Green, Yellow, Yield: End-Host Traffic Scheduling for Distributed Deep Learning with TensorLights

Xin Sunny Huang
Rice University

Ang Chen
Rice University

T. S. Eugene Ng
Rice University

***Abstract*—The recent success of Deep Learning (DL) in a board range of AI services has led to a surging amount of DL workloads in production clusters. To support DL jobs at scale, the parameter server (PS) architecture is the most popular approach for distributing the computation in a compute cluster. Concurrent DL jobs consisting of PS tasks and worker tasks are typically launched on available compute nodes by a cluster resource manager to ensure high cluster resource utilization. As a PS needs to distribute model updates to every remote worker, its communication has very large fan-out. We observe that network contention among colocated PSes would cause stragglers among workers, resulting in application performance degradation and resource under-utilization. To mitigate the straggler effect, we propose *TensorLights*, which introduces traffic prioritization at host NICs to manage traffic contention among PSes. We evaluate TensorLights experimentally and show that it effectively mitigates stragglers, improves the average completion time of DL applications by up to 31%, and increases resource utilization. TensorLights is highly practical as it provides benefits without needing changes to the DL software stack.**

I. Introduction

Today, deep learning (DL) has gained tremendous success in a wide variety of AI services. Besides classic machine learning problems such as image recognition [1] and language processing [2], deep learning has also been applied to problems in system security [3], network congestion control [4], database index structures [5], power grid scheduling [6], and a long list of other challenging problems that conventionally rely on carefully-designed heuristics or manual control. As a result, DL has become a surging workload in today's compute clusters. Training a complex model on a large dataset usually requires intense computation and network communication. To achieve a high accuracy, these applications usually run for a long time, ranging from hours [7] to days [8] and even months [9]. The DL workload will continue to grow, and therefore, improving the efficiency of these emerging applications has become a crucial challenge in a modern compute cluster.

To support computation on complex models and large-scale datasets, training DL models in a distributed mode is beneficial in several ways. For certain DL jobs, a single machine is insufficient due to limited compute power and storage [8]. Distributed DL can also exploit the parallelism in a DL job to speed up the application [8, 10, 11]. Even for a job that has similar performance running on a single machine as running on distributed machines, dividing the compute workloads into multiple machines helps the job to start earlier, because in a production environment, it is usually easier to find a collection of machines with the required resources for the divided tasks, and less likely to find a single machine with sufficient capacity for the aggregate resource demand of a DL job, which is typically large [12]. With distributed DL, divided tasks from concurrent DL jobs are then launched on any available compute nodes by a cluster resource manager to ensure high cluster resource utilization [13].

Distributed training using the parameter server (PS) architecture has gained popularity due to its architectural simplicity and scalability, and it is widely supported by a range of distributed DL frameworks [14, 15, 16, 17, 18]. We will discuss this architecture in more detail in Section II. The PS architecture leverages a logically centralized PS to work with a number of remote workers. The PS communicates with all remote workers back and forth to exchange the model parameters, so the communication at the PS is usually intense with high fan-in and high fan-out. Therefore, the communication efficiency of the PS plays a crucial role in the performance of a distributed DL application.

Because of the heavy communication at the PS, traffic contention among PSes from concurrent DL jobs would lead to stragglers among workers and therefore performance degradation of the applications (Section III). We analyze the straggler effect and identify one cause of such inefficiency to be the conventional packet scheduling policy at the host NIC (Section IV). Based on these observations, we propose *TensorLights*, a traffic scheduler for the end-host NIC to mitigate the straggler effect for distributed DL. In contrast to the conventional first-come-first-serve traffic scheduling policy, TensorLights applies application-aware traffic prioritization to ensure that workers of the same job progress at a similar pace, so as to avoid imbalanced waiting time among workers that leads to stragglers. This not only improves application performance, but also increases machine utilization. In addition to using traffic priorities to mitigate stragglers, TensorLights also provides fairness among concurrent applications, which is desirable for monitoring the accuracy progress of a set of related DL models being trained concurrently.

TensorLights is a lightweight approach readily deployable in modern clusters without having to modify applications, the cluster scheduler, or the underlying hardware. This is in contrast to existing works [13, 19, 20, 21, 22], which require modifications of the distributed DL stacks. We implement TensorLights in Linux and evaluate its performance

in a 21-server testbed. The experimental results show that TensorLights improves the average job completion time for distributed DL applications by up to 31%, and leads to higher machine utilization. Together with its lightweight design, TensorLights provides a practical strategy to support distributed DL efficiently at scale in a cluster.

II. Background

We begin with a brief overview on the life cycle of a distributed DL job. Then we discuss how a large amount of distributed DL jobs are supported in a cluster at scale.

PS vs. worker: Distributed DL based on the parameter server architecture leverages a logically centralized PS to manage the model parameters. Figure 1 illustrates an example workflow of this architecture. The PS manages the trained model and communicates with a number of workers, which concurrently work on the training data. Each worker calculates, based on the worker's local copy of the model, the gradients of model parameters for a batch of samples, i.e. a small fraction of the training data. *Local batch size*[1] describes the number of samples contained in a batch processed by one worker, which typically ranges from a few to hundreds of samples [20, 23], depending on the worker's compute power.

