers and servers from the running jobs, which result in significantly fewer queued jobs than the default PS. Compared to the kill-based approach, EPS results in fewer queued jobs because it takes less time in removing or adding workers and servers so that a new job starts running more quickly.

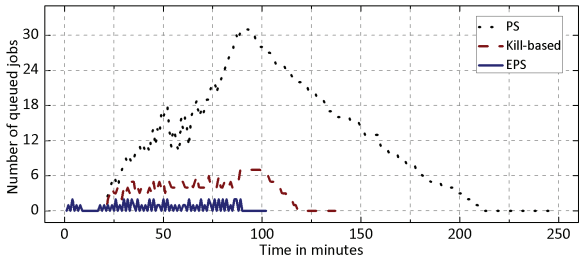


Fig. 16. The number of queued jobs in LR training.

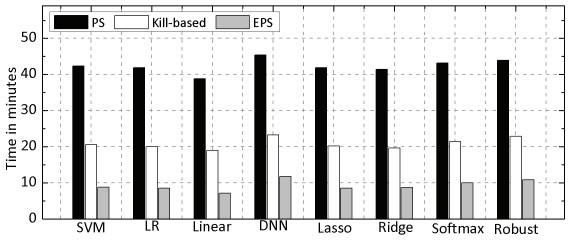


Fig. 17. The average early feedback latency.

7.2 Average Early Feedback Latency

Figure 17 depicts the average early feedback latency due to the three approaches. The latency due to the default PS mainly consists of the queuing delay and the time in the early feedback phase. In the kill-based approach and EPS, a new job has to wait for resources released from the removed workers and servers. Thus, the latency mainly consists of the time spent on removing and adding workers and servers, and the time spent on the early feedback phase. EPS incurs the smallest average early feedback latency since it incurs no queuing delay and the least time in removing or adding workers and servers.

Figure 18 illustrates the early feedback latency at runtime during LR model training. At the beginning, the latency due to the three approaches is similar since the cluster has sufficient resources for new jobs. After the 28_{th} minute, all cluster resources are allocated to the running jobs in the training phase. The latency due to the default PS increases significantly since a new job has to be queued. EPS outperforms the kill-based approach because of its lower overhead in removing or adding workers and servers.

7.3 Resource Utilization

Figure 19 (a)-(c) depict the CPU utilizations and (d)-(e) depict the memory utilizations during LR model training. From the 9_{th} minute to the 23_{rd} minute, the utilizations

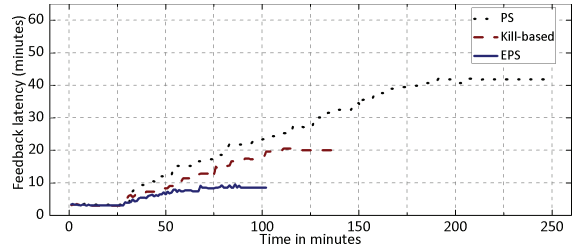


Fig. 18. The early feedback latency in LR training.

due to the default PS are much lower than those due to the kill-based approach and EPS. The cause is that there is no job submitted in the period so that the default PS cannot utilize the idle resources, while the other two approaches can utilize the idle resources by adding workers and servers to the running jobs at runtime. After the 23_{rd} minute, the utilizations due to the two approaches are slightly higher than that due to the default PS. The reason is that the two approaches use the over-provisioned memory space when the input data and parameters on the removed workers and servers are transmitted to and stored in the other workers.

In Figure 19 (b) and Figure 19(e), there are moments (e.g., t1 and t2) when the CPU and memory utilizations due to the kill-based approach drop significantly and then rebound. To remove resources from a running job, the kill-based approach has to first release all workers and servers from the job, and then re-distribute the resources to the job and a new job. This process significantly affects the CPU and memory utilizations.

7.4 Overhead Analysis

Table 2 presents the overhead in training of the eight ML models due to the kill-based approach and EPS. The overhead is measured as the ratio of the total time spent on removing and adding workers and servers to the total runtime of all jobs. The kill-based approach incurs much higher overhead than EPS. The overhead due to EPS is negligible compared to its performance gain.

TABLE 2
The overhead of the kill-based approach and EPS.

