Journal of Network and Computer Applications 158 (2020) 102590

Contents lists available at ScienceDirect

Journal of Network and Computer Applications

journal homepage: www.elsevier.com/locate/jnca

JPAS: Job-progress-aware flow scheduling for deep learning clusters

Pan Zhou [a], Xinshu He [a], Shouxi Luo [b], Hongfang Yu [a,c,*], Gang Sun [a,**]

[a] University of Electronic Science and Technology of China, Chengdu, PR China
[b] Southwest Jiaotong University, Chengdu, PR China
[c] Peng Cheng Laboratory, Shenzhen, PR China

ARTICLE INFO

Keywords:
Machine learning
Deep learning
Flow scheduling
Job progress aware

ABSTRACT

Deep learning (DL) is an increasingly important tool for large-scale data analytics and DL workloads are also common in today's production clusters due to the increasing number of deep-learning-driven services (e.g., online search and speech recognition). To handle ever-growing training datasets, it is common to conduct distributed DL (DDL) training to leverage multiple machines in parallel. Training DL models in parallel can incur significant bandwidth contention on shared clusters. As a result, the network is a well-known bottleneck for distributed training. Efficient network scheduling is essential for maximizing the performance of DL training. DL training is feedback-driven exploration (e.g., hyper-parameter tuning, model structure optimization), which requires multiple retrainings of deep learning models that differ in terms of their configuration. The information at the early stage of each retraining can facilitate the direct search for high-quality models. Thus, reducing the early-stage time can accelerate the exploration of DL training. In this paper, we propose JPAS, which is a flow scheduling system for DDL training jobs that aims at reducing the early-stage time. JPAS uses a simple greedy mechanism to periodically order all DDL jobs. Each host machine sets priorities for its flows using the corresponding job order and offloads the flow scheduling and rate allocation to the underlying priority-enabled network. We evaluate JPAS over a real testbed that is composed of 13 servers and a commodity switch. The evaluation results demonstrate that JPAS can reduce the time to reach 90% or 95% of the converged accuracy by up to 38%. Hence, JPAS can remarkably reduce the early-stage time and thus accelerate the search for high-quality models.

1. Introduction

Deep learning (DL) is becoming increasingly popular and is realizing substantial success in various domains, such as computer vision (Wu and ShenAnton Van Den Hengel, 2019), image processing (Zeng et al., 2019) (Zhang et al., 2018), speech recognition (Heet al., 2019). The increasing volume of big data and the increasing scale of training models (e.g., deep neural networks) significantly improve the learning performance but remarkably increase training time. To deal with ever-growing training datasets and large-scale models, researchers have designed distributed algorithms, along with flexible platforms such as TensorFlow (Agarwal, 2019) and MXNet (Li et al., 2019), for conducting efficient model training in parallel.

Many leading companies have built GPU clusters on which to conduct distributed model training over large datasets for their AI-driven services (Hazelwood et al., 2018) (Jeon et al., 2018). In addition, the clusters are shared by many users to satisfy the rising number of distributed DL (DDL) jobs (Amazon2 ElasticUs.) (U-Accelerated Microsoft) (U on Google Cloud. http). According to the traces of a Microsoft cluster, the number of DL jobs increases 10.5X each year[10]. Under the increasing workload, network competition among DDL jobs significantly impacts the efficiency of model training. The following three key characteristics of DDL jobs pose new challenges to network resource management.

DDL training is communication-intensive. The most common option for DDL training is data parallelism, in which the training dataset is divided into equal-sized parts to feed workers, and each worker occupies a GPU and works on its local copy of the DL model. The workers communicate with each other once per iteration to synchronize the model updates from other workers. After the synchronization, the workers will start the next iteration. The model synchronization generates a large amount of network traffic, which increases with the number of workers and the scale of the models. Production DL models range from a few hundreds of megabytes to a few gigabytes. Larger model sizes correspond to heavier communication overhead and longer

* Corresponding author. University of Electronic Science and Technology of China, Chengdu, PR China.
** **Corresponding author.
E-mail address: yuhfnetworklab@gmail.com (H. Yu).

https://doi.org/10.1016/j.jnca.2020.102590
Received 12 April 2019; Received in revised form 11 February 2020; Accepted 3 March 2020
Available online 11 March 2020

synchronization time per iteration in distributed training.

DL training is typically iterative with diminishing returns. The training is time-consuming and typically requires multiple passes over large datasets (a pass is called an epoch). The training generates a low-accuracy model initially and improves the model's accuracy through a sequence of training iterations until it converges. The accuracy improvement diminishes as additional iterations are completed. In addition, a small number of iterations at the beginning can reach a high accuracy while many remaining iterations are spent on further improving the accuracy until convergence. For example, in Microsoft's cluster, approximately 75% of jobs reach within 0.1% of their best accuracy in 40% of the training epochs (Jeon et al., 2018).

Training of DL models is feedback-driven exploration. Training a DL model is not a one-time effort and often works in an exploratory manner. ML practitioners retrain their models repeatedly to preprocess datasets (Wang et al., 2018), tune hyper-parameters (Riquelme and TuckerJasper Snoek, 2018), and adjust model structures (Carbinet al., 2019). The objective of retraining is to obtain a final model that has the highest accuracy. Since each retraining is time-consuming, it is desirable to explore the trade-off between accuracy and training time. ML practitioners and automatic machine learning systems (AutoMLs) prefer to use information (e.g., training or validation accuracy curve) from the early training stage to predict the final performance of the model configuration (e.g., the structure of the model and the learning rate) and to make the next search decisions rather than wait a long time to train a model to convergence, which can significantly reduce the time cost of the exploratory process.

However, before training, the exact number of epochs that are required to predict the performance with high confidence is unknown (Jeon et al., 2018). Practitioners often submit DL training jobs using more epochs than necessary to obtain high confidence predictions of the final performance. Practitioners typically choose to manually monitor the training accuracy and stop the training when the accuracy curves are sufficient for predicting the final performance with high confidence (Gu et al., 2019) (Xiao et al., 2018). However, the training could take hours or even days, and it is impractical for practitioners to monitor the training accuracy in real time and to stop the training timely (Zhang et al., 2017). As a result, jobs that have received high-confidence predictions of final performance (**later-stage** jobs) are still running in the cluster and cost the same amounts of computational and communicational resources while making only marginal improvements in the model quality (Jeon et al., 2018). Since the cluster is shared by multiple DL jobs that have been submitted by practitioners or AutoMLs, the later-stage jobs compete for network bandwidth with jobs that have not received high-confidence predictions of final performance (**early-stage** jobs). Consequently, the training computation of early-stage jobs is bottlenecked by network competition. Thus, the durations of the early-stage jobs are remarkably increased, which causes the time to start the next retraining to be delayed, thereby remarkably increasing the time for searching for high-quality models.

Therefore, an effective flow schedule is desired to reduce the training time of the early training stage. However, available flow schedulers cannot realize this objective. Flow-level schedulers, such as PIAS (Bai et al., 2017) and s-PERC (Jose et al., 2019), are focused on minimizing the average flow completion time. Coflow schedulers, such as Sincronia (Agarwal et al., 2018) and PRO (Guo et al., 2019), are focused on minimizing the average coflow completion time. All these flow schedulers are unaware of the progress of DDL jobs. The flows from later-stage jobs may be preferentially scheduled over flows from early-stage jobs, which can increase the training time of the early stages. The strategy of preferentially scheduling flows from early-stage jobs is straightforward but effective in reducing the early-stage time. However, in the implementation of this strategy, three challenges are encountered.

First, it is difficult to distinguish later-stage jobs and early-stage jobs as there are no standard criteria that define the early stage and the later stage. Both manual training and automatic training have their own criteria. The criteria for manual training depend on practitioners' domain knowledge and experience, which are highly subjective. The criteria for AutoMLs also differ. For example, AutoKeras (Jin et al., 2018) evaluates a job using the accuracy variance of recent epochs, while Google Vizier (Golovin et al., 2017) evaluates a job based on the probability of reaching the target accuracy. The probability is obtained from a probabilistic model (Domhan et al., 2015) that extrapolates the performance from the first part of the accuracy curve.

Second, it is difficult to minimize the average early-stage time as the duration or exact number of epochs of the early stage are unpredictable. Although the shortest job first (SJF) and shortest remaining time first (SRTF) algorithms are well-known to minimize the average JCT (Mao et al., 2019) (Dell'AmicoMatteo, 2019), they require jobs' running times or remaining times and thus cannot be used.

Third, it is difficult to realize a readily deployable and scalable design. To be deployable, the modifications to the upper DDL frameworks and the underlying network facilities must be minimal. Practitioners use various DDL frameworks, such as TensorFlow (Agarwal, 2019) and MXNet (Li et al., 2019). If the flow scheduling system requires many modifications to DDL frameworks, each DDL framework would be modified, which is difficult. If the flow scheduling system requires some support from the underlying network, any software or hardware modifications to switches will make the scheduling system non-deployable as customized switches are highly expensive. To be scalable, the overload that is caused by the scheduling system must be minimal. The per-flow rate allocation that is adopted by most prior designs requires the reallocation of the rate for every flow in the network when a flow arrives or departs. This reallocation is impractical in large-scale clusters in which thousands of flows may arrive each second. Considering scalability and deployability, practical design for scheduling flows remains elusive.

To overcome these challenges, we propose a novel **j**ob-**p**rogress-**a**ware **s**cheduler (JPAS) system by predicting the potential accuracy improvement of DL jobs and by applying the **m**aximum-**a**ccuracy-**i**mprovement-**f**irst (MAIF) scheduling policy on available priority-enabled network. JPAS orders jobs according to their potential accuracy improvement in the following scheduling period and assigns priority to a flow based on its job order. A job that has higher potential accuracy improvement has higher order; thus, its flows have higher priority. Since the accuracy improvement diminishes as the number of iterations increases, the later-stage jobs have much smaller accuracy improvement than the early-stage jobs (Jeon et al., 2018) (Zhang et al., 2017). Thus, the early-stage jobs are ordered ahead of the later-stage jobs. Then, JPAS can schedule the early-stage jobs ahead of later-stage jobs. In addition, the early-stage jobs with higher accuracy improvement are preferentially scheduled over early-stage jobs with lower accuracy improvement, which can help to reduce the average early-stage time.

JPAS predicts the potential accuracy improvement by using accuracy curves at the early stage, and the data of the accuracy curves can be obtained by reading log files of DDL jobs, which avoids modifying the upper DL frameworks. Moreover, JPAS maps flows to corresponding priority queues of the underlying network and offloads the flow scheduling and rate allocation to the underlying priority-enabled network, which requires no modification to the underlying network. Finally, as DDL jobs are long-term-running jobs, JPAS can update the job orders for a long period (e.g., tens of minutes), which has ultra-low overhead and can be scalable.

This mechanism may be not optimal, but our implementation and experiment show that JPAS performs very well. We have implemented JPAS on top of TCP, with Diff-Serv (Chan et al., 2006) for priority scheduling. Our implementation is work-conserving and efficiently handles the online arrival of flows. We evaluate the JPAS implementation on a 13-server testbed that is interconnected with a 1-Gbps commodity Ethernet switch. We use MXNet (Li et al., 2019) to conduct DDL training using a variety of models and datasets. Our experimental results demonstrate that JPAS can reduce the time to reach 90%/95% of the convergence accuracy by up to 38%.

The key contributions of the paper are as follows:

- We propose minimizing the time of the early training stages via flow scheduling, which can significantly accelerate the exploratory process of deep learning.
- We are the first to consider the training progress and propose an MAIF policy for scheduling flows for DDL jobs, which can overcome the challenge of unknown duration times or epochs of early training stages when minimizing the early stage times.
- We propose a method for predicting the accuracy improvements of DDL jobs, which is important for the MAIF policy.
- We implement MAIF and the method of accuracy prediction in a system, namely, JPAS, which is practical and readily deployable. JPAS does not require any modification to the upper ML frameworks or underlying network facilities.

The remainder of the paper is organized as follows: §2 overviews the background and the motivation of our work. §3 provides an overview of our scheduling system, namely, JPAS. Then, §4 describes the key techniques that are adopted by JPAS. §5 evaluates JPAS via testbed experiments. §6 discusses the related works, and §7 concludes the paper.