Communication patterns: The communication between a worker and the PS proceeds in *iterations* (Figure 1). Each worker will wait for the *model update* from the PS to start computing on a new local batch with the latest model reported by the PS. As soon as a worker finishes processing a local batch, it sends the *gradient update* to the PS. Upon receiving the gradient update, PS modifies its local copy of the model to include the updated gradients. The model update and gradient update to/from a worker in each iteration are typically of the same size, i.e. the total data size of the model parameters.

In the common case of synchronous training, the PS applies a *barrier* to wait for the gradient updates from *all* workers, before sending back the model update to *any* worker. The barrier, which marks the end of one iteration, ensures the models sent to all workers are identical and reflect the aggregated gradients from all workers in the last iteration. It is also possible to train a model asynchronously: after receiving the gradient update from a worker, the PS immediately sends back the latest model to the worker, so that the worker may continue to process the next local batch without waiting for other workers. This allows each worker to proceed at their own pace, but gradients computed by each worker are usually based on a different version of the model. Consequently, the asynchronously trained models may be less accurate [24] due to the potential staleness of models and gradient updates across workers. In this work, we focus on synchronous training which usually results in more accurate models [24, 25].

Local vs. global steps: The progress of a worker can be measured by the number of local batches processed, i.e. *local steps*. The *global step* of a DL job describes the total number of local steps performed by all workers. In the example of Figure 1, at t_3 (or t_5), wk_1 (or wk_2) finishes processing one local batch, and therefore the local step of wk_1 (or wk_2) increases to 1. At t_4 (or t_6), the PS receives gradient updates from wk_1 (or wk_2) and updates its local model. The global step increases to 1 at t_4, and to 2 at t_6.

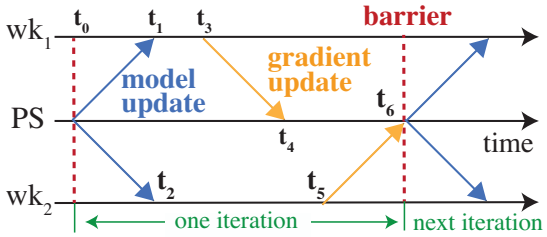

Fig. 1: Workflow of a distributed DL application based on the parameter server architecture. The example DL job has one PS and two remote workers, wk_1 and wk_2.

Distributed DL at scale: To search for the best configuration of a DL model, a common practice is to run a large amount of concurrent training jobs of the same model on the same dataset, so that each individual job is configured with a different combination of model configurations, such as the parameter initialization methods and the strategies to randomize selected parameters to avoid overfitting. This process is commonly known as the *grid search*. The grid search is highly time-consuming and resource intensive, because the number of possible combinations is exponential.

In production environments, the cluster scheduler, such as YARN [26], Borg [12], or Mesos [27], is used to manage the executions of a large amount of distributed DL jobs at scale [13]. The scheduler picks a machine for a task by considering a wide variety of factors, such as the task's resource request and the actual machine usage. To maximize utilization, a machine may be scheduled to host a mixture of different tasks. To achieve fault tolerance, tasks from the same job are usually spread on different machines across power and failure domains [12]. When making task scheduling decisions, the cluster scheduler focuses on the task's resource requirement, such as the usage of CPU, memory and storage, and it is usually agnostic of the task's functionality (e.g. PS vs. worker) in the job; thus, colocation of PS tasks can naturally occur. Cluster designs customized for distributed DL are under active research [13, 22]. For example, a recent proposal [22] develops a specialized PS that can service multiple DL jobs concurrently to improve the communication efficiency for model/gradient updates.

III. Performance Measurements

In this section, we characterize the application performance when multiple concurrent DL jobs coexist in a cluster.

Testbed: Our testbed consists of 21 hosts connected to one Ethernet switch. Each host has 128 GB RAM and six 3.5 GHz dual hyper-threaded CPU cores. All links are 10 Gbps. Our

[1] The *batch size* and *mini-batch size* in prior literature are vaguely defined, which also refer to the total number of samples processed in one iteration. For clarity, we use *local batch size*.

TABLE I: Index of different possible PS placements. The placement with a higher index tends to be more uniform.

Index	PS Placement	Index	PS Placement
#1	21	#5	5, 5, 5, 6
#2	5, 16	#6	4, 4, 4, 4, 5
#3	10, 11	#7	3, 3, 3, 3, 3, 3, 3
#4	7, 7, 7	#8	1, ..., 1 (all ones)

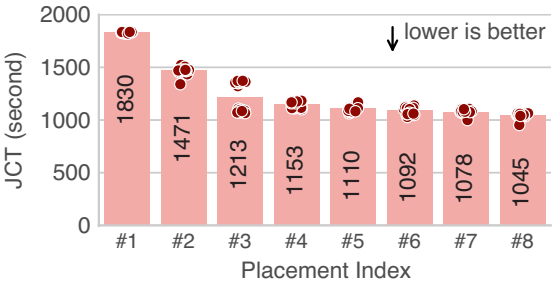

Fig. 2: Job Completion Time (JCT) of concurrent DL jobs under various placements. Scatters show the completion time for individual jobs. Bar heights indicate the average completion time of concurrent jobs in the same experiment. Definitions of placement indexes are in Table I.

measurements require a controlled network environment to be meaningful; this requirement prevents us from conducting our experiments in a public cloud environment (e.g. AWS [28] or Azure [29]) that has high network interference [30].