	SVM	LR	Linear	DNN
Kill	22.3%	21%	29.2%	26.1%
EPS	5.4%	4.9%	6.2%	5.6%

	Lasso	Ridge	Softmax	Robust
Kill	24.1%	24.5%	25.7%	23.5%
EPS	5.1%	5.2%	5.9%	5.3%

We take a closer look at the overhead analysis. We submit a LR job to the cluster and allocate three workers and three servers to the job. We measure the overhead when one worker and one server are removed from the job by the kill-based approach and EPS, respectively. Recall that both approaches take three steps in removing one worker and one server. Figure 20 plots the time spent on the three individual steps by the two approaches. Overall, the kill-based approach incurs much more significant overhead than EPS, as it spends non-negligible time on ML model checkpoint in Angel, job termination and resource release in Yarn, and job initialization and resource re-allocation in Yarn.

The overhead of EPS mainly comes from data and parameter transmissions as well as resource release. There are no job checkpoint, initialization and termination overheads. The amount of resources released or allocated in EPS is much less than that in the kill-based approach. For example, when a subset of workers and servers are removed from a job, EPS requests Yarn to release the resources allocated to the removed workers and servers only. In contrast, the kill-based approach requires Yarn to release the resources of all workers and servers allocated to the job. Furthermore, the

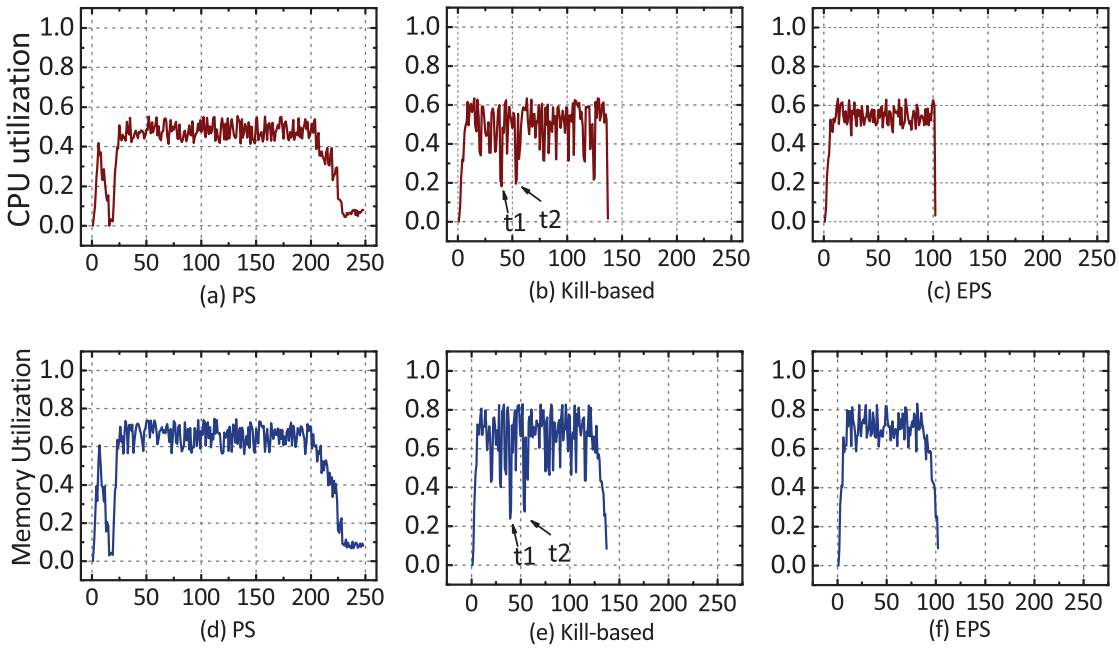


Fig. 19. CPU and memory utilizations during LR training.

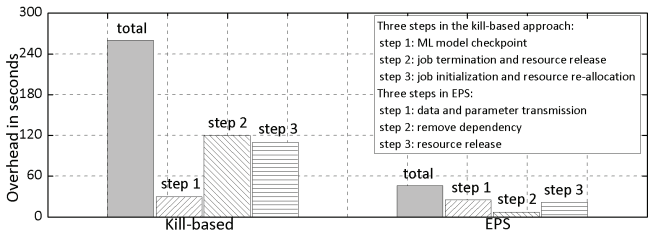
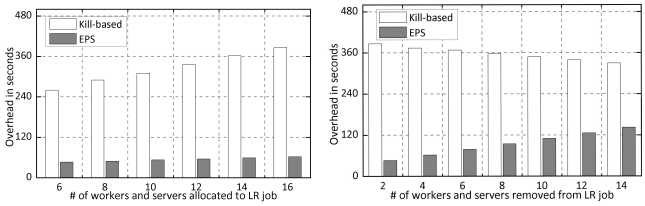


Fig. 20. Overhead in removing one worker & server.