2. Background and motivation

This section provides background and motivation for JPAS. §2.1 discusses how DDL jobs are distributed in the GPU clusters. §2.2 discusses the iterative characteristic and the exploratory process of DL training. §2.3 introduces the challenges of network scheduling among DDL jobs and provides the key strategies of our solution, namely, JPAS. §2.4 discusses the potential benefits of JPAS.

2.1. Distributed deep learning in GPU clusters

Distributed training. As the sizes of the dataset and model increase, training a DNN on a single machine can take an unacceptably long time. Hence, it is common to use distributed training across machines. The most common strategy is data parallelism, where each worker loads a complete copy of the model into its own memory and each worker conducts training using a subset of the whole dataset. At the end of every iteration, all workers exchange gradients to synchronize model updates. This synchronization stage is conducted using either MPI AllReduce (IBM Spectrum MPI, 2017) or parameter servers (Li et al., 2014). The parameter server (PS) architecture is the most popular method for model synchronization. As illustrated in Fig. 1(a), the parameter servers collect model gradients from workers and update model parameters using optimization algorithms (e.g., stochastic gradient descent (SGD) (Bottou et al., 2018)). Then, the updated parameters are sent back to the workers to start the computation of the next iteration. As the model is large, each parameter server maintains a part of the model. Production DL models range from a few hundreds of megabytes to a few gigabytes, as shown in Fig. 2. The largest model can reach 7.5 GB. Larger model sizes correspond to a longer synchronization time per iteration in the distributed training. Typically, millions of such iterations and communications are conducted to realize high accuracy.

GPU clusters. Many leading companies, such as Google and Microsoft, built GPU clusters to support such a computation-intensive and communication-intensive application (Amazon2 ElasticUs.) (U-Accelerated Microsoft) (U on Google Cloud. http). A cluster is shared by multiple production groups. Thus, various DDL jobs are submitted to the cluster over time. Jobs are allocated across machines with free GPUs to maximize the utilization of expensive resources (Jeon et al., 2018) (Jeon et al., 2019). Then, a physical machine would host workers that belong to different DDL jobs, as illustrated in Fig. 1(b). When the jobs at the same machine perform model synchronizations, their data flows compete for network bandwidth at the ingress or egress port of the fabric that is connected to the machine. Due to the large model sizes, as shown in

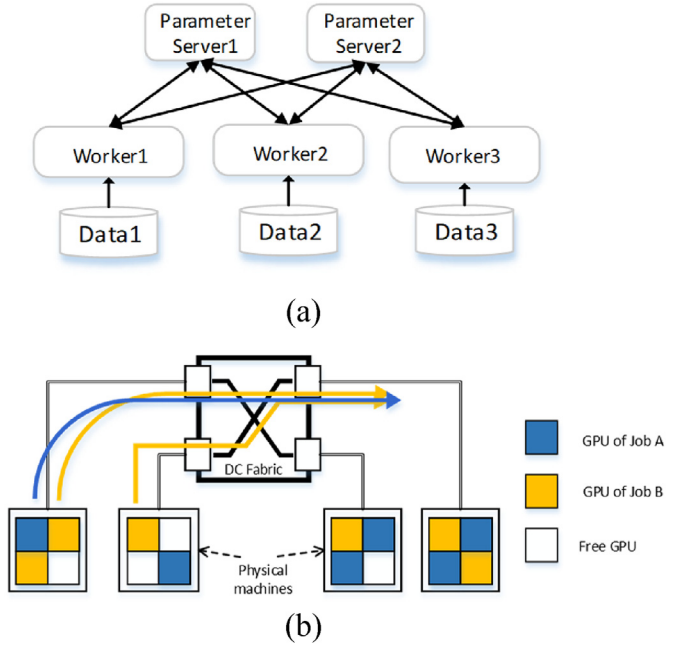


Fig. 1. DDL jobs in GPU clusters. (a) An illustration of the PS architecture for DDL. (b) A GPU cluster with four physical machines. Each machine hosts 4 GPUs. There are multiple DDL jobs that compete for resources in the GPU cluster. Workers from multiple DDL jobs are assigned to the same machine. Thus, the jobs compete for the network bandwidth of the fabric when they are conducting model synchronization.

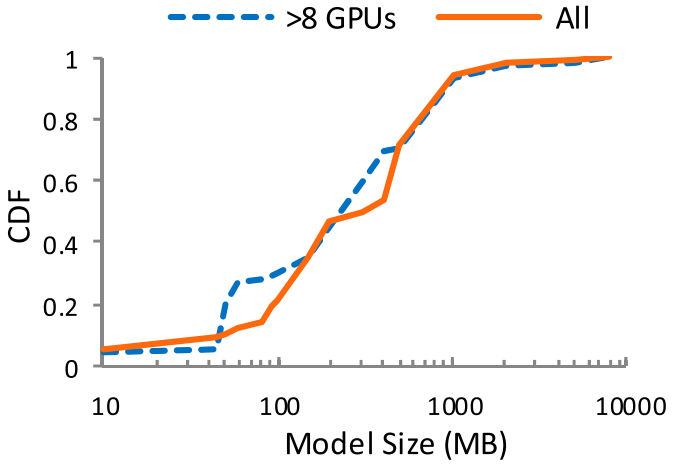


Fig. 2. CDF of the size of the trained model (Jeon et al., 2019). The dashed line is the CDF for models that use more than 8 GPUs. The solid line is the CDF for all jobs.

Fig. 2, the network competition among DDL jobs causes network congestion, even with a 100-Gbps network, and substantially degrades the job-level performance (Jeon et al., 2019). More importantly, GPUs have continued to grow rapidly in computational power and have continued to produce parameter updates faster than can be naively synchronized over the network. As a result, training is typically bottlenecked by network communication. Thus, network activity can significantly impact the training efficiency, which places higher demands on efficient network scheduling for DDL jobs. However, the characteristics of DL pose new challenges for network scheduling.

In the following, we introduce the characteristics of DL in §2.2, and we discuss the challenges of network scheduling and present our solutions in §2.3.

2.2. Characteristics of deep learning

Iteration with diminishing returns. The DL training often minimizes an objective function (the training loss) with an iterative convergent algorithm, such as stochastic gradient descent (SGD) (Bengio et al., 2017) (Bottou et al., 2018). During the training, iterations at the early stages quickly increase the model accuracy. However, iterations in the later stages continue to cost the same amounts of computational and communicational resources while making only marginal improvements in the model accuracy. The accuracy improvement diminishes with the increase of the number of iterations. Fig. 3 presents the training curves of ResNext-110 (Xie et al., 2017) on the CIFAR10 dataset (TheR-10 Dataset. htt, 2009), which correspond to the training/validation accuracy vs. the number of training epochs. For example, in the first 27% of the epochs, the training job realizes 95% of the final accuracy, and the remaining 73% of the epochs further improve the accuracy until convergence.

Feedback-driven exploration. The training of a DL model requires multiple retrainings and is often conducted via a trial-and-error approach (Fig. 4). Practitioners often repeatedly train a DL model on the same dataset to preprocess the dataset (Dunnmon et al., 2019), tune the hyper-parameters (Domhan et al., 2015), and restructure the models (Tung and Mori, 2018).

Data preprocessing plays a highly important role in many deep learning algorithms. Data preprocessing includes data cleaning, data transformation, and feature engineering. The product of data preprocessing is the final training data, which are important for realizing higher training accuracy. The identification of the training data that yield the highest quality relies on both domain knowledge and many training experiments (Dunnmon et al., 2019).

DL models and training algorithms are accompanied by a set of hyper-parameters that describe the high-level complexity of the models. The configuration of these hyper-parameters is vital for the convergence of training algorithms and the final performance of models. Examples of hyper-parameters include the number of hidden layers in a DNN and the learning rate of the training algorithms. It is desirable to try various combinations of hyper-parameter values and to train a model multiple times to identify the combination that yields the best result.

To deploy DL models and conduct inference tasks on resource-constrained systems (e.g., mobile and Internet of things devices), large models must be compressed to reduce the computational requirements and decrease the energy consumption. Many model compression techniques have been proposed. These methods modify models by quantizing the model parameters or pruning the unimportant parameters, retrain the modified model, and then modify again (Tung and Mori, 2018). This requires repeatedly training the model to realize the best compression without sacrificing the performance of the model.

In addition, the interactions among the training data, hyper-parameters and model structures make it even harder to search for the best model configuration. For example, modifying the structure of the model or training data also requires recalibrating the hyper-parameters (e.g., the learning rate). Thus, DL practitioners typically explore

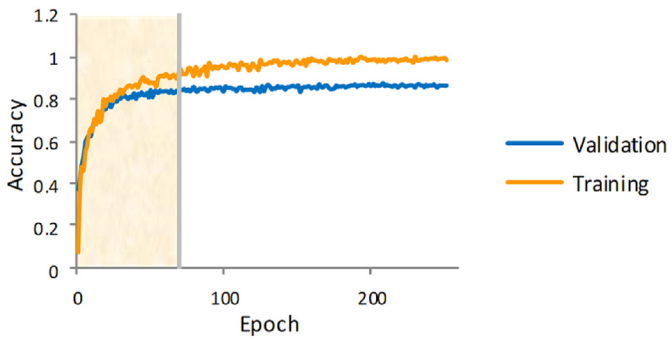


Fig. 3. Training curves of ResNext-110 on the CIFAR10 dataset.

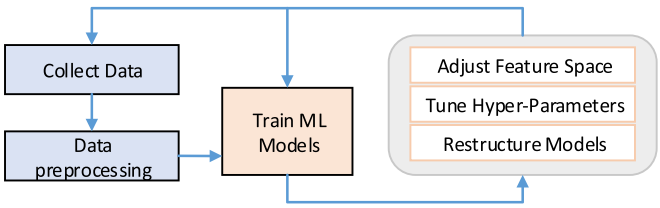


Fig. 4. Retraining of deep learning models.

hundreds of such configurations. Each configuration of DL model training could take hours or even days to converge.

Since each DL retraining is ultra-time-consuming, practitioners leverage accuracy curves of early training stages to predict the final performance of each model configuration and to exclude bad trials rather than train a model until convergence. Thus, significant amounts of time and resources can potentially be saved.

2.3. Motivation

The exact training time and number of epochs that are required for predicting the final performance with high confidence are unknown. Manually monitoring the training progress is impractical as each retraining could take a long time (e.g., several days). Thus, practitioners often submit model training jobs using many more epochs than necessary to predict the final performance (Jeon et al., 2019), thereby leading to more later-stage jobs. However, the iterations of later-stage jobs continue to cost the same amounts of computational and communicational resources while making only marginal improvements in the model quality. Worse, the data flows that are generated by later-stage jobs will compete for network bandwidth or NICs with early-stage jobs that are submitted by practitioners or AutoMLs. As a result, the training computation of early-stage jobs is bottlenecked by network competition, and thus, the durations of early-stage jobs are remarkably increased; hence, the time to start the next retraining is delayed, and thus, the time for searching for high-quality models is also remarkably increased. The demand for efficient flow scheduling in shared DL clusters to reduce the early-stage time is urgent.

Available flow schedulers cannot effectively reduce the early-stage time. Flow-level schedulers (such as PIAS (Bai et al., 2017), s-PERC (Jose et al., 2019), and RAX (Li et al., 2017)) utilize flow-level information (flow sizes) to minimize the average flow completion time. Coflow-level schedulers (such as PRO (Guo et al., 2019) and Sincronia (Agarwal et al., 2018)) adopt coflow-level information (the number and sizes of flows in a coflow) to minimize the average coflow completion time. In addition, Baraat (Dogar et al., 2014) adopts a FIFO-like approach to schedule flows at the task-level to minimize the average task completion time. Communication schedulers of DDL frameworks (e.g., R2SP (Chen et al., 2019) and ByteScheduler (Peng et al., 2019a)) decide the communication orders of workers in the same job, which can reduce the concurrency of data flows and thus alleviate network congestion. However, these flow schedulers are agnostic to the training progress of DDL jobs, and the later-stage jobs may be preferentially scheduled over early-stage jobs, leading to a long early training stage time.