Workload: We focus on the performance of grid search as described in Section II. In each experiment, we deploy 21 concurrent DL jobs. Each job runs synchronous training for the ResNet-32 [7] model on the CIFAR-10 [31] dataset with a local batch size of 4, until the global step reaches 30000. Each worker reads the dataset from the local disk on the host. There are one parameter server and 20 workers for each job. In a more general case where one DL job has multiple PSes, each PS communicates with remote workers in a similar way. We launch all concurrent jobs almost simultaneously at the beginning of each experiment with a small delay (0.1 second) between consecutively launched jobs to avoid overloading RPC or SSH connections in a short time. We use TensorFlow (r1.7) [32] and instrumented its public benchmarks [33] with basic support for our measurements, such as disabling unnecessary checkpoints and adding operators in the execution graph to measure barrier wait time. Our instrumented benchmark along with the job configurations used for our measurement is open source [34].

Task placement: In a production environment, a machine can either host the PS task or the worker task of a DL job. In our experiment, each DL job has one parameter server on one of the 21 hosts, and its 20 workers are distributed evenly on the rest of 20 hosts, so that each host has one worker task. We have evaluated a range of possible placements of PSes from concurrent DL jobs, as shown in Table I. For M concurrent

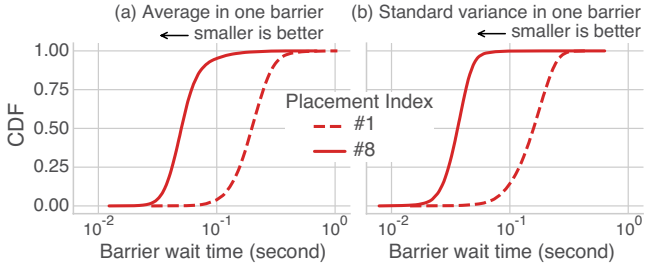

Fig. 3: Distribution of barrier wait time under two placements. Each sample describes (a) the average or (b) the standard variance of waiting time for one barrier among workers of the same job. Samples include all concurrent jobs under a specific placement.

jobs each with a PS, the placement of PS tasks is displayed in the form of $m_1, ..., m_K$, where $M = \sum_{k=1}^{K} m_k$, which means m_k jobs colocate their PSes on the same host. For example, the first placement of "21" indicates colocating all concurrent PSes on the same host, which resembles the architecture design in [22] where one logical node serves as the shared PS for all concurrent DL jobs. The last placement of "1, ..., 1" (twenty-one 1's) means that each host has one PS.

Observation #1: Impact of placement on performance. Figure 2 highlights that the performance of concurrent distributed DL jobs can be highly impacted by the placement of PS tasks. To quantify this sensitivity, we define the *performance gap* as the percentage difference between the best and the worst performance among all possible placements in our study. Figure 2 shows the performance gap in terms of average job completion time can be as large as 75% due to placement of PS tasks. Because a PS needs to distribute model updates to all workers, colocated PSes would contend for the outbound bandwidth on the same physical link to transmit the model updates. The placement of PSes would result in different levels of contention among the model update traffic from concurrent PSes on the same host.

We observe that distributed DL jobs are sensitive to the network bandwidth contention for two reasons. First, the model update traffic is bursty, because the PS will wait for the gradient updates from *all* workers and then send out model updates to *all* workers at once. The bursty traffic pattern would lead to heavy delays when multiple traffic bursts overlap in time. Second, because a worker depends on the model update from the PS to begin computation, a worker may become a straggler if its model update is delayed as a result of traffic contention at the PS side. Because of the barrier enforced in each iteration, any *one* straggling worker will delay the whole iteration including the progress of *all* other workers in the same job. This effect not only leads to performance degradation of applications, but also inefficient machine utilization.

Observation #2: Stragglers under network contention. To quantify the straggler effect, we measure the waiting time for each barrier among all workers of the same job. As an

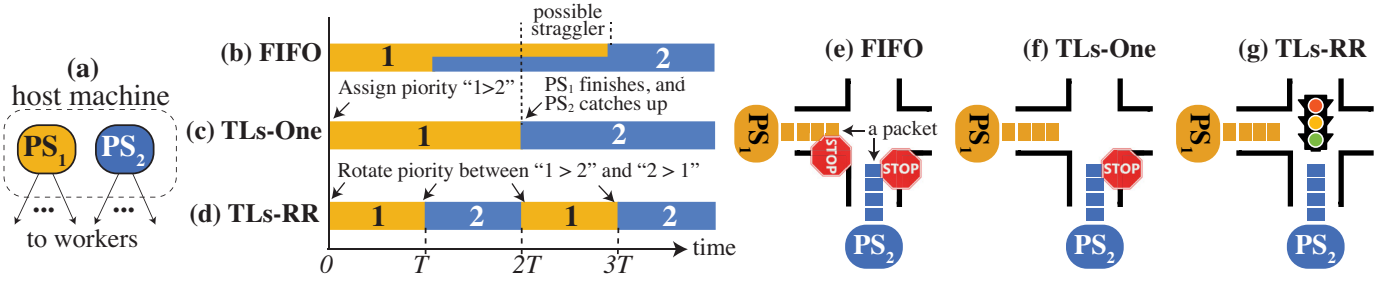

Fig. 4: Scheduling model update traffic from two concurrent DL jobs. (a) The PSes of two DL jobs are colocated on the same host machine. (b) FIFO. Workers receiving the tail part of the model updates from PS_1 are likely to become stragglers. (c) TLs-One. (d) TLs-RR. In (b, c, d), the bar area of the same color represents a job's total amount of model updates sent to all workers in the same iteration. (e, f, g) Analogy to traffic regulation signals on the road.

example, in a DL job at 30k global steps with 20 workers, each worker has finished 30k/20 = 1500 local steps and has correspondingly waited for 1500 barriers. We measure the elapsed time between a worker entering the barrier and exiting the barrier, and calculate the average (or the standard variance) of the elapsed waiting time for a specific barrier among all workers of the same DL job. Figure 3 shows the distribution of the average (or standard variance) of barrier wait time under two extremes of placement in our study.

