A Generic Communication Scheduler for Distributed DNN Training Acceleration

Yanghua Peng*†, Yibo Zhu†, Yangrui Chen*, Yixin Bao*, Bairen Yi†, Chang Lan†
Chuan Wu*, Chuanxiong Guo†
{yhpeng, yrchen, yxbao, cwu}@cs.hku.hk*, {zhuyibo, yibairen.byron, lanchang, guochuanxiong}@bytedance.com†
The University of Hong Kong*, ByteDance Inc.†

Abstract

We present ByteScheduler, a generic communication scheduler for distributed DNN training acceleration. ByteScheduler is based on our principled analysis that partitioning and rearranging the tensor transmissions can result in optimal results in theory and good performance in real-world even with scheduling overhead. To make ByteScheduler work generally for various DNN training frameworks, we introduce a unified abstraction and a *Dependency Proxy* mechanism to enable communication scheduling without breaking the original dependencies in framework engines. We further introduce a Bayesian Optimization approach to auto-tune tensor partition size and other parameters for different training models under various networking conditions. ByteScheduler now supports TensorFlow, PyTorch, and MXNet without modifying their source code, and works well with both Parameter Server (PS) and all-reduce architectures for gradient synchronization, using either TCP or RDMA. Our experiments show that ByteScheduler accelerates training with all experimented system configurations and DNN models, by up to 196% (or 2.96× of original speed).

CCS Concepts • **Computer systems organization** → *Distributed architectures*; *Neural networks*.

Keywords ML frameworks, communication scheduling

1 Introduction

Deep Neural Networks (DNNs) have been extensively used for a wide range of applications, such as Computer Vision, Natural Language Processing, etc. Training DNNs, however, are time-consuming tasks, mainly due to large volumes of data and growing DNN model sizes. The most common way to scale out and accelerate DNN training is data parallelism (§2). Unfortunately, its performance is often far from linear speed-up, due mainly to the communication overhead. As a large online service provider, in many of our internal and publicly available training workloads, communication often consumes a significant portion of total training time. This is also echoed by recent literature [9, 18, 39].

Consequently, many different communication acceleration approaches have been proposed and integrated into popular frameworks, including TensorFlow [6], PyTorch [28], MXNet [12], with drastically different implementations. For example, one can use RDMA to replace TCP, while the RDMA implementations are quite different among frameworks. Or, one can use ring-based all-reduce, either from one of several different MPI implementations or NCCL [4] by NVIDIA, to replace Parameter Servers (PS). Nevertheless, they share the same goal – speeding up *each individual message*.

Recently, a new direction to accelerate distributed DNN training, *i.e.*, communication scheduling, has been explored [18, 21]. The idea is to change the transmission order of different DNN layers, in order to better hide the communication overhead and achieve training speed improvement *without* affecting computation results. For example, Jayarajan *et al.* [21] empirically show that a speed-up of 25% − 66% can be achieved with a certain framework (MXNet with PS and 1Gbps to 10Gbps TCP network). Independently, we got a similar observation with our own initial implementation and deployment. We will further explain the details in §2.2.

In this paper, we will show that priority-based communication scheduling, combined with tensor partitioning, is not only the *theoretically optimal* strategy (§4) assuming no system overhead, but also *generic*. It can accelerate most, if not all, popular frameworks, both PS (synchronous or asynchronous) and all-reduce gradient synchronization, different network transports (RDMA or TCP), and any combinations of them. This could have allowed the whole community/industry to use the expensive GPU cycles more efficiently! Meanwhile, with a unified scheduling module across different frameworks, future developers and researchers can experiment with their ideas much easier in a wide range of settings and achieve a larger impact.

The work of Yanghua Peng, Yangrui Chen and Chuan Wu was supported in part by a ByteDance Research Collaboration Project. Yanghua Peng is also supported by SOSP 2019 Student Scholarship from the ACM Special Interest Group in Operating Systems.

SOSP '19, October 27–30, 2019, Huntsville, ON, Canada

ACM ISBN 978-1-4503-6873-5/19/10…$15.00
https://doi.org/10.1145/3341301.3359642

Unfortunately, the existing designs [18, 21] are far from this vision because of two main reasons.

First, as mentioned above, there are many different combinations of frameworks and network stacks. However, existing communication scheduler designs are just for one: P3 [21] modified several layers in MXNet framework and its PS implementation, and TicTac [18] modified TensorFlow and its PS implementation totally differently from P3. To use them in another framework with different communication methods, *e.g.,* all-reduce, one may have to re-do everything.

In contrast, we design a generic communication scheduling layer that presents a framework-agnostic and communication method-agnostic abstraction, benefiting a broader audience. Building such a generic scheduling layer, however, is non-trivial. For example, different frameworks decide the order of both computation and communication by themselves. We must have a generic way to schedule the order of communication, while complying with the framework engines, without heavy modifications in the respective engines (otherwise it cannot be generic). Also, some frameworks introduce global barriers that prevent communication scheduling (*e.g.*, TensorFlow, PyTorch). Thus, in addition to the unified scheduling abstraction, we propose two designs, *Dependency Proxy* and *layer-wise out-of-engine dependencies*, to address the above challenges, respectively.

Second, existing work does not adapt well to a wide range of system setups. Different communication methods have their own implications on system parameters, such as tensor partition sizes. For PS architecture, it is better to slice tensors into smaller pieces, so that *push* and *pull* can better utilize bi-directional network bandwidth. However, for all-reduce, each partition incurs a synchronization cost among all workers, so small partition sizes may lead to performance penalty. Moreover, different DNNs, bandwidths, and even the number of workers also affect the optimal system parameters.

To address these issues, we present the analysis of how different system setups may impact the final performance and system parameter choices, and propose an auto-tuning algorithm based on Bayesian Optimization. It automatically searches for the best system parameters, like tensor partition sizes and maximum sender credits. The auto-tuning helps the core scheduling algorithm adapt to different models, communication architectures and hardware configurations.

We evaluate our design with MXNet, TensorFlow, and PyTorch, with PS or all-reduce gradient synchronization, using RDMA or TCP transports with different physical bandwidths. Popular CNN and RNN models are tested. The results are promising – the performance improvement is up to 196%, for all different combinations of frameworks/networks/models. In addition, with the auto-tuning for different run-time environments, our design outperforms P3 [21] (in its only scenario, MXNet with PS TCP) by 28% to 43%. All these require zero or little code change to the framework engines, and no more than 5-line change to the user code.

We summarize our contributions as follows:

▹ We design a generic tensor scheduling framework, ByteScheduler, that separates tensor scheduling and partitioning from various training frameworks, gradient update architectures, and network protocols, without modifying their implementations. Our design works for TensorFlow and PyTorch which introduce global barriers between successive iterations.

▹ Our analysis shows that ByteScheduler's scheduling algorithm is optimal when there is no system overhead. The insight is that assigning higher priority to the layers near the DNN input maximizes the overlap with forward computation of the next iteration. With given system overhead, the performance gap from the optimum is bounded.

▹ We identify key system parameters, *i.e.*, partition size and credit size, and design Bayesian Optimization-based auto-tuning. It makes ByteScheduler adapt to various training models and system configurations.

We have open-sourced our implementation, including the core scheduler and plugins for different frameworks [2]. We hope to continue evolving it with the community, since many future directions can be investigated. Examples include variable tensor partition sizes, supporting dynamic models, cross-job co-scheduling, etc. ByteScheduler decouples communication scheduling from the computation frameworks, so that it can be made framework-agnostic and optimized separately. We believe that ByteScheduler, being open-sourced and widely generic, can significantly facilitate future research and development in related directions.

2 Background and Motivation

2.1 DNN Training and Data Parallelism

In deep learning, a DNN model is trained by iterating a large dataset many times (or "epochs"), to minimize a loss function.

Forward and backward propagation. Within each epoch, the dataset is partitioned into mini-batches. In each iteration, one mini-batch travels through the DNN model layer-by-layer and generates a loss. This process is called *forward propagation* (FP). After FP, the gradients are calculated from the last layer to the first layer, and this process is called *backward propagation* (BP). The gradients are then used to update model parameters based on some optimization algorithm, *e.g.*, Stochastic Gradient Descent (SGD). Then, the training moves on to the next mini-batch, starting from FP again.