(a) Overhead of removing one worker and one server. (b) Overhead of removing multiple workers and servers.

Fig. 21. Overhead of removing workers and servers.

three steps in EPS are partially overlapped, while they are executed sequentially in the kill-based approach.

We also measure the overhead of removing one worker and one server when different numbers of workers/servers are allocated to job 3 in the submission. In specific, the number of allocated workers/servers ranges from 6 to 16. The results in Figure 21(a) show that, when the number of workers/servers in job 3 increases, the overhead due to the kill-based approach increases significantly since Yarn spends more time on resource release/re-allocation. The overhead due to EPS increases sightly because more workers/servers are involved in data/parameter transmission.

We further measure the overhead of removing multiple workers and servers at the same time. In specific, we allocate 8 workers and 8 servers to job 3 in the submission and remove different numbers of workers/servers from the job. In Figure 21(b), when the number of removed workers/servers increases, the overhead due to EPS increases since Yarn spends more time releasing resources allocated to the removed workers/servers. The overhead due to the kill-based approach decreases because job 3 re-allocates fewer number of workers/server in job initialization. However, the overhead due to EPS is much lower than that due to the kill-based approach.

7.5 Elasticity Comparison

Figure 22 plots the average iteration time of a RL job due to two PS architectures when the number of nodes available to the job ranges from 1 to 36. The result shows that EPS outperforms Litz-PS when the cluster scales up or scales down to a certain degree. When the number of nodes is small (< 2), Litz-PS distributes 12 workers to these nodes so that each node is overloaded, leading to resource contention and extra communication overhead. In contrast, EPS can avoid the contention and overhead by removing a certain number of workers. When the number of nodes becomes large (> 12), the average iteration time due to Litz-PS does not decrease anymore. The reason is that Litz-PS distributes 12 workers to 12 nodes at most and it cannot utilize the resources in the other nodes. In contrast, EPS can utilize all resources in the cluster by adding more workers.

7.6 Discussions

User transparency. The runtime resource adjustment in EPS can be simply utilized by calling the four APIs using the scheduler. To apply EPS to ML model training, users only need to employ `H-scheduler` without changing ML model source code. Users can also customize their own schedulers to leverage EPS.

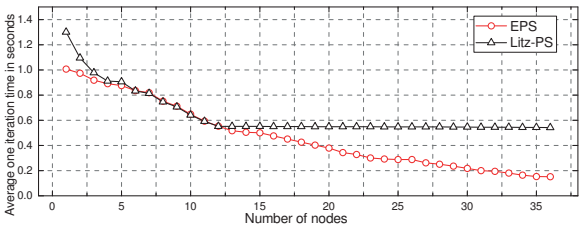


Fig. 22. The average iteration time of a RL job.

PS compatibility. EPS implementation does not change any mechanisms in the current PS. It can be implemented as a PS-compatible architecture in existing distributed ML frameworks so that elasticity can be introduced as a selectively enabled feature in these frameworks. EPS can be extended to other frameworks, such as TensorFlow.

Data transmission. EPS assumes that input data is stored in a local file system. Thus, when a worker is removed, the input data is transmitted from the removed worker to the other workers. Recently, remote file system is also used in practice. In this case, the input data should be transmitted from the remote file system to the other workers. We will extend EPS to support remote file system in the future.

Fairness. The heuristic scheduler does not schedule an incoming job resources based on fairness because the incoming job has high priority than the running jobs. The scheduling of running jobs follows the principle of fairness because it adds resources to each running job in a round-robin manner.

Model convergence. EPS adjusts the resources allocated to a job so that it only impacts the execution time of one iteration. But, it does not impact the convergence speed of the iteration.

In-memory storage. Servers in PS store model parameters in memory which is similar to in-memory database (e.g., Redis [33]). However, the resource adjustment mechanism in these databases only supports adding or removing machines instead of containers. The mechanism cannot be easily integrated into the runtime adjustment of servers.