Heavy traffic contention at the PS leads to longer barrier wait time. Figure 3a shows the average wait time under placement #1 (with heavier contention) is 3.71× of that under placement #8 (with milder contention). Increased barrier wait time leads to increased life span of the applications.

Besides the *average* barrier wait time presented above, we are also interested in the *variance* of barrier wait time, which is an indicator of stragglers. Stragglers would force the peer workers to wait for a longer time while the stragglers themselves usually wait for a shorter time, which results in a high variance of barrier wait time. Figure 3b shows the variance of barrier wait time under placement #1 is 4.37× of that under placement #8. In sum, the traffic contention at the PS not only leads to application delays, but also more stragglers.

IV. TensorLights

In this section, we first explain why the straggler effect can easily arise under the conventional FIFO scheduling policy. Then we propose TensorLights to mitigate the straggler effect.

A. Stragglers under FIFO Scheduling Policy

Conventional packet scheduling at the host NIC applies a first-come-first-serve (FIFO) policy to service concurrent transmissions. However, when model updates from two or more different jobs overlap in time, the contention among PSes may introduce random delay in the model updates for one or more workers of the same job, which would later result in worker stragglers. Stragglers are harmful for a DL job because *all* workers will be delayed by *a single* straggler.

Figure 4a shows an example of contention between two PSes of concurrent jobs. Figure 4b demonstrates the straggler effect under FIFO, where workers receiving the tail part of the model updates from PS_1 are likely to become stragglers. Unfortunately, the job running on PS_2 would not benefit from sharing the bandwidth in advance with PS_1, because workers that have received earlier parts still have to wait for other workers receiving the tail part from PS_2. Note that such straggler effect can be more harmful when more concurrent jobs are contending, because a job can be easily delayed due to any one straggler, and multiple jobs can be *simultaneously* delayed if they each have one or a few stragglers.

B. Mitigating Stragglers with Priority Scheduling

The worker stragglers are a result of contention due to model updates at the PS, so we focus on regulating the model update traffic to reduce stragglers. In contrast to the FIFO scheduling policy, we propose to *assign a distinct priority for the model update traffic of the same DL job*. Our proposal is grounded in the following insights.

Insight #1: Job-level traffic priority mitigates the straggler effect by reducing the variance of barrier wait time. When a distinct priority is assigned to the model update traffic for all workers of the same DL job, workers of a high-priority job would generally wait for less time, and workers of a low-priority job would generally wait more. Across jobs with different priorities, the variance of barrier wait time is reduced. With less variance of barrier wait time, stragglers are reduced because workers of the same job are expected to wait for similar lengths of time.

Insight #2: The communication patterns of a distributed DL job make it easy to take advantage of the job-level traffic priority. The life span of a DL job typically ranges from thousands to millions of iterations [35]. During the lifetime of a DL job, the traffic patterns between a PS and the workers, such as the end points and traffic sizes, remain the same across iterations. This means that, once the job-level priority is deployed, a DL job would continue to benefit over multiple iterations. Besides, the communication of a PS is highly symmetric, because the model update in the outbound direction is equal in data size to the gradient update in the inbound direction. Enforcing the priority for model updates

at the PS also indirectly controls the progress of workers and thus the pace of their gradient updates, which implicitly helps to schedule the inbound traffic to the PS.

In addition, our proposal also comes with several practical advantages. First, because we aim at regulating the model update traffic, the implementation only requires local configuration of traffic priority at the host running PSes (details in Section V). Second, this approach does not require global coordination or modifications to the application, the cluster scheduler or the hardware. Third, manipulating priority is work-conserving, so that PSes may still exploit the full link capacity.

We do not constrain how priorities are assigned. For example, in grid search where all jobs have the same size of data in each model update, a random priority assignment can be adopted. In other cases with concurrent DL jobs of various sizes of model update, a higher priority can be assigned to a job with a smaller model update, so as to avoid head-of-line blocking from a job with larger model update.

In the batch processing mode which allows different progress of concurrent DL jobs, it suffices to reconfigure priority assignment upon job arrival and departure. We refer to such mode of priority assignment as **T**ensor**L**ight**s-One**, or TLs-One. Figure 4c illustrates the benefits of TLs-One. The model updates from PS_1 are prioritized, so that all receiving workers may progress as soon as possible, reducing the chance that any worker becomes a straggler. Model updates from PS_2 yield but the transmission would expect to finish at the same time as in Figure 4b under FIFO.