Thus, the flows from early-stage jobs should be preferentially scheduled over the flows from later-stage jobs. Then, the early-stage time can be reduced to accelerate the exploratory DL training. Meanwhile, if multiple early-stage jobs are competing for network resources, the scheduling system should be able to let some early-stage jobs iterate quickly to help minimize their average early-stage time. However, in realizing these two objectives, we face the following challenges:

First, it is challenging to precisely distinguish later-stage jobs and early-stage jobs as the criteria for manual training and automatic training differ. For practitioners who are manually training their models based on their domain knowledge and experience, distinguishing between the

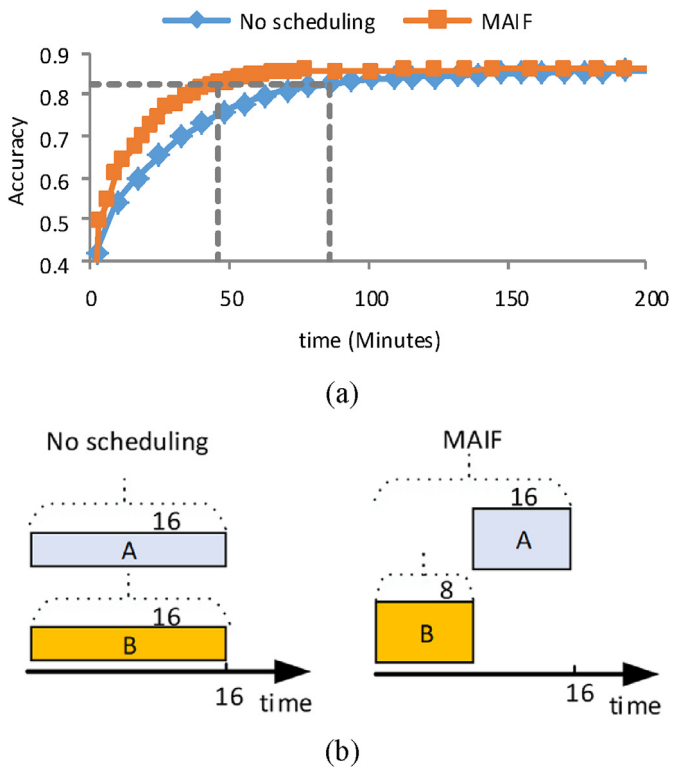


Fig. 5. Potential benefits of MAIF. (a) MAIF can accelerate the early training stage. (b) Left: If job A and job B fairly share the bandwidth, they will terminate the early stage at time 16. The average early-stage time is 16. Right: If B is scheduled first, the early-stage time of B will be reduced to 8 and the early-stage time of A will not change. The average early-stage time is reduced to 12.

early training stage and later training stage is subjective. In addition, the criteria among AutoMLs also differ. For example, AutoKeras (Jin et al., 2018) evaluates a job using the accuracy variance of recent epochs, while Google Vizier (Golovin et al., 2017) evaluates a job based on the probability of realizing target accuracy. Second, it is challenging to minimize the average early-stage time because the duration or remaining time of the early stage is unpredictable and, hence, the well-known algorithms (e.g., SJF (Mao et al., 2019) and SRTF (Dell'AmicoMatteo, 2019)) for minimizing the average JCT cannot be used.

In this paper, we propose a novel job-progress-aware flow scheduling policy, namely, MAIF, for overcoming the above challenges. MAIF is a job-level scheduling policy, where flows from jobs that realize maximum potential accuracy improvement in the next scheduling period are preferentially scheduled over flows from other jobs. This strategy performs well because since the improvement of the training accuracy diminishes as the number of iterations increases, the later-stage jobs are more likely to realize much smaller accuracy improvements than the early-stage jobs (Zhang et al., 2017) (Peng et al., 2018). Thus, flows from early-stage jobs are scheduled before flows from later-stage jobs. Flows from early-stage jobs that have higher potential accuracy improvement are also preferentially scheduled over flows from early-stage jobs that have lower potential accuracy improvement. Thus, early-stage jobs with higher potential accuracy improvement can iterate quickly to help minimize the average early-stage time.

2.4. Potential benefits

With MAIF, one of the potential benefits is that the training speed at the early stage can be significantly accelerated. Fig. 5(a) shows the accuracy change over time. The job was able to realize 95% of the converged accuracy within a much shorter time frame with MAIF than with no scheduler. This level of accuracy is frequently sufficient for exploratory training jobs. Another benefit is the minimization of the average early-stage time. For instance, in the example of Fig. 5(b), if job A and job B fairly share the network bandwidth, they will terminate the early stage at time 16. If we schedule B first (B has a larger accuracy improvement than A), the early-stage time of B will be reduced to 8 and the early-stage time of A will not change.

3. Overview of JPAS

For realizing and evaluating MAIF, we have developed a system, namely, JPAS. JPAS is readily deployable and scalable, as it requires no modifications to upper DDL frameworks and underlying networks. JPAS obtains training information by reading log files of DDL jobs and conducts priority scheduling by using the underlying priority-enabled network. JPAS orders jobs according to their potential accuracy improvement in the following scheduling period, and assigns priority to a

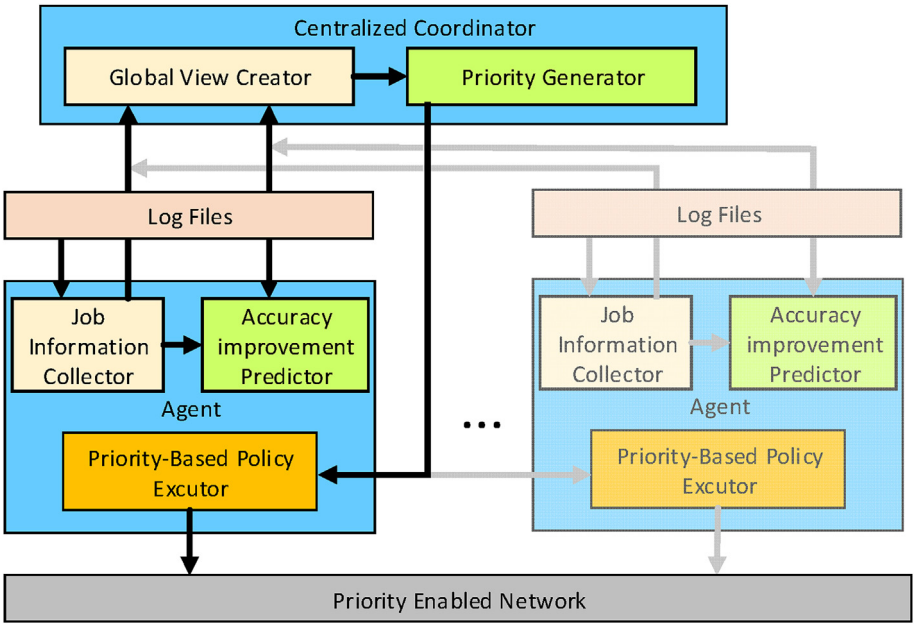


Fig. 6. System architecture overview.

flow based on its job order. A job with higher potential accuracy improvement has higher order; thus, its flows have higher priority. JPAS maps flows to corresponding priority queues of the underlying network and offloads the flow scheduling and rate allocation to the underlying priority-enabled network.

In this section, we present an overview of the architecture of JPAS in §3.1, and we discuss how to assign priorities to individual flows before JPAS offloads the flows to the underlying network in §3.2. We discuss practical issues in §3.3.

3.1. System overview

We have implemented JPAS in Python using approximately 3000 lines of code, which includes a central coordinator, and an agent at each server (see Fig. 6). JPAS agents, one on each machine, collect DL job configurations (e.g., interactions between PS and workers) and send the configurations to the coordinator every *O(1)* minutes. The JPAS coordinator aggregates all configurations of jobs and creates a global view of the network.

To reduce the amount of data exchange among the coordinator and agents as well as the overload at the coordinator, JPAS distributes the predictions regarding the potential accuracy improvements to local agents. The JPAS coordinator regards the accuracy improvement of a worker as its job's improvement. Every $T_{schedule}$ time (e.g., *O(1)* minutes), the JPAS coordinator randomly selects one worker for each DL job and sends a request for the prediction of the accuracy improvement to the worker's agent. After receiving the request, the agent will read the worker's log file and predict the accuracy improvement as described in §4. Then, the agent will send the predicted result to the coordinator as the response. JPAS uses the received responses to product job orders by conducting MAIF. With the job orders, the JPAS coordinator generates a priority scheduling rule according to the underlying priority-enabled network (we will discuss the priority generation in §3.2) and sends the rule to all the agents. Finally, the agents execute the scheduling rule (we also discuss how the rule is executed in §3.2).

3.2. JPAS + existing transport layers

We described the JPAS coordinator and agent functionalities in §3.1. Now, we introduce our JPAS implementation on top of the TCP-based network. As most of today's clusters adopt the non-blocking Clos topology, network congestion often occurs at the edge links (Wang et al., 2019). Therefore, in our implementation on top of TCP, we schedule flows at the ingress and egress ports of the host machines.

Ingress ports. As the NICs and TCP/IP stacks of servers cannot support multiple priority queues for ingress flows, JPAS uses the priority queues of the leaf-switch to schedule the ingress flows. To realize this, JPAS agents of the source hosts modify the DSCP field in the IP header (Chan et al., 2006) by using IP-tables, which is a Linux kernel tool, and the switch maps the flows to corresponding priority queues according to the DSCP value. However, today's commodity switches only support 4–8 queues per port (Mushtaq et al., 2019), and the order of a job may exceed the number of queues. The priority of a flow cannot be simply assigned as the order of the corresponding job. With a finite number M of priority queues, JPAS uses a greedy method (see Pseudocode 1) to approximate the ideal JPAS performance. The JPAS coordinator generates local orders for jobs on the same machine according to the global orders. For example, in Fig. 7, there are four nodes on the same machine from distinct jobs. JPAS maps their global orders (4,9,20,21) to local orders (1,2,3,4). With the local orders, for any flow that is forwarded to the same machine, JPAS can assign it a priority that is equal to the local order of the corresponding job (see Fig. 8). If the local order exceeds M, JPAS assigns priority M to the flow.

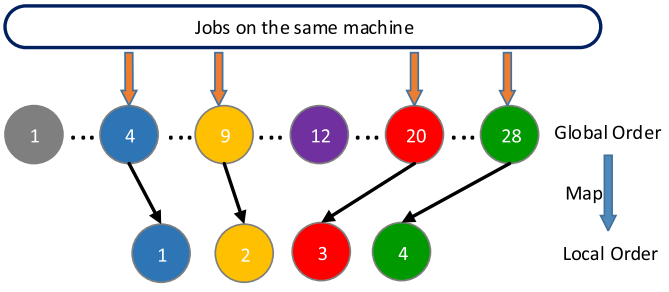


Fig. 7. Example of mapping global orders to local orders.