Dataset. EPS is only aware of dataset size and conducts data repartition based on the size and the number of workers. It is agnostic to the number of features in the dataset and also does not impact the model convergence. Thus, EPS is general approach for different types/sizes of data.

Multiple distributed clusters. The ML job using EPS is agnostic to the machines it runs on. These machines can be either located in one cluster or distributed across multiple clusters as long as data can be transmitted between them. Therefore, EPS is not limited to a single cluster.

In a large-scale cloud environment with multiple distributed clusters, H-scheduler should be extended to schedule workers and servers of a ML job to the same cluster in order to avoid communication overhead. Note that the open-source Apache Yarn is responsible for resource management in such a large-scale cloud environment.

Although the presented EPS works for multiple distributed clusters, it can be further optimized when workers or servers are located in the different clusters. A straightforward optimization approach can avoid cross-cluster communication overhead in EPS from two aspects: (1) when removing a worker from a cluster, the data on the removed worker is only transmitted to the running workers in the same cluster. (2) when adding a worker to a cluster, only the running workers in the same cluster transmit their data to the added worker. Parameter transmission is conducted similarly. However, this approach would incur imbalanced data/parameter distribution across all workers/servers. Therefore, a more sophisticated transmission algorithm should be developed in EPS to achieve the balance between data/parameter distribution and cross-cluster communication.

8 Related Work

PS architecture optimization. GeePS [15] is a PS specialized for scaling deep learning applications in distributed GPUs. Poseidon [14] uses wait-free backpropagation that overlaps backward computation with gradient communication. P3 [16], TicTac [34], and Byte Scheduler [35] change the transmission order of different DNN layers in order to reduce the communication overhead in PS architecture. Gaia [17] is an efficient ML synchronization model for communication between parameter servers across datacenters. Work [36] introduces a distributed GPU hierarchical parameter server for massive scale deep learning ads systems. PHub [37] is a multi-tenant and rack-scale PS design for cloud-based distributed deep neural network training. DynSSP [19] introduces heterogeneity-aware dynamic learning for jobs based on gradient descent so as to improve ML job training speed. Parallax [38] integrates PS with AllReduce architecture to optimize the amount of data transmission. LAG [39] proposes lazily aggregated gradient that adaptively skips the gradient calculations to reduce communication and computation in PS. Work [40] proposes a general distributed compressed SGD with momentum in PS. PS2 [41] is a PS architecture that integrates Spark platoform [28] without hacking the core of Spark. FlexPS [20] focuses on specific ML jobs whose input data size can be dynamically changed at runtime and reduces the training time. It is inapplicable to general ML jobs with static input data size. It also cannot adjust the number of servers at runtime. Complementary to these efforts, EPS augments the PS architecture with elasticity that allows to adjust the number of workers and servers at runtime.

PS based on distributed recommendation systems. KunPeng [22] is a parameter server based distributed learning system that supports recommendation scenarios in Alibaba. Study [23] introduces a simple parameter server based federated learning approach for recommender systems to improve on personalization. It discusses closely-related meta-learning algorithms. Recent work [24] proposes a distributed hierarchical GPU parameter server for massive-scale deep learning ads systems. The GPU parameter server builds a distributed hash table across multiple GPUs and performs direct inter-GPU communications to eliminate the CPU-GPU data transferring overhead. EPS can be integrated into these systems to improve or enable the elasticity.

Elasticity in distributed ML training. Work [42] introduces worker elasticity to AllReduce architecture. Proteus [43] introduces redundant servers in PS and is designed for dynamic resource markets (e.g., AWS). When a redundant

server is removed, Proteus does not maintain the dependencies between workers and servers. Work [44] adjusts the number of tasks within a Yarn container. It does not support adjusting the number of workers and servers in PS. Meta-Daraflows [45] can efficiently execute exploratory workflows. However, it is not designed for concrete data processing workflows such as distributed ML training. PSLD [46] proposes a dynamic PS load distribution algorithm so as to mitigate PS straggler issues and accelerate distributed model training in the PS architecture.

Litz [32] implements worker (named executor in its literature) elasticity in PS from scratch based on event-driven programming model. However, its worker elasticity is only limited to moving workers between different nodes and it cannot adjust the number of workers at runtime. Also, Litz does not support server elasticity.