C. Achieving Fairness with Round-Robin Priority Assignment

We have motivated the benefits of TLs-One. Nevertheless, applying strict priority can harm fairness among concurrent jobs, because one job would always be promoted or demoted over another. However, fairness is desirable in grid search, because when all search instances have made similar progress, a DL engineer may compare the accuracy performance of concurrent grid-search instances. To achieve fairness among concurrent DL jobs while using priority to mitigate straggler, we propose to rotate the priority assignment for the contending jobs once every time interval T. We refer to this version of TensorLights as **T**ensor**L**ight**s-R**ound **R**obin, or TLs-RR.

Figure 4 illustrates TLs-One and TLs-RR for two jobs with their PSes on the same host. At t_0, two jobs are assigned priority $1 > 2$ under both TLs-One and TLs-RR. Under TLs-RR at T, the priority is reconfigured to $2 > 1$, and later priority changes back to $1 > 2$ at $2T$, and so on. TLs-RR resembles the traffic lights on the road, which rotates the signals of "pass" and "yield" ("yield" instead of a complete "pause" as in the real traffic lights) for the contending traffic (Figure 4g). In contrast to the traffic lights, a "stop" sign asks vehicles to yield to the contending traffic, and the tie between two 'stop" signs is broken by vehicle arrival time. FIFO enforces a "stop" sign for the "vehicles" (packets) from both "directions" (PS_1 and PS_2), which is less efficient because both jobs would suffer from stragglers due to later arrival model updates.

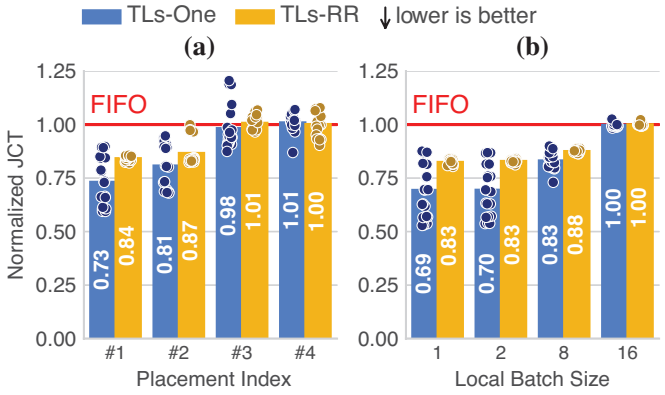

Fig. 5: Normalized Job Completion Time (JCT). The presented JCT is normalized over that of the same job under FIFO. Scatters show the normalized completion time for individual jobs. Bar heights indicate the average of normalized completion time of concurrent jobs in the same experiment. Lower value is better. (a) Normalized JCT with different placements, under local batch size of 4. (b) Normalized JCT with different local batch sizes, under placement #1. In (b), we use the local batch size as a knob to change the intensity of traffic contension, and a smaller local batch size leads to heavier traffic contension. Definitions of placement indexes are in Table I.

TLs-RR mitigates straggler effects by allowing strict priority during a short interval T. Meanwhile, rotating priority assignments allows each job to make fair progress over a longer time scale. Because the lifespan of a DL job usually lasts for hours to days, an interval T in the scale of seconds to minutes is sufficient to achieve fair progress among concurrent jobs.

V. Performance Evaluation

We evaluate TensorLights under the same settings as described in our previous measurements (Section III).

Implementation: We enforce traffic priority with the hierarchical token bucket (htb) available in the `tc` tool on Linux. In a TensorFlow application, the TCP port numbers for the PS and workers are fixed for the lifetime of the application. Therefore, we assign priority for a job based on its PS's TCP port number, so that the job's model update traffic is mapped to a specific priority band. `tc` controls outbound traffic at the sender, so we only need to configure `tc` on the hosts with contending PSes and leave other hosts unchanged to limit the amount of `tc` reconfigurations. Ideally, a host with contending PSes should assign a distinct priority for each job. However, `tc` only supports a limited number of priority bands. In our experiments, we only use up to six distinct priority bands, and multiple jobs may share the same priority band.

We have implemented both TLs-One and TLs-RR based on `tc`. For TLs-RR, the reconfiguration interval is T=20 seconds, which is sufficient for the DL jobs in our experiments that runs for thousands of seconds. The baseline of comparison is the default FIFO policy, which does not involve any `tc` configurations. TensorLights is open source [34].

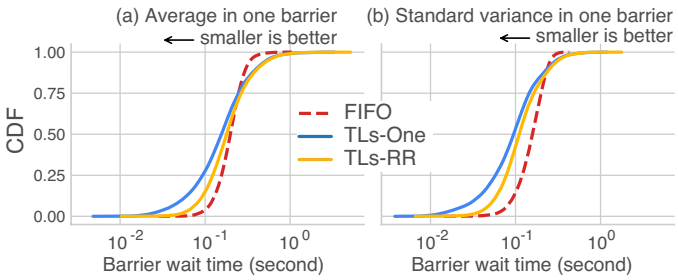

Fig. 6: Distribution of barrier wait time of various network scheduling policies under placement #1. This figure reads in a similar way as Figure 3. Smaller value is better. TensorLights mitigates straggler effect by reducing the variance of barrier wait time.

Result #1: TensorLights improves the average completion time of DL applications. Figure 5a compares the system efficiency in terms of the average job completion time of concurrent DL applications. Compared with FIFO, TLs-One reduces the average job completion time by up to 27%. Under TLs-RR that achieves job fairness, the average job completion time is reduced by up to 16%. For the placement with less model update traffic contention, i.e. placement #4 and above in our study, TensorLights achieves comparable performance as FIFO. Because TensorLights is work-conserving, it improves performance under heavy traffic contention, while preserving performance in other cases with milder contention. Note that under the priority assignment of TLs-One, jobs with higher priority tend to finish earlier and others finish later, which results in progress differences across concurrent jobs. TLs-RR, on the other hand, achieves fair progress among concurrent jobs while improving the system efficiency by mitigating stragglers.