Data parallelism. Due to the complexity of DNN models and large datasets, the training is often not able to be finished within a short time, *e.g.*, it takes 115 minutes to finish training ResNet50 [19] on a DGX-1 machine with 8 V100 GPUs [3]. Data parallelism is a popular strategy for scaling DNN training across many devices. It partitions the dataset onto multiple compute devices ("workers"), where each worker shares the same model parameters. Gradients from all workers are then aggregated before applied to update model parameters. Network communication is involved

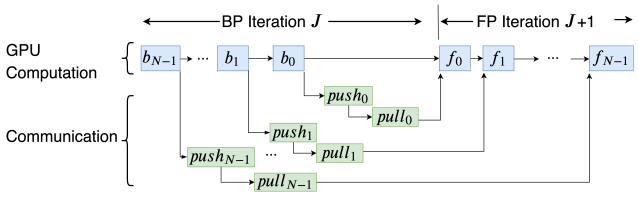


Figure 1. Layer-wise computation and communication in distributed DNN training, *e.g.*, MXNet PS.

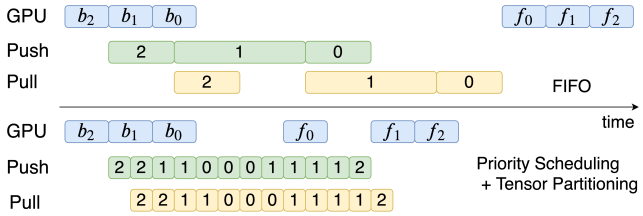


Figure 2. A contrived example showing performance gain with a better scheduling strategy (than FIFO) and tensor partitioning.

in this process, usually using the parameter server architecture [15, 24] or collective routines (*e.g.*, all-reduce) [32].

Parameter Server. A parameter server (PS) is a logically separate device that stores global parameters and provides a key-value interface to workers. Typically, data parallelism with PS has the following steps: (a) each worker computes the gradients using its local data partition and sends them to PS (*push*); (b) PS sums the gradients across workers and updates its parameters (*update*); (c) Workers synchronize parameters with PS (*pull*). A PS architecture enables better fault tolerance and more flexible parameter synchronization.

All-reduce. All-reduce is a collective operation that reduces the target arrays with a specified binary operator (*e.g.*, sum, max) in all processes to a single array and broadcasts the result to all processes. In DNN training, all-reduce computes the sum of gradients across workers, and then each worker updates its parameters accordingly locally.

2.2 Communication Scheduling

Computation-communication dependency DAG. In distributed DNN training, computation and communication of tensors form a dependency DAG (Directed Acyclic Graph). In the DAG, the forward and backward propagation is or can be linearized as a chain of computation across layers (*e.g.*, by grouping or coalescing multiple operators or tensors [36]). Let f_i, b_i, $push_i$ and $pull_i$ be the FP, BP, push and pull of layer i, respectively. Figure 1 shows the layer-wise dependencies of MXNet with PS architecture between two iterations: f_i depends on f_{i-1} and $pull_i$, $pull_i$ depends on $push_i$, $push_i$ depends on b_i, and b_i depends on b_{i+1}. To finish DNN training is to finish such a DAG (spanning all iterations).

Scheduling the order of communication. By default, ML framework engines execute communication operations in a FIFO order, because the underlying communication stack, either PS or all-reduce, TCP or RDMA, is inherently based on FIFO queues. This is shown in Figure 1: since $push_0$ and $push_1$ both require upload bandwidth, $push_1$ gets executed before $push_0$; similarly, $pull_1$ could be executed before $pull_0$.

However, this is sub-optimal. Because f_0 must be executed before f_1, it is better to finish $pull_0$ as soon as possible. In the case that $pull_1$ takes a long time and blocks $pull_0$, FIFO strategy delays $pull_0$, and hence delays the start time of f_0 and the whole iteration process.

Communication scheduling [18, 21] is a good solution to this problem. In the example above, we can prioritize $push_i$ over $push_j$ if $i < j$, and do the same for pull operations. Then the forward propagation of the next iteration can start earlier, and potentially speed up the training.

Tensor partitioning. In DNNs, each layer includes one or multiple tensors (*e.g.*, the "pushed" gradients and "pulled" parameters). Communication scheduling is commonly carried out for such tensors, while tensors in the same layer can have the same scheduling priority (*i.e.*, push tensors have the same priority and the same for all pull tensors) and are scheduled sequentially on the respective resource (*e.g.*, upload and download bandwidth). The tensor sizes can vary significantly (*e.g.*, the smallest tensor is 256B and the largest tensor is over 400MB for VGG16 model [33]). A very large tensor, once en-queued in the communication stack, would block other tensors even if they have higher priorities. Thus, a more efficient scheduling strategy is to partition the tensors before en-queuing, and allow higher-priority tensors to jump ahead of the queue once they arrive.

Tensor partitioning also improves bandwidth utilization of bi-directional network in PS architecture. Without partitioning, the pull flow of a large tensor can start only after the push flow of the whole tensor is done. Given that the network today is usually duplex, this implies 50% bandwidth waste. Finally, partitioning tensors can mitigate load imbalance in PS architecture, especially when one or a few tensors are very large and dominate the total model size.

Potential benefits. To demonstrate the potential benefits of communication scheduling (with tensor partitioning), we show a simple and contrived illustrative example (Figure 2). The DNN has three layers of different sizes, with FP and BP consuming different time. Compared with the default FIFO transmission scheduling, the better schedule can lead to 44.4% training speed-up. Jayarajan *et al.* [21] have implemented similar scheduling strategies on MXNet PS and shown an up-to-66% training speed improvement with 10Gbps (or less) TCP networks, than FIFO.

2.3 The Opportunities and Challenges

We believe that communication scheduling is valuable to general DNN training, not just MXNet with PS and TCP. We explain the rationale and challenges below.

Opportunity 1: one unified scheduler for all. Though there are several different ML frameworks and DNN models, the most popular training jobs have similar DAG structures

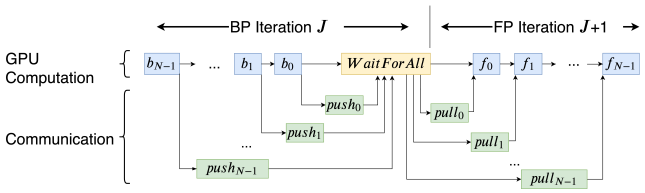
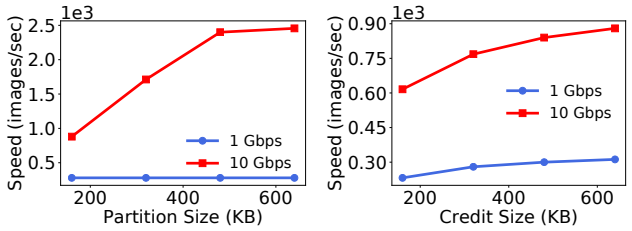


Figure 3. Distributed DNN training with an inter-iteration barrier. TensorFlow with PS is shown. PyTorch also has this barrier.



(a) Different partition sizes **(b)** Different credit sizes

Figure 4. Training VGG16 using MXNet (PS, TCP) with FIFO communication scheduling at different network bandwidth levels.

as shown in Figure 1. Most DNN models have layered structures, and computation takes place one layer after another. Even though the training frameworks have different features, they essentially run the same DAG for the same model, just with different ways (*e.g.*, APIs) of implementing the DAG.

In addition, different communication methods also fit in this DAG model – the network transport (TCP or RDMA) does not change the DAG, and an all-reduce architecture simply replaces a push and a pull in the DAG by the all-reduce operation. Intuitively, the same scheduling algorithm should also apply.

We envision a generic scheduler that can accelerate the execution of DAG by changing the order of communication operations, no matter which frameworks, which communication patterns and which network transports. Also, for the best generality and easier adoption, we seek minimal-to-none modification to existing framework engines and communication libraries.

Opportunity 2: one unified analytical foundation for scheduling algorithms. Once we realize the unified scheduler, we essentially decouple the algorithm design from the framework-related implementation details. This gives us the opportunity to formulate the scheduling problem across all system setups. In contrast, previous work was deeply coupled with specific framework implementations, and only focused on empirical results. In this paper, we show that our scheduling algorithm is not only empirically effective, but also theoretically guaranteed even with system overhead.

However, the opportunities come with challenges.