Multi-job ML cluster optimization. There are innovative efforts on system support for accelerating distributed ML training. Tiresias [47] is a GPU cluster resource manager that minimizes the job completion time based on the characteristic study of Microsoft cluster Philly [48]. FfDL [49] is a cloud-hosted and multi-tenant dependable distributed DL platform used to train DL models at IBM. Litz [32] proposes priority scheduling in that a low-priority ML job can simply use less amount of resource by moving its workers to a fewer number of machines so that a higher-priority job can use the released resource.

Recently, there are a few innovations that aim to speedup feedback-driven exploration. Population based training [50] uses information from the jobs in the training phase to refine the hyperparameters so as to quickly choose the best set of them. This effort focuses on deep learning model training. Hyperdrive [12] proposes to accelerate feedback-driven exploration by dynamically classifying the configuration of hyperparameters into three categories: promising, opportunistic, and poor. It terminates jobs with poor configurations, and prioritizes promising jobs with more GPU resource. SLAQ [25] uses a dedicated cluster to run distributed ML jobs in the early feedback phase. The goal is to maximize the average model accuracy in the phase. It dynamically allocates CPU resource at runtime based on the job resource demand and intermediate model accuracy. Gandiva [13] accelerates DL model training in feedback-driven exploration by dynamically changing GPU usage modes of distributed deep learning jobs at runtime. It can reduce early feedback latency and improve the scheduling efficiency in GPU clusters. Pytorch [26] develops worker elasticity in AllReduce architecture so that the number of allocated workers is adjustable at runtime. However, these approaches do not support PS architecture.

Optimus [21] minimizes the makespan and the average job completion time of multiple jobs in PS architecture. It designs a novel resource-performance model and proposes a scheduling scheme to dynamically adjust the number of allocated workers and servers. It employs the kill-based approach to implement the resource adjustment at runtime, which results in significant overheads. Instead, EPS enables elasticity by adjusting resource allocations at runtime. It is lightweight and efficient in resource utilization.

Preemption technique. Recent efforts leverage lightweight virtualization (e.g., Docker [51]) to make tasks preemptable in cluster scheduling of batch jobs [52], [53]. In these efforts, when latency-critical jobs are submitted to clusters, they preempt tasks of best-effort long jobs so that the submitted jobs can be scheduled to avoid any queuing delays. The preemption idea can be used to adjust the allocated resources of distributed ML jobs with PS architecture. For example, we can preempt workers and servers of running jobs to offer resources to incoming jobs. This idea works when incoming jobs are configured with wrong hyperparameters so that the preempted jobs can resume execution soon after incoming jobs are killed by users. But when incoming jobs are configured with right hyperparameters and run for several hours to days, the preempted jobs are blocked for a long time.

Cloud auto-scaling. Web application providers have been migrating their applications to cloud data centers, attracted by the emerging cloud computing paradigm [54]. One of the appealing features of the cloud is elasticity that autonomously and dynamically provisions and de-provisions a set of resources to cater to fluctuant application workloads [54], [55], [56].

Note that the elasticity in cloud auto-scaling is defined as adjusting the size of cluster resources to cater to fluctuant web application jobs, and thus it can be regarded as cluster-level resource elasticity. In contrast, the elasticity in EPS is a job-level resource elasticity, which adjusts the size of resources used by a ML job. The design and development of EPS and H-scheduler assume that the size of cluster resources is fixed (i.e., no cluster-level resource elasticity). The combination of the two-level elasticities can be a potential future research direction.

9 Conclusion

This paper presents EPS, a lightweight and user-transparent parameter server that accelerates feedback-driven exploration for distributed ML training. EPS is designed with three new components and four function calls, which enable the elasticity in resource adjustments to a distributed ML job at runtime. To leverage EPS, we develop a heuristic scheduler that offers scalable resource scheduling for multiple ML training jobs in a distributed cluster. We implement EPS in Tencent Angel and the scheduler in Apache Yarn. Experimental results show that EPS achieves up to 1.5x and 50% improvement on the ML training speed compared to PS and the kill-based approach deployed in Optimus, respectively, due to its built-in elasticity and lightweight in runtime resource adjustment.

In future work, we will extend EPS to other ML frameworks such as TensorFlow and extend H-scheduler to other applications with elastic mechanisms.