Pseudocode 1
Flow scheduling with finite priority queues

```
Procedure Flow Scheduling Process( )
  For every T_schedule time:
    WorkerList = ∅
    For each job J in all jobs:
      Randomly select a worker W from all workers ∈ J
      WorkerList = WorkerList ∪ W
    Send a request message with WorkerList to all agents
    Wait For Responses()
    Update job orders using MAIF algorithm
    For each agent among all agents:
      machineIP = agent.IP
      NodeSet = GenerateLocalOrder (machineIP)
      GenerateIngressPriority(NodeSet)
      GenerateEgressPriority(NodeSet)
Procedure GenerateLocalOrder (MachineIP mIP)
  JobSet = ∅
  NodeSet = ∅
  For each node N in all nodes: #Find nodes on the given machine
    If N.IP == mIP:
      NodeSet = NodeSet ∪ N
  For each node N' in NodeSet #Find jobs to which the nodes belong
    If N'.Job is not in JobSet:
      JobSet ← JobSet ∪ N.Job
  order = 0
  While (True): #generate local orders according to the global order
    If JobSet == ∅:
      Break
    Find the job J with the smallest global order number in JobSet
    order += 1#compute the local order for job J.
    For each node N'' in NodeSet: #store local order into each node
      if N''.job == J:
        N''.LocalOrder = order
    JobSet = JobSet \ {J}
  Return NodeSet
Procedure GenerateIngressPriority(set NodeSet)
  For each node N in NodeSet:
    PeerNodes = ∅
    PeerNodes = PeerNodes ∪ node for all nodes ∈ N.Job
    For each node N' in PeerNodes:
      If N.LocalOrder ≤ q: # q: number of priority queues
        PrioRule = "flow(N' → N).priority = N.LocalOrder"
      Else
        PrioRule = "flow(N' → N).Priority = q"
      Send PrioRule to the agent of N'
  Return
Procedure GenerateEgressPriority(set NodeSet)
  PrioRuleSet = ∅
  For each node N in NodeSet:
    If N.LocalOrder ≤ q': # q': number of priority queues
      PrioRule = "flow(N → ∀).priority = N.LocalOrder"
    Else
      PrioRule = "flow(N → ∀).Priority = q'"
    PrioRuleSet = PrioRuleSet ∪ {PrioRule}
  Send PrioRuleSet to agent of N
  Return
```

Egress ports. The JPAS agent uses the Linux traffic controller (TC), which is a kernel tool, to manage the egress flows. The TC of each host can create multiple priority queues for egress flows. The filter tool of the

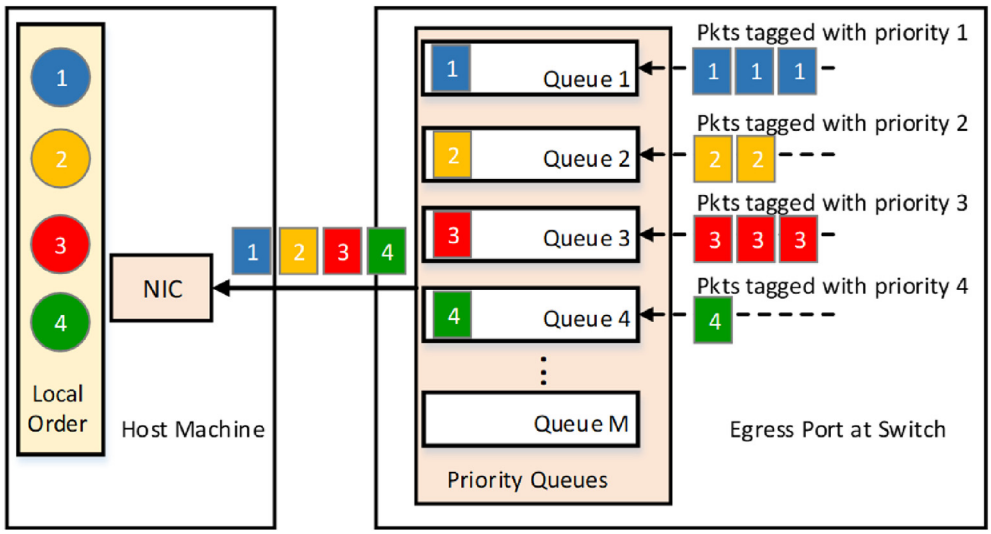


Fig. 8. Ingress flow scheduling by using priority queues at switch.

TC can map the flows to corresponding priority queues by using the TCP/IP header field. Similar to the ingress port, the JPAS agent creates K priority queues (see Fig. 9). For any flow egress from the same machine, JPAS can simply assign it a priority that is equal to the local order of the corresponding job. If the local order exceeds K, JPAS assigns priority K to the flow.

Through the above realization, at any time, a flow from job J is blocked if and only if either its ingress port or its egress port is busy serving a flow from another job J' that has a larger potential accuracy improvement than J.

Discussion. In addition to TCP fabrics, JPAS can also be implemented on other transport-layer mechanisms that support priority scheduling. For example, JPAS implementation on top of Eiffel (Saeed, 2019) is even simpler. Since Eiffel can support an infinite number of priorities, we only need to set flows' priorities equal to their job orders.

3.3. Additional practical considerations

Work conservation and work starvation. JPAS periodically updates job orders and flow priorities and offloads the priority scheduling to the underlying network. Our flow scheduling mechanisms are naturally work-conserving because underlying network priority queues (TC queues and switch queues) are work-conserving. The priority queues can support multiple algorithms for scheduling packets, such as weighted round robin (WRR) and strict priority (SP). All the algorithms are work-conserving and can provide multiple choices for JPAS for the realization of various objectives (e.g., avoid starvation). Indeed, there is a risk of starvation for JPAS in scheduling flows according to the orders of jobs. Starvation can be avoided by configuring the WRR algorithm for priority queues of the underlying network.

Co-existence with other flows. Despite the data exchanges that are generated by synchronization, all DL frameworks have other messages,

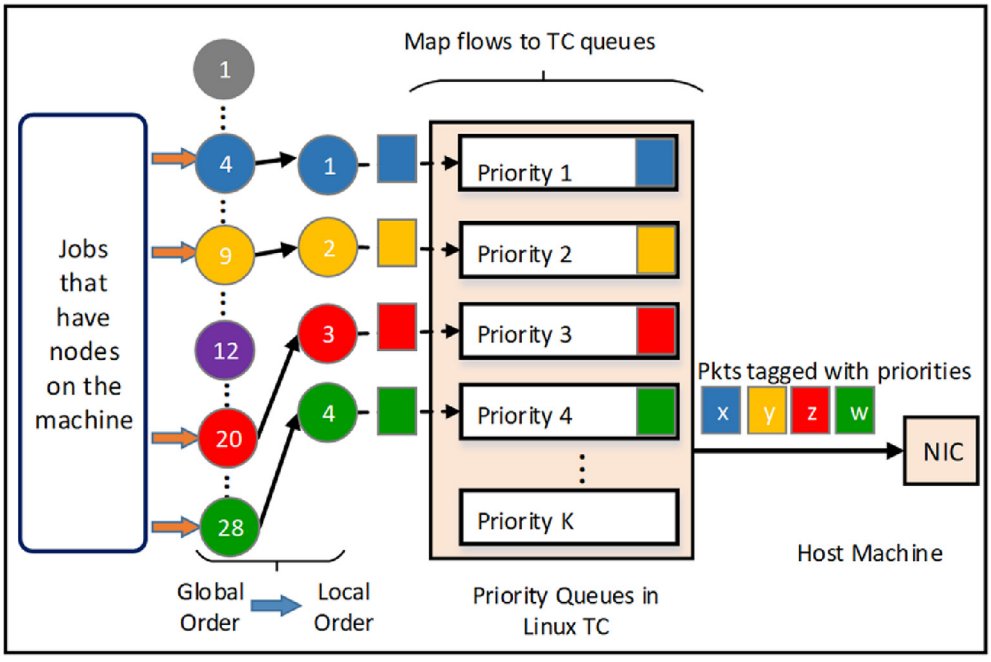


Fig. 9. Egress flow scheduling by using priority queues of Linux TC.

such as control messages and heartbeat messages. These messages do not require high throughput but are latency-sensitive. Thus, in our JPAS implementation, we reserve the highest priority, namely, priority 0, for these messages. In addition, the JPAS agents collect job configurations for a long period to reduce overload. Some jobs may start between collections, and the agents cannot register such jobs to the JPAS coordinator. The flows of such jobs may disturb the scheduling result. In the JPAS implementation, we configure the flows from these jobs with the highest priority 0 by default, which also can help JPAS detect the potential training velocity (see Engineering method in §4.2).

4. Prediction of the accuracy improvement

To predict the accuracy improvement, JPAS estimates the training velocity and use it to predict the number of iterations in the next scheduling period if the job monopolizes all bandwidth, and JPAS uses a curve model to predict the training accuracy of future iterations. Then, the difference between the accuracy at the end of the next scheduling period and the current accuracy is the accuracy improvement. The model of training-accuracy curves is presented in §4.1. We demonstrate the estimation of the training velocity in §4.2, and we discuss the calculation of the potential accuracy improvement by using the curve model and training velocity §4.3.

4.1. Accuracy curve modeling

The accuracy represents the percentage of correctly predicted data points, and the range is always from 0 to 1. In addition, the training accuracy can be output to the log files or Linux consoles, which is easy to obtain. Thus we use the accuracy improvement to measure the quality of a training job. When a DL job is running, we periodically read the training accuracy and model the accuracy curves as functions of the number of iterations. Then, we predict the accuracy improvement in the following scheduling period using the curve model.

Our basic strategy is to model the partially observed learning curve $y_{1:n}$ by a set of parametric curve models $\{f_1, f_1, \cdots, f_K\}$ (Domhan et al., 2015). The shape of these curve models coincides with our prior knowledge regarding the form of learning curves: They are typically increasing and eventually converge. Each curve model f_k is described by a set of parameters θ_k. Assuming additive Gaussian noise $\varepsilon \sim N(0, \sigma^2)$, each model f_k can model the training accuracy at iterative step m as $y_m = f_k(m|\theta_k) + \varepsilon$. The probability of observation y_m under model f_k is expressed as

$$p(y_m|\theta_k, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(y_m - f_k(m|\theta_k))^2}{2\sigma^2}} \tag{1}$$

The likelihood function can be obtained as

$$L(\theta_k, \sigma^2|y_{1:n}) = p(y_{1:n}|\theta_k, \sigma^2) = \prod_{i=1}^{n} p(y_i|\theta_k, \sigma^2) \tag{2}$$

Via the maximization of this likelihood function, we can estimate all parameters of model f_k. In total, we considered $K = 6$ curve models. The performances of these functions in modeling a typical learning curve (e.g., training accuracy of ResNext-110) are presented in Fig. 10, and their parametric formulas are listed in Table 1. Each of these models captures certain aspects of the learning curves, but no single model can describe all learning curves by itself, which motivates us to combine the models in a probabilistic framework (Domhan et al., 2015). The combined model is obtained via the following weighted linear combination:

$$f_{comb}\left(m|\boldsymbol{\xi}\right) = \sum_{k=1}^{K} \omega_k f_k(m|\theta_k) \tag{3}$$

where $\boldsymbol{\xi}$ is a new combined parameter vector, which is expressed as

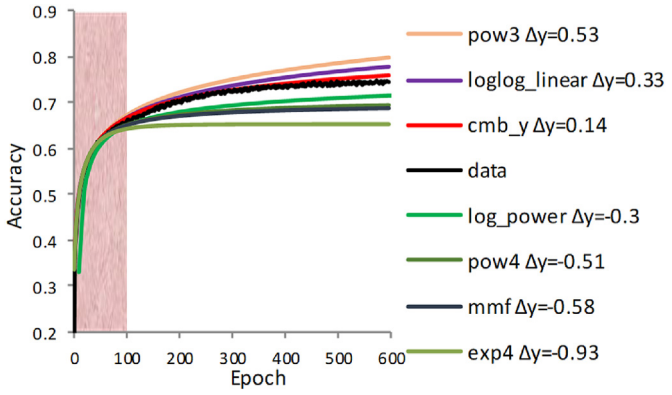


Fig. 10. Example of modeling training curves. Seven parametric models are used to predict the future accuracy curve from its first 100 epochs. The legend is sorted according to the prediction error at epoch 300. The cmb_y is the linearly combined model of the other 6 models.

Table 1
Formulas of the curve models.