Result #2: TensorLights improves system efficiency by effectively reducing straggler. Figure 6 quantifies the straggler effects with the barrier wait time that we used in the previous measurements (Section III). As expected, the span of average barrier wait time in Figure 6a is larger under both TLs-One and TLs-RR, because jobs with higher priority tend to wait less while jobs with low priority tend to wait longer. The average barrier wait time are comparable under the three network scheduling policies.

The variance of barrier wait time is also important, as it speaks directly to the straggler effect. Figure 6b shows that TensorLights effectively reduces the variance of barrier wait time. Compared with FIFO, the average (or median) variance of barrier wait time under TLs-One is reduced by 26% (or 40%), and under TLs-RR by 15% (or 30%). These results confirm our previous observations that the priority strategy taken by TensorLights is an effective approach to reduce the variance of barrier wait time and mitigate stragglers (Section IV).

Result #3: TensorLights can increase machine utilization. We further quantify the efficiency improvement in terms of machine utilization. To precisely capture the machine utilization under steady state of the system, we define an *active window* as a time period of fixed length when all concurrent jobs are active. In our study, the active window is between the 100th and the 1250th second after the launch of concurrent jobs. For each host in our testbed, we measure the userspace CPU utilization with `vmstat`, and the network interface utilization with `ifstat`. Table II shows the normalized machine utilization during the *active window* under placement #1. Compared with FIFO, TLs-One improves the average CPU utilization by 4% on the host supporting PS and by 13% on the hosts supporting workers. In terms of a host's network utilization, TLs-One achieves an improvement of 20% on both the inbound and outbound directions. We also observe similar improvement under TLs-RR. These results confirm our previous observations that reducing stragglers helps to improve machine utilization (Section III). The utilization improvement would translate into fruitful cost savings in a large-scale cluster [12].

TABLE II: Normalized utilization of CPU and network interface under the placement #1. A host's normalized utilization is the average utilization during the "active window", normalized over that under FIFO scheduling. The presented utilization is the average of all hosts of a specific type. Larger value is better.

Resource type	Host type	TLs-One	TLs-RR
CPU	PS	1.04×	1.03×
	Worker	1.13×	1.12×
Network Inbound	All	1.20×	1.21×
Network Outbound	All	1.20×	1.21×

Result #4: TensorLights can effectively handle contention. The impact of network scheduling is more significant when contention among PSes become more intense. To understand the application performance under various levels of traffic contention, we test a range of local batch sizes on placement #1. A smaller local batch size requires less computation overhead for a worker in each local step, resulting in more frequent model (and gradient) updates and therefore more intense traffic contention. In Figure 5b, under more intense traffic contention due to smaller local batch size, TLs-One (or TLs-RR) enlarges the improvement over FIFO in terms of average job completion time to 31% (or 17%). In other words, TensorLights becomes even more effective for very heavy traffic contention scenarios.

The recent trends of scaling up DL applications continue to introduce more traffic contention in the cluster network. At the individual application level, to speed up computation, recent trends are to deploy more workers and computation accelerators such as GPUs [11], both of which would lead to even heavier contention due to larger amount of data exchanged per iteration and faster iterations. At the cluster level, contention would also increase as the amount of DL workloads continues to grow. With more traffic contention expected under these trends, an efficient traffic scheduling policy would play an

increasingly crucial role to improve application performance and cluster utilization to support distributed DL at scale.

VI. Related Work

Optimizing communication for distributed DL: A large body of prior works have explored various techniques to improve the communication efficiency for distributed DL. At the level of an individual application, existing works focus on hiding communication cost in the presence of computation by (1) increasing the fraction of computation with a larger local batch size [36], (2) reducing the communication cost with compressed gradient updates [37, 38], and (3) increasing overlap between communication and computation with better ordering of model parameters in transmission [19, 20, 21]. TensorLights is an inter-job traffic scheduler and thus is complementary to these works – it aims to improve the communication efficiency of distributed DL in a cluster setting. At the level of supporting multiple DL applications in a cluster, [13] proposed a customized cluster scheduler that optimizes the performance of distributed DL jobs by dynamically adjusting the numbers and placement of parameter servers and workers in each job. [22] proposed a new software/hardware architecture to accelerate PS communications. The above prior works require significant modifications to the DL stack at various layers. In contrast, TensorLights is an end-host traffic scheduler that does not require changes to the applications, the cluster scheduler, or the hardware.

Mitigating stragglers in synchronization barrier: Distributed applications often need to apply a global barrier to synchronize parallel workers. For example, in a MapReduce-like application [39, 40, 41], an implicit barrier exists among all mappers because a reducer needs to wait for the output from all mappers before the next round of computation. A common technique to mitigate stragglers is speculative scheduling [24, 42], where a few extra "backup" workers are added to mitigate stragglers. This approach consumes extra compute resources, and also requires modifying the application to coordinate with the extra workers. In contrast, TensorLights consumes no extra resources and requires no changes to applications.