Challenge 1: be generic to different frameworks. Although the DAGs are similar in the end, how the frameworks build such DAGs and execute them is very different. Engines that support *declarative* mode decide the execution order based on DAG dependencies, while *imperative* engines run in a FIFO manner. A generic scheduler must be able to work with both types of engines, manipulating the transmission order without breaking the properties of the original engines.

In addition, existing engines are not designed with communication scheduling in mind. Therefore, some engines, like TensorFlow and PyTorch, introduce a global barrier between iterations, as Figure 3 shows. This would make any scheduling of push/pull (or all-reduce) ineffective.

Challenge 2: adapt to different run-time environments. In the real-world, scheduling and tensor partitioning have networking-related overhead. For example, there is certain overhead for sending a tensor regardless of the size of the tensor. Consequently, tensor partitioning has a performance penalty, especially if the partition size is too small. In Figure 4(a), we show the training speed with different partition sizes, with FIFO transmission scheduling. We see that the partition size affects training speed, especially in networks with larger bandwidth. P3 [21] uses a default partition size of 160KB (the leftmost points in Figure 4(a)). Such a partition size is far from optimal in a 10Gbps network when FIFO scheduling is used, though P3's scheduling out-weighs the non-optimality of partition size and delivers positive gains in the end. The same goes for *credit size* (Figure 4(b)), which is defined by us (§4.2) for filling the sending buffer in the network stack. P3 essentially uses a credit size equal to the partition size, which is again not the best.

The question is, what are the sweet spots for these parameters? They are likely to vary with many factors. For example, in Figure 4, we show that the impact of overhead is very different in 1Gbps and 10Gbps networks. In practice, the physical network bandwidth can range from 1Gbps to 200Gbps, and users use either PS or all-reduce as gradient synchronization method, and TCP or RDMA as the transport. Furthermore, we find that various DNN models have different optimal partition/credit size values because they have different model structures and sizes. We will further show its analytical complexity in §4.1.

A generic scheduling framework must be able to adapt to all these different run-time environments.

3 Design

3.1 Architecture

To make ByteScheduler generic, the key question is – **Which layer should ByteScheduler be implemented in?** From the closest to user to the lowest level, ML frameworks and communication stacks include: 1) user code that declares DNN models, 2) framework frontend with high-level API (*e.g.*, Python frontend), 3) framework engine (which decides how to execute the DAG), 4) message-level communication library, and 5) TCP or RDMA stack. To be generic, we can not modify user code and framework engines heavily. Implementing the scheduler in message-level communication library is a good choice if in a clean slate. Unfortunately, in reality, the communication libraries are quite diverse (*e.g.*,

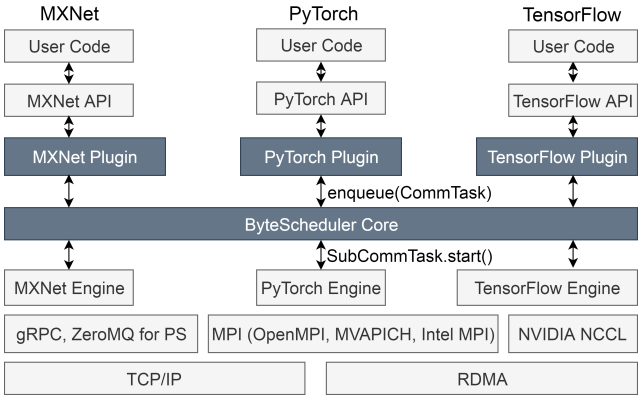


Figure 5. ByteScheduler architecture in the communication stack. The dark grey part is added by ByteScheduler.

lots of RPC and MPI implementations), with their own thread management modules. It would be almost impossible to have a single piece of code work across all of them. Under the TCP or RDMA stack, like socket and verbs, we will lose application-level information, such as message priorities.

Therefore, this leaves us the final choice – implement ByteScheduler at the high-level API implementation layer in the framework. Fortunately, this layer is usually implemented in flexible languages (*i.e.*, Python) that support class inheritance and runtime *monkey patch* [38]. Since this layer is mainly for translating users' DNN declaration into asynchronous operations, it often has a simple threading model (*e.g.*, single-threaded), indicating an easier implementation.

As shown in Figure 5, we design ByteScheduler to be just before the API layer posting communication operations to the engine. For each framework, we design a shim layer, called *plugin*, that wraps the original operation into a unified "communication task" abstraction (§3.2). Then, the *Core*, schedules the task using our (or any) algorithm. Consequently, the same piece of scheduling code would work across frameworks and communication methods. The detailed interaction between these modules is explained in the following subsections.

3.2 Unified Abstraction for Communication Tasks

ByteScheduler Core is designed to be framework-agnostic and communication method-agnostic. It accepts a unified abstraction for communication, called *CommTask* (a communication task associated with one tensor, *e.g.*, all-reduce), as input from the plugins. Except the trivial *init()* and *shutdown()* interfaces, the Core exposes a single interface to the plugins, *i.e.*, Core.enqueue(CommTask).

Each time a communication tensor arrives, the plugin wraps it as a CommTask and assigns priority before enqueuing it. For declarative engines (*e.g.*, TensorFlow), it uses topological sort to obtain the priority. For imperative engines, it assigns a monotonic increasing ID to each (gradient) tensor based on the order they are created (same as the BP order).

Once getting a CommTask, Core partitions it into SubCommTasks and decides when to send each. Based on the functionality of Core, and our observation on the most popular frameworks (TensorFlow, PyTorch, MXNet) and communication methods, we conclude that the following CommTask interfaces are sufficient and implementable in the plugins.

CommTask.partition(size): Core calls this interface to partition a CommTask into one or multiple SubCommTasks with tensors no larger than *size*, which invokes a callback in the plugin as tensor partitioning is framework-dependent. Core then schedules those partitioned CommTasks. This incurs low overhead because all popular frameworks provide zero-copy APIs for partitioning the underlying tensor.

CommTask.notify_ready(): Most ML frameworks are asynchronous – when a communication operation is posted to the engine, possibly the tensor has not been computed or ready to be sent. We let the engine use this interface to notify Core about a tensor being ready, so that Core can actually begin scheduling it. We will explain how we achieve this generically and without modifying the engines in §3.3.

CommTask.start(): Core calls this interface to let engines and the underlying communication stacks send the tensor. It has to be implemented per framework and communication method combination. Fortunately, the differences are very small – simply calling the built-in communication primitives (*e.g.*, ring all-reduce, push or pull) in the framework.

CommTask.notify_finish(): Once the communication of a tensor (all-reduce, push or pull) has been finished, the framework engine must notify Core about this, so that Core can continue scheduling more Tasks. More details are in §3.3.

We believe that one can implement most, if not all, common scheduling algorithms with these interfaces. Using these interfaces, we implement our scheduling algorithm in Core (Algorithm 1 in §4). All the above interfaces must be implemented in the plugins per framework, and start() is also per communication methods. The users can enable plugins with no more than 2 lines of codes (LoC), as shown in §5.

3.3 Interaction with Framework Engines

Next, we explain our generic approach that works with different engines to schedule the order of CommTasks.

The frameworks have their own execution engines that run the computation and communication operations. Declarative engines, *e.g.*, TensorFlow and MXNet, build dependency graphs, while imperative engines like PyTorch, run operations in a FIFO manner. Either way, the Core should not schedule a CommTask before the engine allows, and should be able to delay a CommTask while keeping original dependencies. We propose a new primitive to address this problem.

Dependency Proxy (or *Proxy*). A Proxy is an operation created by ByteScheduler. It can be posted into the framework engines, and claim dependencies from/to other operations. Declarative engines all provide direct APIs to define such dependencies since this is the fundamental feature of such engines. In the case of imperative engines, Proxy can

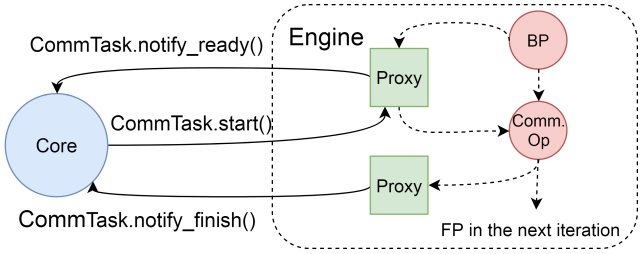


Figure 6. Interaction between Core and framework engines without global barrier, *e.g.*, MXNet. Only one DNN layer is shown (assume one tensor in the layer). Operations in squares are added by ByteScheduler. The solid arrows are direct function calls, while the dashed arrows mean dependencies.