References

[1] J. Jiang, L. Yu, J. Jiang, Y. Liu, and B. Cui, "Angel: a new large-scale machine learning system," *National Science Review*, vol. 5, no. 2, pp. 216–236, 2018.

[2] J. Jiang, B. Cui, C. Zhang, and F. Fu, "Dimboost: Boosting gradient boosting decision tree to higher dimensions," in *Proc. of ACM SIGMOD*, 2018.

[3] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "Tensorflow: a system for large-scale machine learning." in *Proc. of USENIX OSDI*, 2016.

[4] H. Cui, A. Tumanov, J. Wei, L. Xu, W. Dai, J. Haber-Kucharsky, Q. Ho, G. R. Ganger, P. B. Gibbons, G. A. Gibson *et al.*, "Exploiting iterative-ness for parallel ml computations," in *Proc. of ACM SoCC*, 2014.

[5] E. P. Xing, Q. Ho, W. Dai, J. K. Kim, J. Wei, S. Lee, X. Zheng, P. Xie, A. Kumar, and Y. Yu, "Petuum: A new platform for distributed machine learning on big data," *IEEE Transactions on Big Data*, vol. 1, no. 2, pp. 49–67, 2015.

[6] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang, "Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems," *arXiv preprint arXiv:1512.01274*, 2015.

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

[8] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. H. Katz, S. Shenker, and I. Stoica, "Mesos: A platform for fine-grained resource sharing in the data center." in *Proc. of USENIX NSDI*, 2011.

[9] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth *et al.*, "Apache hadoop yarn: Yet another resource negotiator," in *Proc. of ACM SoCC*, 2013.

[10] J. Bergstra, B. Komer, C. Eliasmith, D. Yamins, and D. D. Cox, "Hyperopt: a python library for model selection and hyperparameter optimization," *Computational Science & Discovery*, vol. 8, no. 1, p. 014008, 2015.

[11] L. Li, K. Jamieson, G. DeSalvo, A. Rostamizadeh, and A. Talwalkar, "Hyperband: A novel bandit-based approach to hyperparameter optimization," *The Journal of Machine Learning Research*, vol. 18, no. 1, pp. 6765–6816, 2017.

[12] J. Rasley, Y. He, F. Yan, O. Ruwase, and R. Fonseca, "Hyperdrive: Exploring hyperparameters with pop scheduling," in *Proc. of ACM/IFIP/USENIX Middleware*, 2017.

[13] W. Xiao, R. Bhardwaj, R. Ramjee, M. Sivathanu, N. Kwatra, Z. Han, P. Patel, X. Peng, H. Zhao, Q. Zhang *et al.*, "Gandiva: Introspective cluster scheduling for deep learning," in *Proc. of USENIX OSDI*, 2018.

[14] H. Zhang, Z. Zheng, S. Xu, W. Dai, Q. Ho, X. Liang, Z. Hu, J. Wei, P. Xie, and E. P. Xing, "Poseidon: An efficient communication architecture for distributed deep learning on gpu clusters," in *Proc. of USENIX ATC*, 2017.

[15] H. Cui, H. Zhang, G. R. Ganger, P. B. Gibbons, and E. P. Xing, "Geeps: Scalable deep learning on distributed gpus with a gpu-specialized parameter server," in *Proc. of EuroSys*, 2016.

[16] A. Jayarajan, J. Wei, G. Gibson, A. Fedorova, and G. Pekhimenko, "Priority-based parameter propagation for distributed dnn training," in *Proc. of SysML*, 2019.

[17] K. Hsieh, A. Harlap, N. Vijaykumar, D. Konomis, G. R. Ganger, P. B. Gibbons, and O. Mutlu, "Gaia: Geo-distributed machine learning approaching lan speeds," in *Proc. of USENIX NSDI*, 2017.

[18] S. Wang, A. Pi, and X. Zhou, "Scalable distributed dl training: Batching communication and computation," in *Proc. of AAAI*, 2019.

[19] J. Jiang, B. Cui, C. Zhang, and L. Yu, "Heterogeneity-aware distributed parameter servers," in *Proc. of ACM SIGMOD*, 2017.

[20] Y. Huang, T. Jin, Y. Wu, Z. Cai, X. Yan, F. Yang, J. Li, Y. Guo, and J. Cheng, "Flexps: Flexible parallelism control in parameter server architecture," in *Proc. of VLDB*, 2018.