Network scheduling for distributed applications: Several recent works [43, 44, 45, 46] have demonstrated the benefits of leveraging application-level traffic requirements using the abstraction of Coflows [47] to improve the communication efficiency for MapReduce-like applications [39, 40, 41, 48]. The communication patterns of a MapReduce-like application are different from those of a distributed DL application. For example, a MapReduce-like application usually organizes its communication in several consecutive stages, and the traffic patterns, such as the end points involved and traffic sizes, can be different across stages. In a distributed DL application, the traffic pattern for each repeated step is fixed during the lifetime of the application. Furthermore, a DL model can take up to millions of steps to train [35], while a MapReduce-like application usually consists of tens of communication stages [49]. To accommodate the dynamics in Coflows, the Coflow-based solutions generally apply global coordination to orchestrate Coflow transmissions. In contrast, TensorLights only requires traffic priority configurations at local hosts.

VII. Future Work

Two interesting future directions of this work would be optimizing communication efficiency for distributed DL (1) at the cluster scheduler and (2) at the transmission layer.

At the level of the cluster scheduler, an effective approach to mitigate contention due to model updates is to better schedule the placement of PS tasks before starting a DL job. The scheduler may be notified with the task type (e.g. PS vs. worker) as well as the job type (e.g. distributed DL vs. MapReduce-like), so that special treatment is applied when scheduling PS tasks. This approach does not require aggressively adjusting the application configurations at run time as in [13]. Nevertheless, under novel cluster architectural designs [22] where one logical node serves as the shared PS for multiple DL jobs, the PS location becomes fixed.

At the level of the transmission layer, instead of using conventional network transmission protocols such as TCP (e.g., as in PyTorch [15]) or gRPC/HTTP2 (e.g., as in TensorFlow [14]), a customized protocol to coordinate model/gradient updates may be deployed, so that the update traffic would be orchestrated by a logically centralized coordinator, which is similar to the strategies in [43, 50]. However, this approach incurs non-trivial coordination overhead. It further requires modifications to various layers of the cluster stack, such as modifying the application to use the new protocol, modifying the host machines to install necessary libraries, and even modifying the hardware to provide support in the switches. Many existing solutions [43, 44, 45, 50, 51] apply transmission rate control at the sender to orchestrate traffic. However, inaccurate rate allocation would lead to lower network utilization.

In summary, these future directions also require careful designs to handle the subtle interactions among the applications, the scheduler, and the underlying network architecture.

VIII. Conclusions

We have presented TensorLights, a traffic scheduler at end-host NICs to mitigate the straggler effect for distributed DL applications under traffic contention. TensorLights acts like "traffic lights" for the model update traffic from the PS, so that contending DL applications take turns to pass or yield, which is helpful to improve all applications' barrier waiting efficiency. Testbed evaluations show that TensorLights improves the average job completion time for distributed DL by up to 31%, increasing the utilization of CPUs and network bandwidth. As the trends of scaling up DL applications continue to introduce more traffic contention in the cluster network, an efficient traffic scheduler such as TensorLights would play an increasingly crucial role to support distributed DL efficiently at scale.

ACKNOWLEDGMENT

We thank the BOLD Lab members and the anonymous reviewers for their useful feedback. This research is sponsored by the NSF under CNS-1422925, CNS-1718980, CNS-1801884, and CNS-1815525.