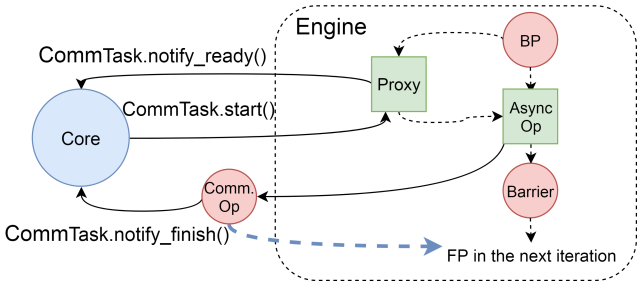


Figure 7. Interaction between Core and framework engines when a global barrier exists, *e.g.*, TensorFlow. The thick dashed arrow means dependencies enforced by ByteScheduler, not by engines.

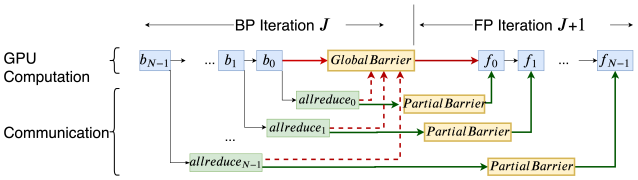


Figure 8. The new DAG with layer-wise out-of-engine dependencies (assume one layer has one tensor). Dashed arrows indicate the original dependencies removed by ByteScheduler.

be simply posted immediately before or after an operation, *e.g.*, with "hooks" in PyTorch.

If a communication operation must happen after a computation operation in the original order, *e.g.*, the computation operation generates the gradients to be sent, we will post a Proxy to the engine. It claims to depend on the finish of the computation operation, and also to be precedent to the communication operation, as shown in Figure 6.

Proxy does only two things. First, once it is started by the engine, it will trigger the corresponding *CommTask.notify_ready()* via a callback, because the start of Proxy means the original precedent operations of the communication operation have been finished. Second, it will not finish until Core tells it to end via *CommTask.start()*. Because it is one precedence of the communication operation, it effectively delays the communication operation until being scheduled by Core.

Proxy has very low overhead – before Proxy ends, it is blocked by a lock and yields the CPU resources (in the case of imperative engines), or simply "finishes" without calling the *on_complete()* callback provided by the engine (in the case of declarative engines). Core will call *CommTask.start()*, which includes the engine's *on_complete()*.

For MXNet, we post another type of Proxy right after the communication operation and claim dependency on it. This Proxy only generates completion signal using *CommTask.notify_finish()* once it is started by the engine. For TensorFlow and PyTorch, we have to generate the completion signal a bit differently – see the next subsection.

3.4 Crossing the Global Barrier

As explained in §2.3, some frameworks (*e.g.*, TensorFlow and PyTorch) introduce a global barrier between two consecutive iterations. This makes scheduling largely ineffective – the global barrier waits for all communication operations to finish before moving on to the next iteration, so changing the order of communication does not matter. We cannot simply take the global barrier away, because it would require significant change on the framework and the user code.[1]

Layer-wise out-of-engine dependencies. To facilitate scheduling, we make the global barrier *not* wait for the *actual* completion of communication operations. We replace the actual communication operation by an asynchronous operation, as Figure 7 shows. The asynchronous operation will start the actual communication in the background, and returns immediately to let the global barrier pass. In other words, the actual communication operations are run *outside* the engine and computation graph, by Core, even after the barrier. At the end of each actual communication operation, we add a callback to notify Core about completion.[2]

Only moving the actual communication out of the DAG may cause incorrectness. Because neither TensorFlow nor PyTorch engines can track cross-iteration dependencies, the FP in the next iteration may start before the actual communication finishes. Hence we add another type of Proxy before the FP of each DNN layer, and let the Proxy block until Core gets CommTask.notify_finish(). Thus, the dependencies are enforced by ByteScheduler instead of by the engines. Figure 8 summarizes the design.

4 Scheduling in the Wild

Next, we first present the analysis of our core scheduling algorithm (Algorithm 1). We show that it is the optimal scheduling in an ideal scenario (*i.e.*, no system overhead), and it is within a certain bound to the ideal case in practice. Then, we explain our auto-tuning mechanism to reduce the gap between real-world performance and the theoretical optimum.

[1]The most common way to program in TensorFlow and PyTorch is to wait for each iteration's finishing in a while-loop.

[2]Fortunately, even if the operation is run outside the main engine, TensorFlow and PyTorch still allow us to add an on_complete() callback.

4.1 The Analysis of ByteScheduler Algorithm

The following theorem shows that in the ideal case with assumptions, the optimal scheduling strategy turns out to be *priority queuing*. The priorities of tensors are determined by the order of the layers in the DNN model. Detailed proof of the theorem can be found in the appendix [1].

Theorem 1. *The following priority queuing scheduler:*

- *in a PS architecture, prioritize $pull_i$ over $pull_k$, and $push_i$ over $push_k$, $\forall i < k$*
- *in an all-reduce architecture, prioritize $allreduce_i$ over $allreduce_k$, $\forall i < k$*

is the optimal solution for minimizing the time for each training iteration, if the following conditions are met:

1. (Sequential GPU computation operations) The subgraph of DAG containing only f_i and b_i is a chain (i.e., no parallel layers in the DNN).

2. (Optimal GPU scheduling) Whenever the dependencies of f_i or b_i are satisfied, GPU will run the computation operation without preemption. This is the optimal GPU scheduler because the GPU computation operations are in a chain.

3. (Tensor partition) Tensors in each DNN layer are partitioned, such that flow preemption can happen; with PS, if the push flow in a layer is only partially done before being preempted, the done part can be pulled.

4. (Infinitely small partition) Suppose the partition size is δ and $\delta \to 0$.

5. (Flow preemption) Higher priority push/pull or all-reduce can preempt lower priority push/pull or all-reduce immediately without extra overhead.

With infinitely small partition, the push flow and pull flow can start at the same time to increase pipelining in PS architecture. Minimizing the time for one training iteration is equivalent to maximizing the overlap between communication and computation. Priority queuing can lead to more potential overlap between forward computation and communication than other policies. The optimality of Theorem 1 can be proved by induction on the number of layers.

In the all-reduce architecture, all workers form a topology (usually a ring). Each tensor is partitioned (or segmented, in MPI terms) and reduced by walking the ring from different starting points. This is the same as the PS case with small partition and without preemption overhead.

Assumptions 1 and 2 are typically true for existing ML frameworks and DNN models, especially those models consisting of layers where one stacks on another. Unfortunately, assumptions 4 and 5 do not hold in practice – the preemption in scheduling cannot be as ideal as assumed, and partitions cannot be infinitely small due to system overhead (see §2.3).

Analysis with partition size and overhead. Below, we consider non-infinitely-small partition sizes. In addition to the network transmission time $t = size/bandwidth$, real-world systems usually spend extra time on handling a message, like the RPC serialization and ACK time, or synchronization time for all-reduce. We refer to this extra time as *partition overhead* (about 300us in our testbed). With the overhead, the scheduling problem is much more complicated.

It causes three kinds of extra delay, comparing its time for one training iteration with the infinitely-small partition case: 1) In PS architecture, the pull flow will start later because a partition must be pushed before being pulled. The partition size determines this delay. 2) Sending each partition will need longer time due to overhead. 3) Preemption cannot happen at any time but only when a partition transmission is done. Therefore, partition size affects the preemption granularity.

Fortunately, for the first kind of delay, we find that the gap to the ideal case is bounded by the transmission time of one partition, since the start of all push/pull flows and preemption need to wait for the ongoing partition to be transmitted first. The smaller the partition is, the closer it is to the ideal case. For the second kind of delay – system slowdown directly caused by partition overhead, we consider the worst case that the bottleneck during one iteration is the communication time. The extra delay is bounded by $O(n\theta)$ where θ is a constant partition overhead, and n is the number of partitions in all layers in the DNN.