Reference name	Formula
pow4	$c - (ax + b)^{-\alpha}$
pxp4	$c - e^{-ax^{\alpha}+b}$
exp3	$c - ax^{-\alpha}$
log log linear	$\log(a\log(x) + b)$
log power	$\frac{a}{1 + \left(\frac{x}{e^b}\right)^c}$
MMF	$\alpha - \frac{\alpha - \beta}{1 + (kx)^{\delta}}$

$$\boldsymbol{\xi} = \left(\omega_1, \cdots, \omega_k, \theta_1, \cdots, \theta_K, \sigma^2\right) \tag{4}$$

The parameter vector $\boldsymbol{\xi}$ consists of the individual parameters θ_k and the corresponding weight ω_k for each model f_k, along with noise variance σ^2. Then, the observed y_m from the curve model with parameter vector $\boldsymbol{\xi}$ can be expressed as follows

$$y_{m|\xi} = f_{comb}(m|\boldsymbol{\xi}) \tag{5}$$

The future point $y_m (m > n)$ can be predicted as the expectation $\hat{y}_m$ of observations $y_{m|\xi}$ under the curve model with various parameter vector $\boldsymbol{\xi}$, which can be formulated as $\hat{y}_m = \mathbb{E}[y_{m|\xi}]$. By combining this with Eq. (5), we obtain

$$\hat{y}_m = \mathbb{E}[f_{comb}(m|\boldsymbol{\xi})] = \int f_{comb}(m|\boldsymbol{\xi}) p(\boldsymbol{\xi}|y_{1:n}) d\boldsymbol{\xi} \tag{6}$$

where $p(\boldsymbol{\xi}|y_{1:n})$ is the posterior probability of the curve model with parameter vector $\boldsymbol{\xi}$ under the observation of historical training accuracies $y_{1:n}$. $p(\boldsymbol{\xi}|y_{1:n})$ is expressed as

$$p(\boldsymbol{\xi}|y_{1:n}) \propto p(y_{1:n}|\boldsymbol{\xi}) p(\boldsymbol{\xi}) \tag{7}$$

where $p(y_{1:n}|\boldsymbol{\xi})$ denotes the probability of the observed training accuracies of the first n iterations to be $y_{1:n}$ when the curve model's parameter vector is a specified vector $\boldsymbol{\xi}$. $p(y_{1:n}|\boldsymbol{\xi})$can be formulated as the following joint probability

$$p\left(y_{1:n}|\xi\right) = \prod_{i=1}^{n} p(y_i|\xi\right) \tag{8}$$

where $p(y_i|\boldsymbol{\xi})$ is the probability of a single observation at iteration i under the curve model with parameter vector $\boldsymbol{\xi}$. Assuming additive Gaussian noise $\varepsilon \sim N(0, \sigma^2)$, the curve model with parameter vector $\boldsymbol{\xi}$ can model the

training accuracy at iterative step i as $y_i = f_{comb}(i|\boldsymbol{\xi}) + \varepsilon$. Hence $p(y_i|\boldsymbol{\xi})$ is expressed as follows:

$$p(y_i|\boldsymbol{\xi}) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(y_i - f_{comb}(m|\boldsymbol{\xi}))^2}{2\sigma^2}} \tag{9}$$

In addition, $p(\boldsymbol{\xi})$ in Eq. (7) is the prior distribution over parameter vector $\boldsymbol{\xi}$, which is proportional to the joint probability of all types of parameters and thus can be formulated as follows

$$p(\boldsymbol{\xi}) \propto \left(\prod_{k=1}^{K} p(\omega_k) p(\theta_k) \right) p(\sigma^2) \mathrm{I}(f_{comb}(1|\xi) < f_{comb}(m|\xi)) \tag{10}$$

where $p(\theta_k)$ and $p(\sigma^2)$ are prior distributions over parameters θ_k and σ^2, respectively. Both are uninformative; hence, it would be simplest to choose an uninformative prior, such that $p(\theta_k) \propto 1$ and $p(\sigma^2) \propto 1$. To ensure that the modeled curves are non-decreasing functions, the following two restrictions are imposed. First, the indicator function $\mathrm{I}(f_{comb}(1|\xi) < f_{comb}(m|\xi))$ in Eq. (10) is equal to 1 if $f_{comb}(1|\boldsymbol{\xi}) < f_{comb}(m|\boldsymbol{\xi})$, otherwise, it is equal to 0, which ensures that the curve model that decreases from the initial value to the point of prediction m is assigned no probability mass. Second, we restrict the weights in Eq. (3) to non-negative values by setting the uninformative prior to the weights $p(\omega_k)$ as follows

$$p(\omega_k) \propto \begin{cases} 1 & if\ \omega_k > 0 \\ 0 & otherwise \end{cases}, \forall k \in [1, K] \tag{11}$$

The exact value of $p(\boldsymbol{\xi}|y_{1:n})$ in equation (7) is not known; hence, the integral in equation (6) cannot be calculated. Thus, we conduct Markov Chain Monte Carlo (MCMC) sampling over the space of combined parameter vector $\boldsymbol{\xi}$ by drawing S samples $\xi_1, \xi_2, \cdots, \xi_S$ from the posterior distribution $P(\boldsymbol{\xi}|y_{1:n})$ in equation (7). According to the law of large numbers, we can approximate $\hat{y}_m$ as

$$\hat{y}_m = \frac{1}{S} \sum_{s=1}^{S} f_{comb}(m|\xi_s) \tag{12}$$

To accelerate the sampling procedure, we set all initial model parameters θ_k to their (per-model) maximum likelihood estimates. The model weights are initialized to $\omega_k = \frac{1}{K}$. The noise variance σ^2 is also initialized to its maximum-likelihood estimate $\sigma^2 = \frac{1}{n} \sum_{i=1}^{n} (y_i - f_{comb}(m|\boldsymbol{\xi}))^2$. According to Fig. 10, the combined model can realize the highest prediction accuracy.

4.2. Estimating training velocity

Two methods are available for obtaining the training velocity of a job that is monopolizing all bandwidth: a mathematical method for modeling the training velocity as a function of the bandwidth and an engineering method for running the job and measuring the velocity.

Mathematical method. For each worker, we can define the training velocity as the number of training steps per second, which is expressed as

$$V_{train} = \frac{1}{Tstep} \tag{13}$$

where $Tstep$ is the duration of one iterative step. Each training step consists of a pull stage, a gradient computing stage, a push stage and a parameter synchronization stage as illustrated in Fig. 11. In the design of DL frameworks (such as TensorFlow (Agarwal, 2019) and MXNet (Li et al., 2019)), these stages are partially overlapping to reduce the synchronization latency. In the worst case, the overlap of these stages is minimal and can be ignored. Then, we can obtain maximum step time given as

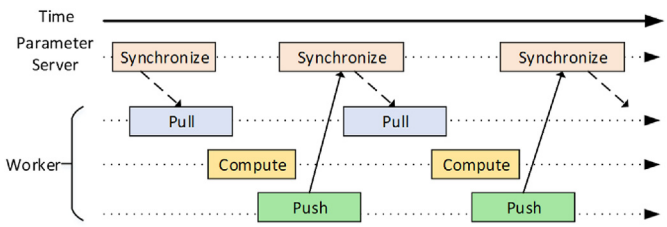


Fig. 11. Work flow of distributed machine learning using PS. Each training step consists of a pull phase, a gradient computing phase, a push phase and a parameter synchronization phase. These phases are partially overlapping.

$$Tstep_{max} = T_{pull} + T_{compute} + T_{push} + T_{syn} \tag{14}$$

where T_{pull}, $T_{compute}$, T_{push} and T_{syn} are the durations of the pull stage, gradient computing stage, push stage and synchronization stage, respectively. In the best case, the communication stages (the push stage and the pull stage) and the computing stages (the gradient computing stage and the parameter synchronization stages) are completely overlapped, and we can obtain minimal step time as

$$Tstep_{min} = Max(T_{pull}, T_{push}, T_{compute} + T_{syn}) \tag{15}$$

Suppose that a DL job has w workers and p parameter servers. The bandwidth capacity of each server is B_{ps}, the bandwidth capacity of each worker is B_w and the model size is MS(the parameter size in bytes). If the parameters are evenly distributed on parameter servers, the size of the gradients or updated parameters that are sent between a worker and a parameter server is MS/p. As pushing gradients and pulling updated parameters are symmetric processes, the duration of the pull stage and the push stage are the same:

$$T_{pull} = T_{push} = \frac{MS/p}{Min(B_{ps}/w, B_w/p)} + T_{latency} \tag{16}$$

where $\frac{MS/p}{Min(B_{ps}/w, B_w/p)}$ is the transmission delay, and the end-to-end latency $T_{latency}$ is the sum of the propagation delay and the queuing delay in the network, which can be directly measured. $T_{compute}$ and T_{syn} are closely related to the computational power of the physical servers and the software realization of each DL framework. However, both $T_{compute}$ and T_{syn} can be directly measured online by adding code into the DL frameworks. Thus, we can reformulate $Tstep_{min}$ and $Tstep_{max}$ as functions of the bandwidth capacity:

$$Tstep_{max}(B_{ps}, B_w) = 2\left(\frac{MS/p}{Min(B_{ps}/w, B_w/p)} + T_{latency}\right) + T_{compute} + T_{syn} \tag{17}$$

$$Tstep_{min}(B_{ps}, B_w) = Max\left(\frac{MS/p}{Min(B_{ps}/w, B_w/p)} + T_{latency}, T_{compute} + T_{syn}\right) \tag{18}$$

The duration of one training step can be modeled as

$$Tstep(B_{ps}, B_w) = (1 - \gamma) \cdot Tstep_{max}(B_{ps}, B_w) + \gamma \cdot Tstep_{min}(B_{ps}, B_w) \tag{19}$$

where γ is a tunable parameter.

Fig. 12 shows the measured iterative velocity and the modeled speed curves when we train an image classification model, namely, VGG16, by using 6 workers and a parameter server on MXNet (Li et al., 2019). The measured curve is between the theoretical maximum and minimum curves. As DL models are very large, the PS divides the models into multiple parts, which are synchronized separately. Thus, the push stage and pull stage are typically overlap. The overlapped time between push and pull increases with the model size. As a result, the measured velocity curve is closer to $Tstep_{min}$. In addition, the bigger the proportion of the communication time in one training step is, the closer the velocity curve

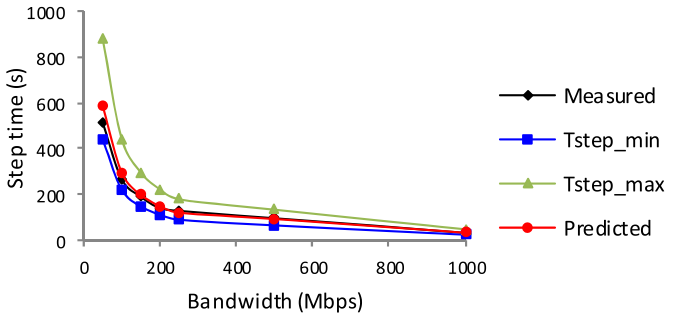


Fig. 12. Training velocities of VGG16 under various bandwidths. The model can precisely estimate the velocity.

is to $Tstep_{min}$. Therefore, we are going to set γ as follows:

$$\gamma = \frac{T_{pull}}{T_{pull} + T_{compute}} \cdot \frac{Tstep_{max}}{Tstep_{max} + Tstep_{min}} \tag{20}$$

The red line in Fig. 12 represents the estimated result. The averaged error is 6.23%.

Engineering method. We assign a newly arriving job the highest priority. The DL frameworks can output their training accuracies and iteration times at each iterative step. We choose the smallest iteration times as the estimate of one step time under the condition of monopolizing all bandwidth. Then, we can obtain training velocity via using equation (13). There may be multiple new jobs that have been assigned the highest priority. However, the flows of DL jobs exhibit an on-off pattern, and it is possible for one of the jobs to monopolize all bandwidth at an iterative step, as shown in Fig. 13.

Discussion on the two methods. The mathematical method can always calculate the training velocity. However, it requires modifications to DL frameworks to obtain various information (e.g., *MS*, $T_{compute}$, and T_{syn}). The engineering method may not always yield the true training velocity. However, it requires no modifications to DL frameworks. Thus, our evaluation in Section 6 adopts the engineering method, and our experiments show that it performs well.