REFERENCES

[1] The ImageNet dataset. http://image-net.org/about-stats.
[2] P. Rajpurkar, J. Zhang, K. Lopyrev, and P. Liang, "SQuAD: 100,000+ questions for machine comprehension of text," in *Empirical Methods in Natural Language Processing (EMNLP)*, 2016.
[3] M. Abadi and D. G. Andersen, "Learning to protect communications with adversarial neural cryptography," *arXiv preprint arXiv:1610.06918*, 2016.
[4] A. Valadarsky, M. Schapira, D. Shahaf, and A. Tamar, "Learning to route," in *ACM HotNets*, 2017.
[5] T. Kraska, A. Beutel, E. H. Chi, J. Dean, and N. Polyzotis, "The case for learned index structures," in *ACM SIGMOD*, 2018.
[6] R. Evans and J. Gao. (2016) Deepmind AI reduces Google data centre cooling bill by 40%. https://deepmind.com/blog/deepmind-ai-reduces-google-data-centre-cooling-bill-40/.
[7] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *IEEE CVPR*, 2016.
[8] J. Dean, G. S. Corrado, R. Monga, K. Chen, M. Devin, Q. V. Le, M. Z. Mao, M. Ranzato, A. Senior, P. Tucker, K. Yang, and A. Y. Ng, "Large scale distributed deep networks," in *Advances in Neural Information Processing Systems (NIPS)*, 2012.
[9] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton *et al.*, "Mastering the game of Go without human knowledge," *Nature*, 2017.
[10] P. Goyal, P. Dollár, R. Girshick, P. Noordhuis, L. Wesolowski, A. Kyrola, A. Tulloch, Y. Jia, and K. He, "Accurate, large minibatch SGD: Training ImageNet in 1 hour," *arXiv preprint arXiv:1706.02677*, 2017.
[11] H. Mikami, H. Suganuma, Y. Tanaka, Y. Kageyama *et al.*, "ImageNet/ResNet-50 training in 224 seconds," *arXiv preprint arXiv:1811.05233*, 2018.
[12] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, "Large-scale cluster management at Google with Borg," in *EuroSys*, 2015.
[13] Y. Peng, Y. Bao, Y. Chen, C. Wu, and C. Guo, "Optimus: An efficient dynamic resource scheduler for deep learning clusters," in *EuroSys*, 2018.
[14] TensorFlow. https://www.tensorflow.org/.
[15] PyTorch. http://pytorch.org/.
[16] Apache MXNet. https://mxnet.apache.org/.
[17] P. Moritz, R. Nishihara, S. Wang, A. Tumanov, R. Liaw, E. Liang, M. Elibol, Z. Yang, W. Paul, M. I. Jordan *et al.*, "Ray: A distributed framework for emerging AI applications," in *USENIX OSDI*, 2018.
[18] The Microsoft Cognitive toolkit (CNTK). https://www.microsoft.com/en-us/cognitive-toolkit/.
[19] 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 *USENIX ATC*, 2017.
[20] S. H. Hashemi, S. A. Jyothi, and R. H. Campbell, "TicTac: Accelerating distributed deep learning with communication scheduling," in *SysML*, 2019. [Online]. Available: http://arxiv.org/abs/1803.03288
[21] A. Jayarajan, J. Wei, G. A. Gibson, A. Fedorova, and G. Pekhimenko, "Priority-based parameter propagation for distributed DNN training," in *SysML*, 2019.
[22] 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 *ACM SoCC*, 2018.
[23] N. S. Keskar, D. Mudigere, J. Nocedal, M. Smelyanskiy, and P. T. P. Tang, "On large-batch training for deep learning: Generalization gap and sharp minima," *arXiv preprint arXiv:1609.04836*, 2016.
[24] J. Chen, R. Monga, S. Bengio, and R. Jozefowicz, "Revisiting distributed synchronous SGD," in *International Conference on Learning Representations (ICLR) Workshop Track*, 2016. [Online]. Available: https://arxiv.org/abs/1604.00981
[25] 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 *USENIX OSDI*, 2016.
[26] 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 *ACM SoCC*, 2013.
[27] 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 *USENIX NSDI*, 2011.
[28] Amazon Web Services (AWS). https://aws.amazon.com.
[29] Microsoft Azure Cloud Computing. https://azure.microsoft.com/.
[30] G. Wang and T. E. Ng, "The impact of virtualization on network performance of Amazon EC2 data center," in *IEEE INFOCOM*, 2010.
[31] A. Krizhevsky, "Learning multiple layers of features from tiny images," University of Toronto, Tech. Rep., 2009.
[32] TensorFlow on Github. https://github.com/tensorflow/tensorflow.
[33] TensorFlow benchmarks. https://github.com/tensorflow/benchmarks.
[34] TensorLights on Github. https://github.com/TensorLights.
[35] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the inception architecture for computer vision," in *IEEE CVPR*, 2016.
[36] F. N. Iandola, M. W. Moskewicz, K. Ashraf, and K. Keutzer, "FireCaffe: Near-linear acceleration of deep neural network training on compute clusters," in *IEEE CVPR*, 2016.
[37] D. Alistarh, D. Grubic, J. Li, R. Tomioka, and M. Vojnovic, "QSGD: Communication-efficient SGD via gradient quantization and encoding," in *Advances in Neural Information Processing Systems (NIPS)*, 2017.
[38] W. Wen, C. Xu, F. Yan, C. Wu, Y. Wang, Y. Chen, and H. Li, "TernGrad: Ternary gradients to reduce communication in distributed deep learning," in *Advances in Neural Information Processing Systems (NIPS)*, 2017.
[39] Apache Hadoop. http://hadoop.apache.org/.
[40] Apache Tez. http://tez.apache.org.
[41] Apache Hive. http://hive.apache.org.
[42] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica, "Improving MapReduce performance in heterogeneous environments," in *USENIX OSDI*, 2008.
[43] M. Chowdhury, Y. Zhong, and I. Stoica, "Efficient coflow scheduling with Varys," in *ACM SIGCOMM*, 2014.
[44] M. Chowdhury and I. Stoica, "Efficient coflow scheduling without prior knowledge," in *ACM SIGCOMM*, 2015.
[45] H. Zhang, L. Chen, B. Yi, K. Chen, M. Chowdhury, and Y. Geng, "CODA: Toward automatically identifying and scheduling coflows in the dark," in *ACM SIGCOMM*, 2016.
[46] S. Agarwal, S. Rajakrishnan, A. Narayan, R. Agarwal, D. Shmoys, and A. Vahdat, "Sincronia: Near-optimal network design for coflows," in *ACM SIGCOMM*, 2018.
[47] M. Chowdhury and I. Stoica, "Coflow: A networking abstraction for cluster applications," in *ACM HotNets*, 2012.
[48] Apache HDFS. https://hortonworks.com/apache/hdfs/.
[49] T. Chiba and T. Onodera, "Workload characterization and optimization of TPC-H queries on Apache Spark," in *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2016.
[50] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu *et al.*, "B4: Experience with a globally-deployed software defined WAN," in *ACM SIGCOMM*, 2013.
[51] A. Kumar, S. Jain, U. Naik, A. Raghuraman, N. Kasinadhuni, E. C. Zermeno, C. S. Gunn, J. Ai, B. Carlin, M. Amarandei-Stavila *et al.*, "BwE: Flexible, hierarchical bandwidth allocation for WAN distributed computing," in *ACM SIGCOMM*, 2015.