Summing up the gaps in the above two delays, we obtain that the extra delay due to finite partition size and partition overhead, as compared to the ideal case in Theorem 1, is at most $\sum_{i=0}^{N-1}(\lfloor \frac{s_i}{\delta} \rfloor \theta) + \theta + 2\frac{\delta}{bandwidth}$ for PS, where s_i is the size of $push_i$, δ is the partition size and N is the number of layers in the DNN, and $\sum_{i=0}^{N-1}(\lfloor \frac{allreduce_i}{\delta} \rfloor \theta) + \frac{\delta}{bandwidth}$ for all-reduce, where $allreduce_i$ is the size for one all-reduce operation. This delay for one iteration is a function of partition size δ, composed by an increasing function and a decreasing function, and has a trend to decrease first and increase later. Due to the rounding operator, this function is not smooth nor differentiable everywhere, and it is hard to find the near-optimal partition size using classical optimization methods. We will explain how to tune the best partition size in §4.3.

For the last kind of delay related to preemption granularity, it is influenced even more by another system parameter, *credit size*, as explained below.

4.2 Credit-based Preemption

To realize preemption, a naïve approach is stop-and-wait: the sender keeps only one tensor unacknowledged and sends the next tensor after receiving the acknowledgement. Existing schedulers like [21] adopted this approach. However, depending on runtime environment, sending tensors one-by-one may not fully utilize network bandwidth (§2.3).

Our solution is *credit-based* preemption. It works like a sliding window and the credit is the window size. Multiple tensors (instead of one tensor) in the window can be sent concurrently (to the underlying FIFO network transmission queue). This improves network utilization due to filling the sending buffer in the network stack. However, preemption may happen less timely. Suppose while tensor 1 is being transmitted, tensors 2, 3 and 4 arrive sequentially (priorities

Algorithm 1 The scheduling algorithm in Core

Input:
commtasks ← a set of unready SubCommTasks
queue ← priority queue of SubCommTasks
credit ← credit size for credit-based preemption
unit ← partition size that is used to partition a CommTask

```
1:  /* Partition a CommTask in Core*/
2:  procedure PARTITION(task)
3:      subtasks = task.partition(unit)
4:      commtasks.add(subtasks)
5:  /* Enqueue a ready SubCommTask based on priority*/
6:  procedure READY(priority, subtask)
7:      queue.put(priority, subtask)
8:      commtasks.remove(subtask)
9:  /* Return credits when a SubCommTask is finished*/
10: procedure FINISH(subtask)
11:     credit = credit + subtask.tensor.size
12: /* Schedule SubCommTasks in the priority queue */
13: procedure SCHEDULE
14:     while True do
15:         priority, subtask = queue.get()
16:         if credit > subtask.tensor.size then
17:             credit = credit - subtask.tensor.size
18:             subtask.start()
19:         else
20:             wait until a subtask is finished and returns credits
```

$p_1 < p_2 < p_3 < p_4$). Using the existing approach, the transmission order will be 1->4->3->2 due to stop-and-wait. Using our solution with a sliding window of size 2, the order will be 1->2->4->3. Once tensor 2 arrives, it will be sent to the underlying FIFO queue though tensor 1 is still being transmitted, to improve network utilization; later when tensor 3 and 4 arrive, they can not preempt tensor 2 as the first approach, because tensor 2 has been delivered to the underlying queue before their arrival. A larger credit size leads to better bandwidth utilization but with less timely preemption. We leave it as a system parameter to balance the trade-off and will explain how to tune its best value in §4.3.

Algorithm 1 shows the preemptive scheduling algorithm in ByteScheduler Core. In one iteration, each communication operation is wrapped as a CommTask and partitioned into SubCommTasks (line 3). As the BP goes, the SubCommTasks become ready one by one and are enqueued into a priority queue (line 7). A scheduling thread runs the SCHEDULE procedure (line 13) and polls the queue constantly. Whenever the credit is enough for the highest priority SubCommTask (line 16), ByteScheduler pops the SubCommTask from the queue and starts transmission (line 18). The credit is decreased by the tensor size (line 17), and increased when the tensor's transmission is completed (line 11).

4.3 Auto-Tuning Partition Size and Credits

Credit size and partition size are two knobs that decide the trade-off between optimal scheduling and system overhead.

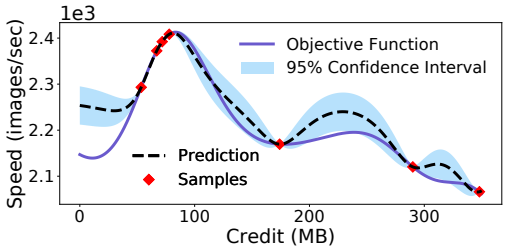


Figure 9. Bayesian Optimization example: 7 samples; tuning credit size of VGG16 in MXNet (all-reduce)

Unfortunately, formally modeling and solving this problem is hard, since we can not explicitly express training speed as a function of two knobs. We leave the two knobs undecided until runtime, and resort to runtime searching.

To search the best credit size and partition size, one simple way is to use grid search, *i.e.*, enumerate all possible combinations of credit size and partition size. But the cost of enumeration is too high and is infeasible for the continuous solution space. Some simple heuristic algorithm like SGD with momentum [30] may work when the training speed has a trend of unimodality, but we can not get the accurate gradient of the curve as it is non-parametric. The derivatives approximated by slope are "noisy" due to runtime jitters, and SGD is easy to be stuck in a local optimum.

We experimented with a few algorithms (see §6), and finalize on using Bayesian Optimization (BO) to tune credit and partition size *together*. BO does not have any limitation of the function format. The function of partition size and credit size can be treated as non-parametric. In addition, it tries to minimize the number of trials to find a near-optimal solution, which reduces the tuning cost. BO is also known to be noise-resilient since it quantitatively evaluates the uncertainty region of the performance curve [8].

We aim to optimize the training speed $D(\delta, c)$ and the decision variables are partition size δ and credit size c. The objective function $D(\cdot)$ is unknown beforehand, but its values can be observed through profiling. At each given (δ, c), the objective function value follows a distribution and we use Gaussian as it is widely accepted as a good surrogate model for BO [8]. A 95% confidence interval is associated with $D(\delta, c)$, which describes the region that $D(\delta, c)$ most likely resides in. With one more trial, the confidence interval is updated with the new observation according to the posterior distribution. With more trials, the confidence interval decreases and the estimate of $D(\cdot)$ becomes more accurate.

To minimize the number of trials, BO selects the next configuration (δ, c) by maximizing an acquisition function, which is also updated with the confidence interval. We use the most common acquisition function, Expected Improvement [11], which picks (δ, c) to maximize the expected speed improvement over the current best candidate. A BO hyperparameter EI balances the exploitation and the exploration of the searching process (we use the default value 0.1 in the experiments). In this way, BO can quickly learn the objective

function and only suggest (δ, c) in the areas that most likely contain the optimal solution. For illustration, Figure 9 shows BO's search process for credit size for training VGG16. Each sample is a pair (credit, speed).

5 Implementation Details

ByteScheduler is implemented using C++11 and Python. The Core is a common Python library for all ML frameworks. For each framework, a plugin is required for PS architecture or all-reduce architecture. The plugins are usually a mixture of C++ and Python code. The current implementation has 4279 LoC in total, with Core as the largest part (1503 LoC).

ByteScheduler Core. The Core runs a single main thread, handling events like Core.enqueue(CommTask), and all CommTask APIs. The key data structure is a priority queue of CommTasks, and the main algorithm is described in Algorithm 1. Auto-tuning algorithms are also implemented.

Every worker runs an instance of Core. However, only one worker's (with ID 0) Core runs as a master, which runs auto-tuning algorithms to decide the partition size and credit size and broadcasts them to other workers. Also, to avoid deadlocks in all-reduce, only the master Core determines the order of sending tensors and broadcasting to other workers, so that all workers can perform the same all-reduce operation simultaneously. For PS that supports asynchronous push and pull, all Cores schedule the order independently.

MXNet plugin. We implement plugins for both PS and all-reduce architectures for MXNet≥ 1.4. The PS plugin is based on MXNet's native communication abstraction, *i.e.*, KVStore. We wrap calls of KVStore APIs (*e.g.*, init, push, pull) as CommTasks and post them to ByteScheduler Core for scheduling. We will evaluate ByteScheduler performance on both TCP and RDMA for MXNet PS, shown in Sec. 6. So far, *ps-lite*, the underlying PS library of MXNet, only supports TCP communication between workers and parameter servers. Internally, we added RDMA support to ps-lite. It is out of the scope of this paper, so we omit the details.