[21] Y. Peng, Y. Bao, Y. Chen, C. Wu, and C. Guo, "Optimus: an efficient dynamic resource scheduler for deep learning clusters," in *Proc. of EuroSys*, 2018.

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

[23] A. Jalalirad, M. Scavuzzo, C. Capota, and M. Sprague, "A simple and efficient federated recommender system," in *Proc. of IEEE/ACM DBCAT*, 2019.

[24] W. Zhao, D. Xie, R. Jia, Y. Qian, R. Ding, M. Sun, and P. Li, "Distributed hierarchical gpu parameter server for massive scale deep learning ads systems," in *Proc. of MLSys*, 2020.

[25] H. Zhang, L. Stafman, A. Or, and M. J. Freedman, "Slaq: quality-driven scheduling for distributed machine learning," in *Proc. of ACM SoCC*, 2017.

[26] "Pytorch elastic," https://github.com/pytorch/elastic.

[27] K. Nguyen, L. Fang, G. Xu, B. Demsky, S. Lu, S. Alamian, and O. Mutlu, "Yak: A high-performance big-data-friendly garbage collector," in *Proc. of USENIX OSDI*, 2016.

[28] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proc. of USENIX NSDI*, 2012.

[29] S. Wang, W. Chen, A. Pi, and X. Zhou, "Aggressive synchronization with partial processing for iterative ml jobs on clusters," in *Proc. of ACM/IFIP Middleware*, 2018.

[30] T. M. Chilimbi, Y. Suzue, J. Apacible, and K. Kalyanaraman, "Project adam: Building an efficient and scalable deep learning training system." in *Proc. of USENIX OSDI*, 2014.

[31] C.-C. Chang and C.-J. Lin, "Libsvm: A library for support vector machines," *ACM transactions on intelligent systems and technology*, vol. 2, no. 3, pp. 1–27, 2011.

[32] A. Qiao, A. Aghayev, W. Yu, H. Chen, Q. Ho, G. A. Gibson, and E. P. Xing, "Litz: Elastic framework for high-performance distributed machine learning," in *Proc. of USENIX ATC*, 2018.

[33] "Redis," https://https://redis.io.

[34] S. H. Hashemi, S. A. Jyothi, and R. H. Campbell, "Tictac: Accelerating distributed deep learning with communication scheduling," in *Proc. of SysML*, 2019.

[35] Y. Peng, Y. Zhu, Y. Chen, Y. Bao, B. Yi, C. Lan, C. Wu, and C. Guo, "A generic communication scheduler for distributed dnn training acceleration," in *Proc. of ACM SOSP*, 2019.

[36] W. Zhao, D. Xie, R. Jia, Y. Qian, R. Ding, M. Sun, and P. Li, "Distributed hierarchical gpu parameter server for massive scale deep learning ads systems," in *Proc. of MLSys*, 2020.

[37] L. Luo, J. Nelson, L. Ceze, A. Phanishayee, and A. Krishnamurthy, "Parameter hub: a rack-scale parameter server for distributed deep neural network training," in *Proc. of ACM SoCC*, 2018.

[38] S. Kim, G.-I. Yu, H. Park, S. Cho, E. Jeong, H. Ha, S. Lee, J. S. Jeong, and B.-G. Chun, "Parallax: Sparsity-aware data parallel training of deep neural networks," in *Proc. of EuroSys*, 2019.

[39] T. Chen, G. Giannakis, T. Sun, and W. Yin, "Lag: Lazily aggregated gradient for communication-efficient distributed learning," in *Proc. of NeurIPS*, 2018.

[40] S. Zheng, Z. Huang, and J. T. Kwok, "Communication-efficient distributed blockwise momentum sgd with error-feedback," in *Proc. of NeurIPS*, 2019.

[41] Z. Zhang, B. Cui, Y. Shao, L. Yu, J. Jiang, and X. Miao, "Ps2: Parameter server on spark," in *Proc. of ACM SIGMOD*, 2019.

[42] A. Or, H. Zhang, and M. Freedman, "Resource elasticity in distributed deep learning," in *Proc. of MLSys*, 2020.

[43] A. Harlap, A. Tumanov, A. Chung, G. R. Ganger, and P. B. Gibbons, "Proteus: agile ml elasticity through tiered reliability in dynamic resource markets," in *Proc. of EuroSys*, 2017.