4.3. Calculation on accuracy improvement

The accuracy improvement A_{imp} in the next scheduling period $T_{schedule}$ can be expressed as

$$A_{imp} = \hat{y}_{t+T_{schedule} \cdot V_{train}} - y_t \tag{21}$$

where $T_{schedule} \cdot V_{train}$ is the number of iterative steps in the next scheduling period. Combining equations (21) and (12) yields

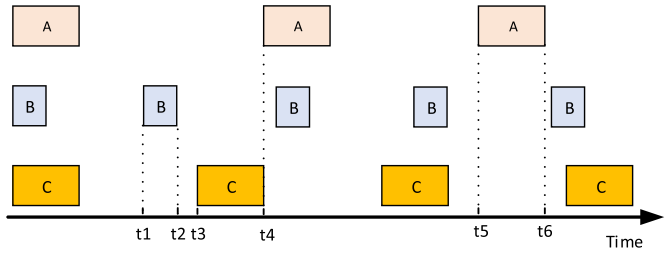


Fig. 13. Example of a job monopolizing the bandwidth at an iterative step. The parameter servers of three jobs (A, B, and C) are in the same host machine. Workers of the jobs exchange data with their parameter servers at time 0. In time interval t1-t2, job B monopolizes the bandwidth. In time interval t3-t4, job C monopolizes the bandwidth; and in time interval t5-t6, job A monopolizes the bandwidth.

$$A_{imp} = A_{imp} = \frac{1}{S} \sum_{s=1}^{S} f_{comb}(t + T_{schedule} \cdot V_{train} | \xi_s) - y_t \tag{22}$$

Then, we can obtain the job orders by sorting the jobs in decreasing order of accuracy improvement.

5. Evaluation

5.1. Experimental settings

Testbed. We built a testbed that consists of 10 CPU servers and 3 GPU servers that are connected to a 48-port Edge-core AS4610-54 T 1 Gb E switch, as shown in Fig. 14. Each port of the switch has 8 priority queues. Each CPU server has two 8-core Intel E5-2609 CPUs, 32 GB memory, and two 300 GB HDDs. Each GPU server has one 8-core Intel E5-2650 CPU, two NVIDIA 1080Ti GPUs, 64 GB memory, one 500 GB SSD and one 4 TB HDD. We use Kubernetes 1.7 (Kubernetes. https://kuber, 2017) to schedule the arrived jobs. The servers use Ubuntu 16.04 as the operating system.

Workload. The job arrival intervals are similar to those of Microsoft's cluster (Jeon et al., 2018) (Gu et al., 2019) as shown in Fig. 15. As our cluster is much smaller than Microsoft's cluster, the unit of the arrival interval time is extended from seconds to minutes. Upon an arrival event, we randomly choose a job among the examples in Table 2 and run it using asynchronous training or synchronous training randomly. We set the ratio of the number of parameter servers to the number of workers to 1:1. Limited by the scale of our testbed, the number of the workers/servers of each DDL job is randomly chosen between$[2, 4]$. For jobs that involve training large models (e.g., VGG19), we downscale their dataset sizes to ensure that the experiment can be completed in an acceptable time; otherwise, each experiment would take several weeks. After downscaling, one experiment takes approximately 32 h, and we repeat each experiment 3 times to obtain the average results. In addition, MXNet (Li et al., 2019) is used to conduct distributed training.

Baselines. We compare JPAS with three mechanisms: (i) coflow scheduler Sincronia (Agarwal et al., 2018), which also utilizes underlying network priorities to schedule coflows; (ii) task-aware scheduler Baraat (Dogar et al., 2014), which regards the flows within a task as a single flow and schedules flows from tasks via a FIFO approach; (iii) pure TCP without any flow schedulers.

Metrics. For exploratory training jobs, 90% of the converged accuracy is frequently sufficient for terminating the early stage (Zhang et al., 2017). We use the average times to reach 90%/95%/99% of the converged accuracy as indicators of the system performance. In addition, we evaluate the overall accuracy improvement as an indicator of the network bandwidth efficiency.

5.2. Performance

Comparison with baselines. Fig. 16 shows that JPAS can reduce the average training time to reach 90% or 95% of the converged accuracy by up to 38%, compared with pure TCP; while the reduction in training time to reach 99% of the converged accuracy is approximately 13%. With Sincronia and Baraat, the time to reach 90% or 95% of converged accuracy is increased by up to 13% and 24%, respectively. As Sincronia and Baraat are both agnostic to the training progress of DDL jobs, the later-stage jobs may be preferentially scheduled before the early-stage jobs, which can accelerate the iterative velocity of later-stage jobs and reduce the total job completion time (JCT). In contrast, JPAS slightly increases JCT by 4%.

Fig. 17 presents the training times to reach 95% of the converged accuracy (normalized to TCP) for various model sizes under several schedulers. When JPAS is used, the training time to reach 95% of the converged accuracy is lower than those of the other three methods. For models of less than 32 MB, the proportion of the communication time relative to the iteration time is low, and the scheduler performance is not

Fig. 14. Testbed topology.

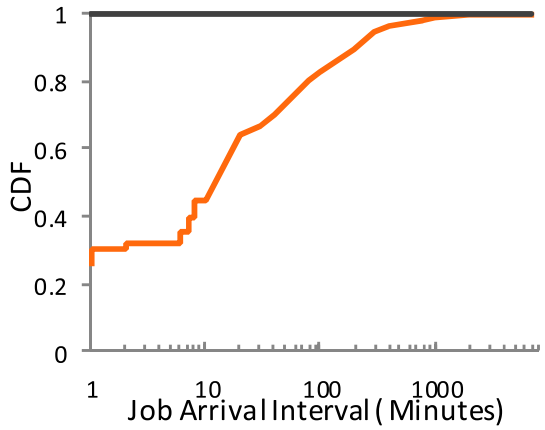


Fig. 15. CDF of the DDL job arrival intervals.

Table 2
Deep learning jobs for tests and experiments.

Models	Size(MB)	Dataset	Application
ResNext-110	6.8	CIFAR10	Image classification
RNN-LSTM-Dropout	18	PTB	Language modeling
DenseNet121	32	CIFAR10	Image classification
Inception_V3	95	ImageNet	Image classification
ResNet50	100	ImageNet	Image classification
DeepSpeech2	152	Libri Speech	Speech recognition
AlexNet	244	ImageNet	Image classification
VGG19	578	ImageNet	Image classification

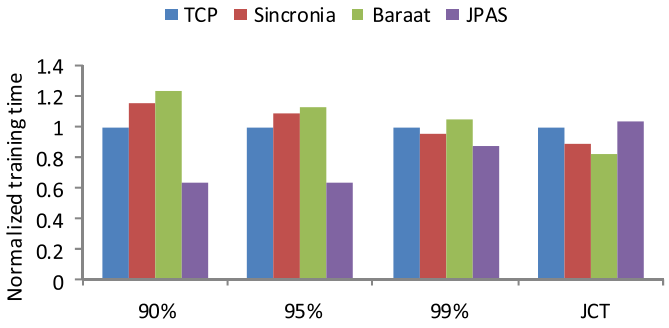


Fig. 16. Comparisons of the training times to reach 90%/95%/99% of the converged accuracy and comparison of the average job completion time (JCTs). JPAS can reduce the times to reach 90%/95% of the converged accuracy by up to 38%.

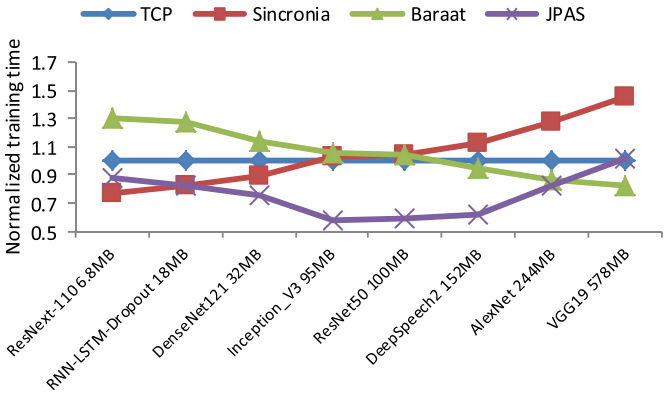


Fig. 17. Comparisons of the training times of models of various sizes to reach 95% of the converged accuracy. When JPAS is used, the training time to reach 95% of the converged accuracy is lower than those of the other three methods.

outstanding. As the model size increases (e.g., Inceptioin_V3 and DeepSpeech2), the proportion of the communication time relative to the iteration time increases and the training time to reach 95% of the converged time can be reduced by up to 43%. If the models (e.g., VGG19) are too large, their iterative velocities are much smaller than those of small models. Thus, the accuracy improvements of such large models in a scheduling period are also less than those of small early-stage models. As a result, JPAS will preferentially schedule small early-stage models, which cannot accelerate the early-stage training of such large models when they coexist with small early-stage models.

When Baraat is adopted, the jobs are scheduled via a FIFO approach, and small models are easily blocked by large models. As a result, Baraat can reduce the time to reach 95% of the converged accuracy for bigger models (larger than 152 MB), but the time to reach 95% of the converged accuracy for smaller models (less than 95 MB) is increased. When Sincronia is adopted, flows from the small models have higher priority. Thus, Sincronia can reduce the time to reach 95% of the converged accuracy for smaller models (less than 95 MB), but it increases the time to reach 95% for larger models.

JPAS can maintain higher total accuracy improvement compared with the other three methods, as shown in Fig. 18. The reason is that JPAS preferentially schedules jobs with higher accuracy improvement. Baraat and Sincronia can maintain a larger accuracy improvement than TCP when some early-stage jobs happen to be priority scheduled.

JPAS can also maintain a larger total loss reduction compared with the other three methods, as shown in Fig. 19. The reason is that JPAS preferentially schedules jobs with higher accuracy improvement. These jobs typically have larger loss reductions. For Baraat and Sincronia, when early-stage jobs happen to be priority scheduled, their total loss reductions are larger than the loss reduction of TCP.

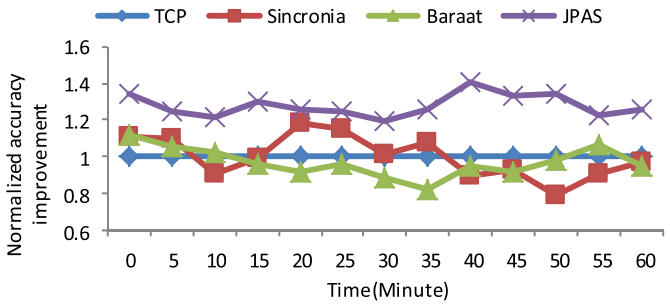


Fig. 18. Comparison of the normalized total accuracy improvement (normalized to TCP). JPAS can maintain a larger total accuracy improvement than the other three methods.

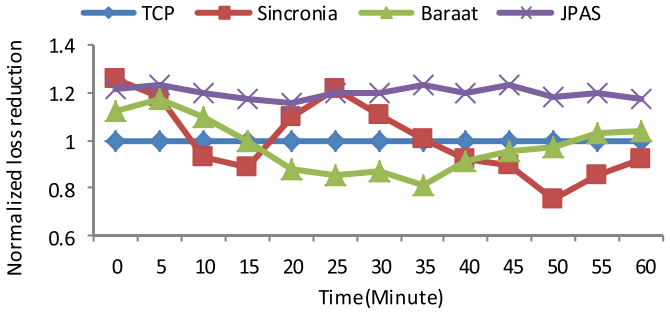
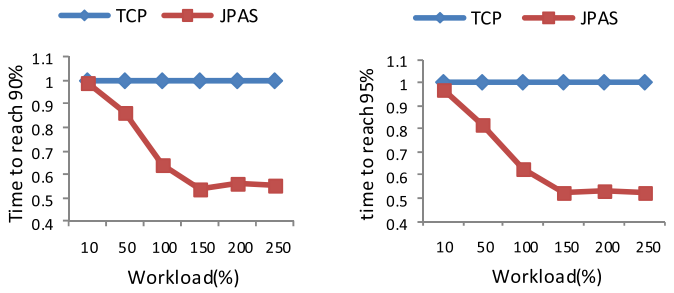


Fig. 19. Comparison of normalized total loss reduction (normalized to TCP). JPAS maintains a larger total loss reduction than the other three methods.

Bandwidth allocation among DDL jobs. Fig. 20 shows the bandwidth allocation among jobs that differ in terms of accuracy improvement when JPAS is adopted. The top 30% of jobs in terms of accuracy improvement occupy more than 60% of the bandwidth. The second 30% of jobs occupy approximately 30% of the bandwidth. The bottom 40% of jobs occupy less bandwidth than the other jobs. Fig. 20 illustrates that JPAS can effectively reduce the communication times of jobs that have large accuracy improvements.