For all-reduce, we develop the plugin based on Horovod, which wraps MXNet optimizer with Horovod DistributedOptimizer [32]. We further wrap Horovod DistributedOptimizer in our ScheduledOptimizer for MXNet, so that all-reduce operators are scheduled before executed by Horovod.

PyTorch plugin. We implement PyTorch plugin for only all-reduce architecture because PyTorch does not support PS. In order to support layer-wise out-of-engine dependencies, we add hooks (via register_hook function provided by PyTorch) to backward propagation so that all-reduce operators will be called after each gradient tensor is ready; we also add hooks to forward propagation (via register_forward_pre_hook function in PyTorch) so that forward computation of each layer will not start until the all-reduce of this layer is completed.

We again leverage Horovod, which supports all-reduce communication for PyTorch≥ 0.4 by wrapping PyTorch optimizer with Horovod DistributedOptimizer [32]. Similar to the implementation of MXNet all-reduce plugin, we wrap Horovod DistributedOptimizer in our ScheduledOptimizer.

TensorFlow plugin. The plugin is implemented for TensorFlow≥ 1.13 and PS architecture. It is an opt-in TensorFlow graph optimization module, using the TensorFlow Grappler framework [5]. Grappler supports altering run-time behavior of graph execution, such as instrumenting OpKernel implementation, adding and removing data or control dependencies of graph nodes. The TensorFlow plugin goes through the data-flow graph before its execution, and adds Dependency Proxies to schedule the timing of worker sending gradients and parameter server sending model parameters. The invocation of gradient updates is moved out of graph, *i.e.*, the global barrier is removed, as discussed in Sec. 3.4. Among all three frameworks, only TensorFlow requires 13-line code change in its engine, which delays the clean-up of communication channels when we cross the global barrier.

Auto-tuning support. Each worker needs to adjust partition size and credit size during BO tuning. For training DNNs in PS architecture, adjusting the partition size requires re-partitioning the parameters and can cause errors due to tensor shape mismatch. Hence, each time when the partition size is changed, we checkpoint and restart the training (*e.g.*, by adding a batch_end_callback hook). For all-reduce, we can change the values of the two knobs dynamically without stopping training.

Usage. Users need to add 2 lines of code in their Python≥ 2.7 program to enable ByteScheduler. Take MXNet PS architecture as an example, a user needs to wrap MXNet KVStore using our ScheduledKVS, shown as follows.

```
# After user created an MXNet KVStore object kvs
from bytescheduler.mxnet.kvstore import ScheduledKVS
kvs = ScheduledKVS(kvs)
# Continue using kvs without any further modification
```

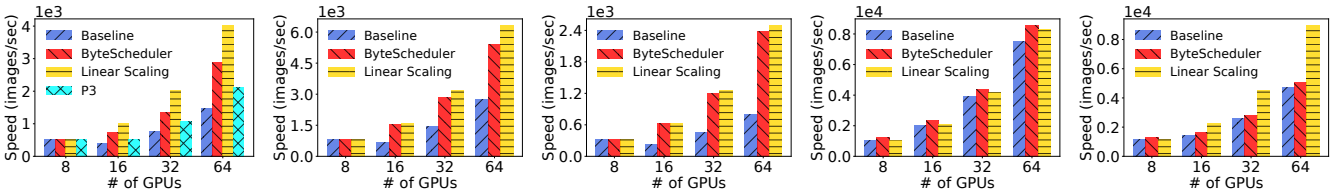
ByteScheduler may cause confusion for users due to crossing the global barrier. From our experience, the impact is little since the main metrics (*e.g.*, loss and accuracy) are calculated during forward pass and hence are not affected.

6 Evaluation

6.1 Methodology

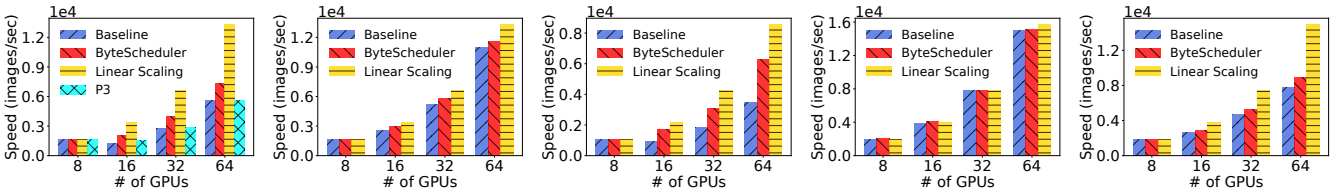
Testbed setup. Our testbed has 16 physical machines, each with 64 CPU cores, 256GB memory, 8 Tesla V100 GPUs *without* NVLinks, and 100Gbps bandwidth between any two servers using Mellanox CX-5 single-port NICs.

Benchmarks. We choose 2 CNN models, VGG16 and ResNet50, and 1 RNN model, Transformer, as our benchmark models. We have run ByteScheduler in 8 different setups: MXNet PS, MXNet NCCL, TensorFlow PS, and PyTorch NCCL, each with TCP or RDMA, respectively. ByteScehduler accelerates training in all setups. Due to space limit, we only show results in 5 setups: MXNet PS TCP, MXNet PS RDMA, TensorFlow PS TCP, MXNet NCCL RDMA, and PyTorch NCCL TCP.



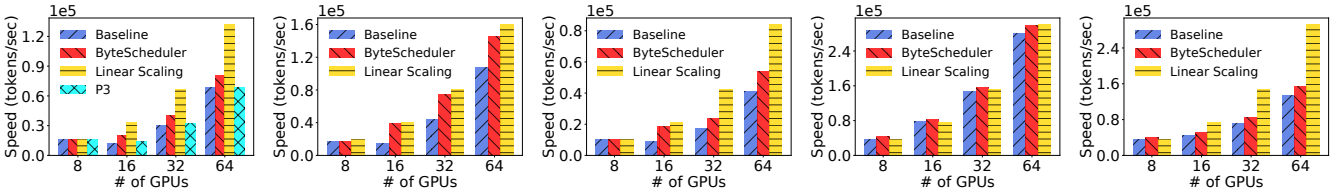
(a) MXNet, PS, TCP (80%-94%) **(b)** MXNet, PS, RDMA (97%-125%) **(c)** TensorFlow, PS, TCP (170%-196%) **(d)** MXNet, NCCL, RDMA (14%-20%) **(e)** PyTorch, NCCL, TCP (7%-13%)

Figure 10. Training VGG16 model. The numbers in parentheses are ByteScheduler speedup percentages as compared with the baseline.



(a) MXNet, PS, TCP (32%-62%) **(b)** MXNet, PS, RDMA (6%-16%) **(c)** TensorFlow, PS, TCP (70%-89%) **(d)** MXNet, NCCL, RDMA (1%-7%) **(e)** PyTorch, NCCL, TCP (11%-15%)

Figure 11. Training ResNet50 model.



(a) MXNet, PS, TCP (18%-72%) **(b)** MXNet, PS, RDMA (34%-171%) **(c)** TensorFlow, PS, TCP (31%-102%) **(d)** MXNet, NCCL, RDMA (6%-14%) **(e)** PyTorch, NCCL, TCP (11%-18%)

Figure 12. Training Transformer model.

For PS architecture, workers and parameter servers are on different machines, the number of workers is equal to the number of parameter servers, and each worker has 8 GPUs. Only the results of synchronous training is shown as we find the training speedup of asynchronous mode is similar. For all-reduce, each worker process has 1 GPU. The batch sizes of VGG16, ResNet50, and Transformer are 32, 32, 512 samples per GPU by default.

Baselines. We use the vanilla ML frameworks as baselines. We also present the linear scalability, which is used as an optimal case in many works [21, 32, 39]. It is calculated by the training speed on 1 machine (with a vanilla ML framework) multiplied by the number of machines. We use training speed (samples/sec) as the performance metric. All the reported speed numbers are averaged over 500 training iterations after a warm-up of 10 iterations.