[44] B. Huang, M. Boehm, Y. Tian, B. Reinwald, S. Tatikonda, and F. R. Reiss, "Resource elasticity for large-scale machine learning," in *Proc. of ACM SIGMOD*, 2015.

[45] R. Castro Fernandez, W. Culhane, P. Watcharapichat, M. Weidlich, V. Lopez Morales, and P. Pietzuch, "Meta-dataflows: Efficient exploratory dataflow jobs," in *Proc. of ACM SIGMOD*, 2018.

[46] Y. Chen, Y. Peng, Y. Bao, C. Wu, Y. Zhu, and C. Guo, "Elastic parameter server load distribution in deep learning clusters," in *Proc. of ACM SoCC*, 2020.

[47] J. Gu, M. Chowdhury, K. G. Shin, Y. Zhu, M. Jeon, J. Qian, H. Liu, and C. Guo, "Tiresias: A gpu cluster manager for distributed deep learning," in *Proc. of USENIX NSDI*, 2019.

[48] M. Jeon, S. Venkataraman, A. Phanishayee, J. Qian, W. Xiao, and F. Yang, "Analysis of large-scale multi-tenant gpu clusters for dnn training workloads," in *Proc. of USENIX ATC*, 2019.

[49] K. Jayaram, V. Muthusamy, P. Dube, V. Ishakian, C. Wang, B. Herta, S. Boag, D. Arroyo, A. Tantawi, A. Verma *et al.*, "Ffdl: A flexible multi-tenant deep learning platform," in *Proc. of ACM/IFIP Middleware*, 2019.

[50] M. Jaderberg, V. Dalibard, S. Osindero, W. M. Czarnecki, J. Donahue, A. Razavi, O. Vinyals, T. Green, I. Dunning, K. Simonyan *et al.*, "Population based training of neural networks," *arXiv preprint arXiv:1711.09846*, 2017.

[51] C. Boettiger, "An introduction to docker for reproducible research," *ACM SIGOPS Operating Systems Review*, vol. 49, no. 1, pp. 71–79, 2015.

[52] W. Chen, J. Rao, and X. Zhou, "Preemptive, low latency datacenter scheduling via lightweight virtualization," in *Proc. of USENIX ATC*, 2017.

[53] P. Delgado, D. Didona, F. Dinu, and W. Zwaenepoel, "Kairos: Preemptive data center scheduling without runtime estimates," in *Proc. of ACM SoCC*, 2018.
[54] C. Qu, R. N. Calheiros, and R. Buyya, "Auto-scaling web applications in clouds: A taxonomy and survey," *ACM Computing Surveys*, vol. 51, no. 4, pp. 1–33, 2018.
[55] ——, "A reliable and cost-efficient auto-scaling system for web applications using heterogeneous spot instances," *Journal of Network and Computer Applications*, vol. 65, pp. 167–180, 2016.
[56] M. Ghobaei-Arani, S. Jabbehdari, and M. A. Pourmina, "An autonomic resource provisioning approach for service-based cloud applications: A hybrid approach," *Future Generation Computer Systems*, vol. 78, pp. 191–210, 2018.

Shaoqi Wang received the BS degree in engineering from Anhui Normal University in 2012. He received the MS degree in engineering from the University of Science and Technology of China in 2015. He obtained Ph.D. degree in Computer Science from the University of Colorado, Colorado Springs in 2020. His research interests include big data processing, distributed machine learning systems, and deep learning. He is a student member of the IEEE.

Aidi Pi received his B.S. and M.S degree in Computer Science from Tongji University in 2015 and 2016, respectively. He obtained Ph.D. in Computer Science from the University of Colorado, Colorado Springs in 2021. His research interests include distributed systems, troubleshooting, cloud computing. He is a student member of the IEEE.

Xiaobo Zhou obtained the BS, MS, and PhD degrees in Computer Science from Nanjing University, in 1994, 1997, and 2000, respectively. Currently he is a professor of the Department of Computer Science, University of Colorado, Colorado Springs. His research lies in distributed systems, Cloud computing and datacenters, data parallel and distributed processing, autonomic and sustainable computing. He was a recipient of the NSF CAREER Award in 2009. He is a senior member of the IEEE.