5.3. Sensitivity analysis

Varying workloads. The realized performance of JPAS strongly depends on the cluster workload. As the workload increases, the network becomes more congested, and the efficient utilization of the bandwidth becomes more important. In this experiment, we vary the arrival interval of new jobs, which, in turn, varies the number of concurrent jobs, and we observe how JPAS and pure TCP without any scheduler handle network contention under various workloads. The job arrival interval follows the distribution in Fig. 15. We compress or expand the scale of the horizontal ordinate in Fig. 15 to generate various workloads.



(a) Time to reach 90% (b) Time to reach 95%

Fig. 21. Times to reach 90% and 95% of the converged accuracy under various workloads.

Fig. 21 plots the times to reach 90% and 95% of the converged accuracy under various workloads when JPAS is or is not used. As the workload increases, the reduced time to reach 90% or 95% of the converged accuracy increases. This is because the proportion of the communication time relative to the iteration time increases as the workload becomes heavier. The scheduling of flows can significantly impact the communication time of DDL jobs and thus impact the iterative velocity. Fig. 21 also shows that when the workload exceeds 200%, the performance of JPAS is not changed. This is because the resource of the cluster is limited and the number of running jobs is also limited. When the workload exceeds 200%, the number of running jobs is not changed and the newly arrived jobs are more likely to be queued to wait for resources to become available.

Robustness of prediction. JPAS relies on a prediction of the future accuracy improvement of a job under the specified total bandwidth (see Section 3). We examine the extent to which JPAS is affected by the prediction errors of the accuracy improvement. We conduct experiments under various error levels: Suppose the prediction *Pre* of JPAS is precise and an error e is artificially added. We use $Pre \cdot (1+e)$ or $Pre \cdot (1-e)$ as the initial input to JPAS to produce job orders. We conduct experiments under various errors, where e is set to $10\%, 20\%, \cdots, 50\%$.

Fig. 22 shows the times to reach 95% of the converged accuracy under various errors and illustrates that JPAS is robust to prediction errors. When the error is as large as 30%, JPAS can still reduce the time to reach 95% of the converged accuracy by up to 28%. When the error exceeds 40%, the performance is decreased. However, JPAS still can reduce the time to reach 95% by approximately 15%. This is because JPAS is not a precise rate allocation scheduler and it performs priority scheduling by ordering jobs. The jobs are ordered according to their prediction of accuracy improvements. A small scale error cannot affect the job orders.

Scalability of JPAS. The scheduling of JPAS relies on the prediction of the future accuracy improvement and the periodic updating of job orders. If the scheduling period is second-level or sub-second-level, JPAS

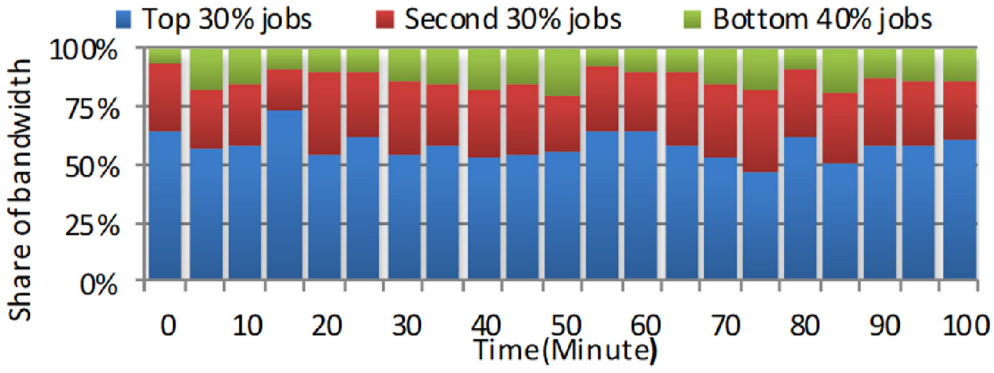


Fig. 20. Bandwidth allocation across jobs when using JPAS.

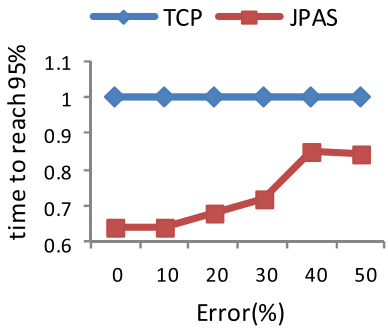


Fig. 22. Times to reach 95% of the converged accuracy under various prediction errors.

could cause a high overload for each machine and impact the computation of DL jobs. However, the scheduling period of JPAS is minute-level, which does not cause an overload for any machine.

Fig. 23 plots the times to reach 95% of the converged accuracy under various scheduling periods. When the scheduling period exceeds 50 min, the performance of JPAS is almost the same as that of TCP. However, Fig. 23 also illustrates that JPAS can update the scheduling policies every 10 min without reduction in performance. Thus, JPAS does not cause overloads and has satisfactory scalability.

6. Related work

Distributed learning frameworks. The most common option for DDL training is data parallelism, where the training dataset is divided into equal-sized parts to feed workers. The workers synchronize the model updates from other workers once per iteration by using MPI-Allreduce (IBM Spectrum MPI, 2017) or parameter servers (Li et al., 2014). Most distributed ML/DL frameworks (e.g., MXNet (Li et al., 2019), TensorFlow (Agarwal, 2019) and Angel (Jiang et al., 2017)) employ the parameter server (PS) architecture (Li et al., 2014). The optimization of DDL frameworks is aimed at allocating resources inside a job and thus improving the training speed of a single job. In contrast, our work is aimed at scheduling resources among multiple jobs and improving the overall performance of jobs in the cluster. Thus, the research on DDL frameworks and the research on resource scheduling are complementary.

Resource management in DDL clusters. Many resource management systems for DDL clusters have been proposed for sharing a single cluster with multiple DDL jobs using various frameworks. Dorm (Sun et al., 2017) advocates partitioning the cluster and runs one application per partition. Dorm can dynamically resize each partition at application runtime to realize resource efficiency and fairness by solving a mixed integer linear program (MILP). If a new job arrives, Dorm can withdraw some resources from the running jobs and allocate the resources to new jobs. Schedulers, such as Harmony (Bao et al., 2019) and DL2 (Peng et al., 2019b), use reinforcement learning to place DDL jobs in GPU clusters, which can improve the resource utilization and reduce the JCT. Optimus

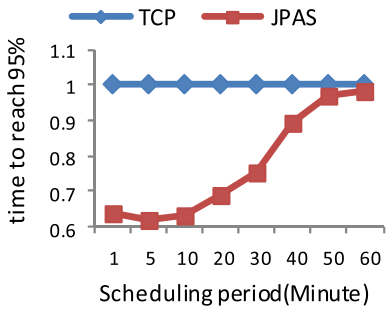


Fig. 23. Times to reach 95% of the converged accuracy under various scheduling periods.

(Peng et al., 2018) and SLAQ (Zhang et al., 2017) build resource-performance models on the fly and dynamically adjust the resource allocation to minimize the JCT or reduce the early-stage time. In addition, Tiresias (Gu et al., 2019) and Gandiva (Xiao et al., 2018) can reduce the early-stage time for DDL jobs by realizing GPU time-sharing and simultaneously scheduling multiple jobs from the same retraining. All the above resource managers are focused on computing resource (e.g., GPUs and memory) allocation and ignore the flow scheduling. Thus, these resource managers are complementary to JPAS.

Flow scheduling in data centers. s-PERC (Jose et al., 2019) and RAX (Li et al., 2017) realizing flow scheduling by using proactive explicit rate control. Eiffel (Saeed, 2019) is a novel programmable packet scheduling system, which can facilitate the realization of decentralized in-network prioritization and the implementation of various packet scheduling policies at switches and hosts. PIAS (Bai et al., 2017) leverages in-network priority queues to perform least-attained-service-first flow scheduling. PAM (Luo et al., 2020) adopts a priority-based multicast protocol to realize small completion times. These flow-level scheduling methods are aimed at minimizing average flow completion time (FCT), and most advanced coflow schedulers are proposed for minimizing coflow completion time (CCT) (Wang et al., 2019). For example, Sincronia (Agarwal et al., 2018) uses priority queues of the underlying network to schedule coflows to minimize the average CCT. PRO (Guo et al., 2019) jointly optimizes tasks placement and the routing of coflows to reduce CCT. None of these approaches consider the key characteristics of DDL training, which are discussed in Section 1, and fail to realize the performance objectives for accelerating deep learning. In contrast, JPAS fully utilizes the characteristics of DDL jobs and schedules flows for DDL jobs to minimize the training time of early stages, which can significantly accelerate the exploratory process of deep learning.

Recently, job-level schedulers, such as Baraat (Dogar et al., 2014) and multi-stage coflow schedulers (Tian et al., 2018), were proposed. For example, Baraat takes flows from the same job as a single flow and schedules flows from jobs via a FIFO approach with limited multiplexing. This approach cannot be used to schedule flows for DDL since early-stage jobs can be blocked by later-stage jobs, which can significantly increase the time for searching for a satisfactory model configuration. The multi-stage coflow schedulers (Tian et al., 2018) are aimed at minimizing the JCT. They typically assume that the duration of a job or a stage is known in advance. However, the durations or numbers of epochs of early stages of training jobs cannot be obtained; hence, the use of these schedulers to accelerate deep learning is unpractical. Instead, JPAS uses a maximum accuracy improvement first policy to schedule flows from DDL jobs, and the accuracy improvement can easily be predicted.

Communication schedulers for DL frameworks, such as R2SP (Chen et al., 2019) and ByteScheduler (Peng et al., 2019a), enable workers to synchronize parameters with parameter servers in a well-designed order, which can reduce the concurrency of data flows and thus can alleviate network congestion. However, they cannot schedule flows between DDL jobs and thus cannot improve the overall performance of jobs in the cluster. JPAS is designed for scheduling flows from multiple DDL jobs and for reducing their training times in the early stages.

7. Conclusion

In this paper, we show that DDL jobs in GPU clusters compete for network bandwidth and that the network becomes one of the major bottlenecks for DDL training. Thus it is urgent to implement an efficient flow scheduling method to improve the training efficiency. We also show that deep learning requires the minimization of the durations of the early training stages to accelerate the exploratory process. However, the most formidable challenge is that the exact durations and numbers of epochs of early stages of training jobs are unknown. Thus, available algorithms, such as SJF and SRTF, cannot be used to reduce early-stage training time. To overcome the challenge, our scheduling system JPAS proposes a new method, MAIF, for scheduling flows by considering the jobs' potential

accuracy improvements in the next scheduling period instead of using the unknown durations or remaining times of the early stages. The potential accuracy improvement can be easily predicted by using the proposed accuracy curve model. To be readily deployable, JPAS adopts priority queues of the underlying networks and assigns flows of jobs to different queues according to their potential accuracy improvements. The flows of jobs with higher accuracy improvements will be assigned to higher priority queues. Through testbed experiments, JPAS can effectively reduce early-stage times for DDL jobs and thus can accelerate the exploratory training process. Currently, JPAS focuses on flow scheduling without the use of network congestion control, while congestion control schemes (Li et al., 2017) (Pan et al., 2019a) (Pan et al., 2019b) can improve the network latency and throughput and thus can further improve the performance of JPAS. Moreover, the reference (Pan et al., 2019b) uses a data-driven method, namely, reinforcement learning, to enable their congestion control scheme to learn an optimal control strategy automatically. The data-driven method is also a promising approach for facilitating JPAS in obtaining an optimal scheduling policy. In future work, we will extend JPAS to a data-driven and congestion-aware scheduling scheme with the assistant of congestion-sensing techniques like (Pan et al., 2014) (Pan et al., 2019a) (Pan et al., 2019b). In addition, the co-design of flow scheduling, job scheduling, and job placement will be explored.

Author contribution statement

Pan Zhou: Conceptualization, Methodology, Software, Writing. Xinshu He: Software., Shouxi Luo: Writing., Hongfang Yu: Supervision. Gang Sun: Writing, Supervision.

Acknowledgment

This research was partially supported by the National Key Research and Development Program of China (2019YFB1802800), PCL Future Greater-Bay Area Network Facilities for Large-scale Experiments and Applications (PCL2018KP001), Fundamental Research Funds for the Central Universities (2682019CX61), and China Postdoctoral Science Foundation (2019M663552).