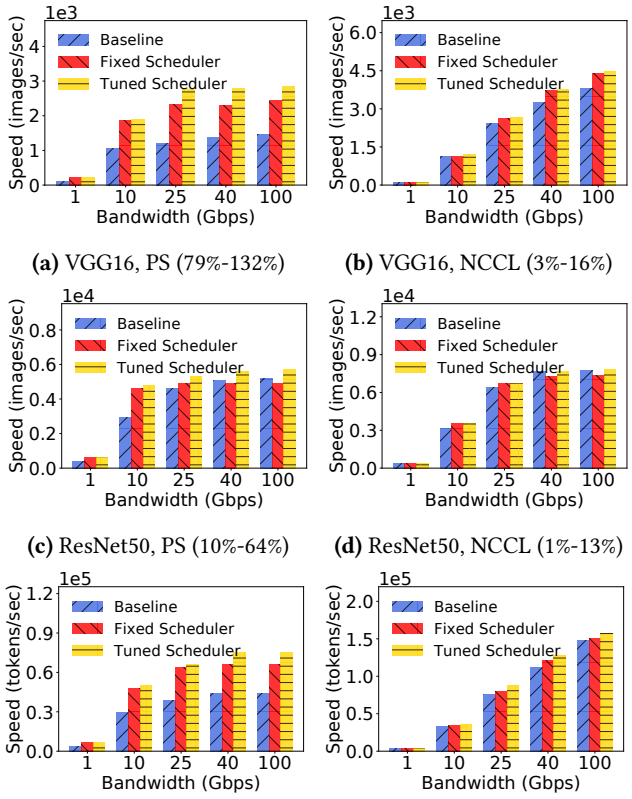
6.2 Speedup in Different Setups

Figure 10, 11 and 12 show the training speeds of baseline, ByteScheduler, and linear scaling when running the 3-model 5-case benchmarks under 100Gbps network bandwidth with the number of GPUs ranging from 8 to 64. Since each machine has 8 GPUs, the number of machines is calculated by dividing the number of GPUs by 8.

Comparison with P3. We compare ByteScheduler with P3, which only works in the MXNet PS TCP case. We use the source code available on Github and the default partition configuration of P3. We tested other partition sizes and obtained no better results. From Figure 10 (a), 11 (a) and 12 (a), we see that ByteScheduler outperforms P3 by 28% – 43% across the 3 benchmark models. The main reason is that P3 cannot utilize the bandwidth fully due to its stop-and-wait tensor transmission and system overhead as shown in §2.3. We did not compare with TicTac [18] as it is not fully open-sourced.

TCP vs. RDMA. From (a) and (b) in Figure 10, 11, 12, we see that the speedup varies largely case by case. In general, ByteScheduler has larger benefits with RDMA. We suspect that this is because the overhead due to small partition is lower with RDMA than with TCP, based on its more efficient network stack. The outlier is ResNet50 with PS RDMA: the baseline is already close to linear scaling, so there is not much room for ByteScheduler to improve.

PS vs. all-reduce. In most cases, ByteScheduler has larger speedup in PS architecture than in all-reduce. Take VGG16 training on 16 GPUs as an example. ByteScheduler achieves 124.9% and 19% improvement for MXNet PS RDMA and MXNet NCCL RDMA, respectively. The reason is that: for PS, removing the global barrier and partitioning the tensors



(a) VGG16, PS (79%-132%) **(b)** VGG16, NCCL (3%-16%)

(c) ResNet50, PS (10%-64%) **(d)** ResNet50, NCCL (1%-13%)

(e) Transformer, PS (67%-70%) **(f)** Transformer, NCCL (7%-15%)

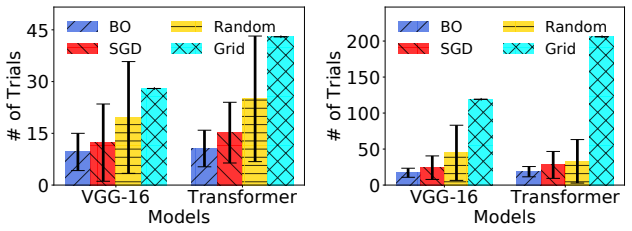
Figure 13. Speed comparison by training models with MXNet PS RDMA and MXNet NCCL RDMA under different bandwidths.

create better pipelining of push and pull and hence the communication time is also reduced significantly. For all-reduce training with NCCL RDMA, ByteScheduler is very close to linear scaling; occasionally, it even outperforms linear scaling (based on vanilla MXNet), because ByteScheduler also benefits the communication between GPUs *inside* a machine.

Different DNN models. We see ByteScheduler has larger speedup for VGG16 and Transformer than ResNet50. The low gain of ResNet50 is expected, since training it with 100Gbps RDMA is not communication-bound, *i.e.*, the ratio of communication to computation is low. We also test other models. For example, using 32 GPUs in MXNet PS RDMA setting, speedups for AlexNet [23] and VGG19 [33] are 96% and 60%, respectively. We omit other numbers due to space limit.

PS load balancing. In some cases, ByteScheduler achieves significantly higher improvement than what we expect, *e.g.*, 171% speed-up when training Transformer using 2 workers (16 GPUs) in MXNet PS RDMA. In such cases, we find that the PS are severely imbalanced in baseline (due to the imbalance in tensor sizes and the naïve tensor-to-PS assignment like round-robin by default) and smaller partitions used in ByteScheduler actually balance the PS load very well.

Working with different network bandwidths. We examine how much improvement ByteScheduler can achieve compared with the baseline under typical bandwidth settings



(a) MXNet PS RDMA **(b)** MXNet NCCL RDMA

Figure 14. Search costs of different auto-tuning algorithms. Error bars show standard deviation.

Table 1. Best partition size (MB) and credit size (MB)

(partition, credit)	VGG16	ResNet50	Transformer
MXNet PS RDMA	(6, 21)	(3, 17)	(5, 29)
MXNet NCCL RDMA	(88, 171)	(56, 64)	(56, 103)

(*i.e.*, 1Gbps, 10Gbps, 25Gbps, 40Gbps, and 100Gbps). In this experiment, we use 4 machines with 32 GPUs in total to run the workers. We set the available bandwidth by limiting the speed of Mellanox NIC card using the mlnx_qos command. We measure the training speed of the benchmarks in cases of MXNet PS RDMA and MXNet NCCL RDMA. Figure 13 shows the training speeds of baseline and ByteScheduler with auto-tuning ("Tuned Scheduler" bars).

From Figure 13, we observe that: (1) ByteScheduler with auto-tuning consistently speeds up the training of VGG16 and Transformer in all bandwidth settings. The improvement is larger with PS, which is also consistent with previous figures. (2) The improvement for ResNet50 is high when the bandwidth is less than 25Gbps (*e.g.*, 64.4% speedup with PS under 10Gbps bandwidth) but decreases as the bandwidth increases (*e.g.*, 9.6% speedup with PS under 100Gbps bandwidth). This is because ResNet50 is more computation-intensive and has less parameters than the other two models.

6.3 Auto-tuning's Contribution and Overhead

Figure 13 also shows the training speed when disabling auto-tuning (*i.e.*, the "Fixed Scheduler" bars). Specifically, we fix the partition and credit sizes to be values given by our auto-tuning algorithm under *1Gbps* bandwidth, and use them under all bandwidth settings. We find that auto-tuning achieves higher speed than without it in all cases. Auto-tuning is also necessary as in some cases, without auto-tuning, ByteScheduler even performs worse than baseline.

In addition, we show the best partition size and credit size of the 3 models using 32 GPUs for training under 100Gbps bandwidth in Table 1. We see that: (1) The best configurations are different. (2) NCCL requires much larger partition size and credit size than PS, mainly due to the larger synchronization costs of all-reduce primitive. (3) The best partition and credit sizes also differ between different models. For example, ResNet50 is computation-heavy, so achieving timely preemption is more important than paying less partition overhead. VGG16 is the opposite.

Search cost. To compare with BO used in our auto-tuning algorithms, we choose 3 classic alternatives, *i.e.*, random search, grid search and SGD with momentum [30]. Random search tests points randomly picked. Grid search splits each dimension by interval in search space and tests grid points one by one. For SGD with momentum [30], we randomly choose an initial point and restart the algorithm in case that it is stuck in a local optimum. We test different hyper-parameters and show the results achieved with the best parameters. For each algorithm, we stop searching when it reaches the optimal configuration (as identified by grid search). We compare the search cost (*i.e.*, number of searches) of algorithms for VGG16 and Transformer in PS and NCCL.