References

Agarwal, Ashish, 2019. Static automatic batching in TensorFlow. In: International Conference on Machine Learning. ICML.

Agarwal, Saksham, Rajakrishnan, Shijin, Narayan, Akshay, Agarwal, Rachit, Shmoys, David, Amin, Vahdat, 2018. Sincronia: near-optimal network design for coflows. In: Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication. ACM, pp. 16–29.

Amazon EC2 Elastic GPUs. https://aws.amazon. com/ec2/elastic-gpus/.

Bai, Wei, Chen, Li, Chen, Kai, Han, Dongsu, Tian, Chen, Wang, Hao, 2017. PIAS: practical information-agnostic flow scheduling for commodity data centers. IEEE/ACM Trans. Netw. 25 (4), 1954–1967.

Bao, Yixin, Peng, Yanghua, Wu, Chuan, 2019. Deep learning-based job placement in distributed machine learning clusters. IEEE INFOCOM 2019-IEEE Conference on Computer Communications. IEEE.

Bengio, Yoshua, Goodfellow, Ian, Courville, Aaron, 2017. Deep Learning, vol. 1. MIT press.

Bottou, Léon, Curtis, Frank E., Nocedal, Jorge, 2018. Optimization methods for large-scale machine learning. SIAM Rev. 60 (2), 223–311.

Carbin, Michael, et al., 2019. "The Lottery Ticket Hypothesis: Finding Sparse, Trainable Neural networks." the Seventh International Conference on Learning Representations. ICLR.

Chan, Kwok Ho, Babiarz, Jozef, Baker, Fred, 2006. Configuration guidelines for Diff Serv service classes. https://tools.ietf.org/html/rfc4594.

Chen, Chen, Wang, Wei, Li, Bo, 2019. Round-robin synchronization: mitigating communication bottlenecks in parameter servers. -IEEE Conference on Computer Communications, 2019. IEEE. IEEE INFOCOM.

Dell'Amico, Matteo, 2019. Scheduling with Inexact Job Sizes: the Merits of Shortest Processing Time First arXiv preprint arXiv:1907.04824.

Dogar, Fahad R., Karagiannis, Thomas, Ballani, Hitesh, Antony, Rowstron, 2014. Decentralized task-aware scheduling for data center networks. In: ACM SIGCOMM Computer Communication Review, vol. 44. ACM, pp. 431–442, 4.

Domhan, Tobias, Tobias Springenberg, Jost, Hutter, Frank, 2015. Speeding up automatic hyperparameter optimization of deep neural networks by extrapolation of learning curves. In: Twenty-Fourth International Joint Conference on Artificial Intelligence.

Dunnmon, Jared, Christopher, Ré, et al., 2019. Cross-modal data programming enables rapid medical machine learning, p. 11101 arXiv preprint arXiv:1903.

Golovin, Daniel, Benjamin, Solnik, Moitra, Subhodeep, Kochanski, Greg, Karro, John, Sculley, D., 2017. Google vizier: a service for black-box optimization. In: Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. ACM, pp. 1487–1495.

Gu, Juncheng, Chowdhury, Mosharaf, Shin, Kang G., Zhu, Yibo, Jeon, Myeongjae, Qian, Junjie, Liu, Hongqiang, Guo, Chuanxiong, 2019. Tiresias: a GPU cluster manager for distributed deep learning. In: 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19), pp. 485–500.

Guo, Yingya, et al., 2019. Joint optimization of tasks placement and routing to minimize Coflow Completion Time. J. Netw. Comput. Appl. 135, 47–61.

Hazelwood, Kim, Bird, Sarah, Chintala, Soumith, Diril, Utku, Dzhulgakov, Dmytro, Mohamed, Fawzy, et al., 2018. Applied machine learning at Facebook: a datacenter infrastructure perspective. In: IEEE International Symposium on High Performance Computer Architecture (HPCA). IEEE, pp. 620–629, 2018.

He, Yanzhang, et al., 2019. Streaming end-to-end speech recognition for mobile devices. In: IEEE International Conference on Acoustics, Speech and Signal Processing. ICASSP.

Jeon, Myeongjae, Shivaram Venkataraman, Junjie Qian, Amar Phanishayee, Wencong Xiao, and Fan Yang. Multi-tenant GPU Clusters for Deep Learning Workloads: Analysis and Implications. Vol. vol. 13. MSR-TR-2018, 2018.

Jeon, Myeongjae, Venkataraman, Shivaram, Phanishayee, Amar, Qian, Junjie, Xiao, Wencong, Fan, Yang, 2019. Analysis of large-scale multi-tenant GPU clusters for DNN training workloads arXiv preprint arXiv:1901.05758.

Jiang, Jie, Yu, Lele, Jiang, Jiawei, Liu, Yuhong, Cui, Bin, 2017. Angel: a New Large-Scale Machine Learning System. National Science Review, p. nwx018, 2017.

Jin, Haifeng, Song, Qingquan, Hu, Xia, 2018. Efficient neural architecture search with network morphism, p. 10282 arXiv preprint arXiv:1806.

Jose, Lavanya, et al., 2019. A distributed algorithm to calculate max-min fair rates without per-flow state. Proceedings of the ACM on Measurement and Analysis of Computing Systems 3 (2), 21.

Kubernetes, 2017. https://kubernetes.io.

Li, Mu, David G. Andersen, Park, Jun Woo, Smola, Alexander J., Ahmed, Amr, Josifovski, Vanja, Long, James, 2014. Eugene J. Shekita, and Bor-Yiing Su. "Scaling distributed machine learning with the parameter server. In: 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14), pp. 583–598.

Li, Ziyang, Bai, Wei, Chen, Kai, Han, Dongsu, Zhang, Yiming, Li, Dongsheng, Yu, Hongfang, 2017. Rate-aware flow scheduling for commodity data center networks. In: IEEE INFOCOM 2017-IEEE Conference on Computer Communications. IEEE, pp. 1–9.

Li, Mingfan, et al., 2019. Improving the performance of distributed MXNet with RDMA. Int. J. Parallel Program. 47 (3), 467–480.

Luo, Shouxi, et al., 2020. Efficient file dissemination in data center networks with priority-based adaptive multicast. In: IEEE Journal on Selected Areas in Communications. IEEE.

IBM, "IBM Spectrum MPI," 2017. [Online]. Available: https://www.ibm.com/us-en/marketplace/spectrum-mpi

Mao, Hongzi, et al., 2019. Learning scheduling algorithms for data processing clusters. In: Proceedings of the ACM Special Interest Group on Data Communication. ACM.

Mushtaq, Aisha, et al., 2019. Datacenter congestion control: identifying what is essential and making it practical. Comput. Commun. Rev. 49 (3), 32–38.

Pan, Shengli, et al., 2014. End-to-end measurements for network tomography under multipath routing. IEEE Commun. Lett. 18 (5), 881–884.

Pan, Shengli, et al., 2019a. General identifiability condition for network topology monitoring with network tomography. Sensors 19 (19), 4125.

Pan, Shengli, et al., 2019b. A Q-learning based framework for congested link identification. IEEE Internet Things J. 6 (6), 9668–9678.

Peng, Yanghua, Bao, Yixin, Chen, Yangrui, Wu, Chuan, Guo, Chuanxiong, 2018. Optimus: an efficient dynamic resource scheduler for deep learning clusters. In: Proceedings of the Thirteenth EuroSys Conference. ACM, p. 3.

Peng, Yanghua, et al., 2019a. A generic communication scheduler for distributed DNN training acceleration. In: Proceedings of the 27th ACM Symposium on Operating Systems Principles. ACM.

Peng, Yanghua, et al., 2019b. DL2: a deep learning-driven scheduler for deep learning clusters arXiv preprint arXiv:1909.06040.

Riquelme, Carlos, Tucker, George, Jasper Snoek, 2018. Deep bayesian bandits showdown: an empirical comparison of bayesian deep networks for thompson sampling arXiv preprint arXiv:1802.09127.

Saeed, Ahmed, et al., 2019. Eiffel: efficient and flexible software packet scheduling. NSDI 532–540.

Sun, Peng, Wen, Yonggang, Yan, Shengen, 2017. Towards distributed machine learning in shared clusters: a dynamically-partitioned approach. In: IEEE International Conference on Smart Computing (SMARTCOMP). IEEE, pp. 1–6, 2017.

The CIFAR-10 dataset, 2009. https://www.cs.toronto.edu/~kriz/cifar.html.

Tian, Bingchuan, et al., 2018. Scheduling coflows of multi-stage jobs to minimize the total weighted job completion time. In: IEEE INFOCOM 2018-IEEE Conference on Computer Communications. IEEE.

Tung, Frederick, Mori, Greg, 2018. Deep neural network compression by in-parallel pruning-quantization. In: IEEE Transactions on Pattern Analysis and Machine Intelligence.

GPU on Google Cloud. https://cloud.google. com/gpu/.

GPU-Accelerated Microsoft Azure. https://www.nvidia.com/en-us/data-center/gpu-cloud-computing/microsoft-azure/.
Wang, Wei, Gao, Jinyang, Zhang, Meihui, Wang, Sheng, Chen, Gang, Ng, Teck Khim, Ooi, Beng Chin, Shao, Jie, Reyad, Moaz, 2018. Rafiki: machine learning as an analytics service system. Proc. VLDB Endowment 12 (2), 128–140.
Wang, Zhiliang, et al., 2019. Efficient scheduling of weighted coflows in data centers. IEEE Trans. Parallel Distr. Syst. 30, 9.
Wu, Zifeng, Shen, Chunhua, Anton Van Den Hengel, 2019. Wider or deeper: revisiting the resnet model for visual recognition. Pattern Recogn. 90, 119–133.
Xiao, W., Bhardwaj, R., Ramjee, R., Sivathanu, M., Kwatra, N., Han, Z., Patel, P., Peng, X., Zhao, H., Zhang, Q., Yang, F., 2018. Gandiva: introspective cluster scheduling for deep learning. In: 13th USENIX Symposium on Operating Systems Design and Implementation OSDI, vol. 18, pp. 595–610.
Xie, Saining, Girshick, Ross, Dollár, Piotr, Tu, Zhuowen, He, Kaiming, 2017. Aggregated residual transformations for deep neural networks. In: Proc. Of the 30th IEEE Conference on Computer Vision and Pattern Recognition (CVPR).
Zeng, Kai, Ding, Shifei, Jia, Weikuan, 2019. Single image super-resolution using a polymorphic parallel CNN. Appl. Intell. 49 (1), 292–300.
Zhang, Haoyu, Logan, Stafman, Andrew Or, Michael, J., Freedman, 2017. SLAQ: quality-driven scheduling for distributed machine learning. In: Proceedings of the 2017 Symposium on Cloud Computing. ACM, pp. 390–404.
Zhang, Jian, Ding, Shifei, Zhang, Nan, 2018. An overview on probability undirected graphs and their applications in image processing. Neurocomputing 321, 156–168.

Pan Zhou is a PhD candidate of Communication and Information systems at University of Electronic Science and Technology of China. His research areas include networks, cloud computing, distributed systems and machine learning.

Xinshu He is a master student of Communication and Information systems at University of Electronic Science and Technology of China. His research areas include deep learning and distributed systems.

Shouxi Luo received his B.S. degree in Communication Engineering and Ph.D. degree in Communication and Information System from University of Electronic Science and Technology of China in 2011 and 2016, respectively. From Oct. 2015 to Sep. 2016, he was an Academic Guest at the Department of Information Technology and Electrical Engineering, ETH Zurich. His research interests include data center networks and software-defined networks.

Hongfang Yu received her B.S. degree in Electrical Engineering in 1996 from Xidian University, her M.S. degree and Ph.D. degree in Communication and Information Engineering in 1999 and 2006 from University of Electronic Science and Technology of China, respectively. From 2009 to 2010, she was a Visiting Scholar at the Department of Computer Science and Engineering, University at Buffalo (SUNY). Her research interests include network survivability, network security and next generation Internet.

Gang Sun is an associate professor of Computer Science at University of Electronic Science and Technology of China (UESTC). His research interests include network virtualization, cloud computing, high performance computing and cyber security.