Figure 14 shows that: (1) on average Bayesian Optimization can reach optimal configuration with much less cost than other algorithms, *e.g.*, it incurs 28% − 51% less number of trials than momentum and hence mitigates the restarting overhead (see §5) in PS architecture. (2) BO is much more stable than random search and SGD with momentum (smaller standard deviation), as it selects next point based on expected improvement. So users can expect more stable performance in the tuning phase.

BO tuning runs in parallel with model training. For Horovod NCCL, it does not add extra overhead. For PS architecture like TensorFlow PS, the restarting overhead varies among models, *e.g.*, 5 seconds for VGG16 and 9 seconds for ResNet50 in each restart. Since BO typically runs within 10 trials (Figure 14), the overhead is small compared to the entire training.

7 Discussion and Future Directions

Dynamic partition size and credit size. We use auto-tuning to search for the best parameters, *i.e.*, partition size and credit size, at the beginning of training, and assume their values stay constant throughout the training. We may further allow dynamic partition size and credit size over the training course, by consistently searching for the best values using newly profiled results. Also, we may use different partition and credit sizes for different layers in the DNN. Both improvements may incur significantly more search costs. We leave efficient search algorithms as open problems.

Co-scheduling in a shared cluster. For simplicity, our scheduling algorithm omits the consideration of possible shared resources between training jobs: (a) shared worker resources (*e.g.*, CPUs and NICs); (b) shared network with congestion; (c) shared PS resources (*e.g.*, GPU/CPU). However, in the multi-tenancy cloud setting, resource sharing is common. The performance impact is not negligible when the shared resource is the bottleneck. A potential solution is to cooperatively schedule the communication within the shared cluster, and possibly integrate it with DNN cluster job scheduler [17, 29] or Coflow scheduler [13].

ByteScheduler for other ML frameworks. Our current implementation supports TensorFlow, PyTorch and MXNet. Though they are probably the three most popular frameworks, there are many other alternatives, like Caffe [22], CNTK [31] and Spark [27, 35]. We believe that we can apply ByteScheduler to them in similar ways, since the ByteScheduler design is quite generic. We leave this as future work.

8 Related work

Speed up communication in DNN training. Existing approaches include: (1) speeding up individual messages by using RDMA [25] or NCCL [4]; (2) compressing data transmission such as gradient quantization [9, 37] and sparse parameter synchronization [7]; (3) optimizing communication approach, *e.g.*, Stale Synchronous Parallel [20] for PS and different all-reduce algorithms [10, 14]; (4) minimizing network flow completion time by using flow control [26] or Coflow scheduling [13]. These work mainly focus on accelerating a single communication operation, and are orthogonal and complementary to ByteScheduler.

Overlap communication with computation. Most DNN frameworks, *e.g.*, TensorFlow, PyTorch, MXNet and Poseidon [39], support overlapping communication with backward propagation. On top of this, P3 [21] further attempts to overlap communication with forward propagation by layer partitioning and scheduling on MXNet PS architecture. TicTac [18] proposes a similar idea but shows a much smaller training speedup (less than 20%) based on TensorFlow PS TCP. We suspect that it is due to the global barrier (§2.3).

P3 and TicTac are probably the closest related work to ByteScheduler. However, ByteScheduler is generic across multiple DNN frameworks, provides analysis on the scheduling algorithm, overcomes their shortcomings like not adapting to system overhead and suffering from a global barrier, and achieves much higher speedup.

Parameter tuning with BO. BO is also used in searching the best cloud configuration for big data analytics [8], tuning the learning parameters of machine learning algorithm [34], and online parameter tuning of databases [16]. ByteScheduler is parallel to them, as it searches the optimal scheduling parameters to accelerate distributed DNN training.

9 Conclusion

ByteScheduler is a generic communication scheduler for distributed DNN training acceleration. Our implementation supports multiple ML frameworks including MXNet, PyTorch, and TensorFlow, with both PS and all-reduce for gradient synchronization, and using either TCP or RDMA. ByteScheduler achieves up to 196% end-to-end speedup. The key design points include a unified abstraction for communication operations, *Dependency Proxy*, and system parameter auto-tuning. We have open-sourced our implementation and expect that the community will add support to more existing and future frameworks.

References

[1] 2019. ByteScheduler Appendix. https://www.dropbox.com/s/smoq6xd6pr7av81/bytescheduler_appendix.pdf?dl=0.

[2] 2019. ByteScheduler Source Code. https://github.com/bytedance/byteps.

[3] 2019. MLPerf Training v0.6 Results. https://mlperf.org/training-results-0-6/.

[4] 2019. NVIDIA Collective Communications Library (NCCL). https://developer.nvidia.com/nccl.

[5] 2019. TensorFlow Grapper. https://github.com/tensorflow/tensorflow/tree/master/tensorflow/core/grappler.

[6] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. TensorFlow: A System for Large-Scale Machine Learning. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

[7] Alham Fikri Aji and Kenneth Heafield. 2017. Sparse Communication for Distributed Gradient Descent. *arXiv preprint arXiv:1704.05021* (2017).

[8] Omid Alipourfard, Hongqiang Harry Liu, Jianshu Chen, Shivaram Venkataraman, Minlan Yu, and Ming Zhang. 2017. Cherrypick: Adaptively Unearthing the Best Cloud Configurations for Big Data Analytics. In *Proceedings of USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.

[9] Dan Alistarh, Demjan Grubic, Jerry Li, Ryota Tomioka, and Milan Vojnovic. 2017. QSGD: Communication-Efficient SGD via Gradient Quantization and Encoding. In *Proceedings of Advances in Neural Information Processing Systems (NIPS)*.

[10] Ammar Ahmad Awan, Ching-Hsiang Chu, Hari Subramoni, and Dhabaleswar K Panda. 2018. Optimized Broadcast for Deep Learning Workloads on Dense-GPU Infiniband Clusters: MPI or NCCL?. In *Proceedings of the 25th European MPI Users' Group Meeting*.

[11] Eric Brochu, Vlad M Cora, and Nando De Freitas. 2010. A Tutorial on Bayesian Optimization of Expensive Cost Functions, with Application to Active User Modeling and Hierarchical Reinforcement Learning. *arXiv preprint arXiv:1012.2599* (2010).

[12] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. 2016. MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems. In *Proceedings of NIPS Workshop on Machine Learning Systems*.

[13] Mosharaf Chowdhury, Yuan Zhong, and Ion Stoica. 2014. Efficient Coflow Scheduling with Varys. In *Proceedings of ACM Special Interest Group on Data Communication (SIGCOMM)*.

[14] Jeff Daily, Abhinav Vishnu, Charles Siegel, Thomas Warfel, and Vinay Amatya. 2018. GossipGraD: Scalable Deep Learning using Gossip Communication based Asynchronous Gradient Descent. *arXiv preprint arXiv:1803.05880* (2018).

[15] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Andrew Senior, Paul Tucker, Ke Yang, Quoc V Le, et al. 2012. Large Scale Distributed Deep Networks. In *Proceedings of Advances in Neural Information Processing Systems (NIPS)*.

[16] Songyun Duan, Vamsidhar Thummala, and Shivnath Babu. 2009. Tuning Database Configuration Parameters with iTuned. In *Proceedings of Very Large Data Bases (VLDB) Endowment*.

[17] Juncheng Gu, Mosharaf Chowdhury, Kang G Shin, Yibo Zhu, Myeongjae Jeon, Junjie Qian, Hongqiang Liu, and Chuanxiong Guo. 2019. Tiresias: A GPU Cluster Manager for Distributed Deep Learning. In *Proceedings of USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.

[18] Sayed Hadi Hashemi, Sangeetha Abdu Jyothi, and Roy H Campbell. 2019. TicTac: Accelerating Distributed Deep Learning with Communication Scheduling. In *Proceedings of Systems and Machine Learning (SysML)*.

[19] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

[20] Qirong Ho, James Cipar, Henggang Cui, Seunghak Lee, Jin Kyu Kim, Phillip B Gibbons, Garth A Gibson, Greg Ganger, and Eric P Xing. 2013. More Effective Distributed ML via a Stale Synchronous Parallel Parameter Server. In *Proceedings of Advances in Neural Information Processing Systems (NIPS)*.

[21] Anand Jayarajan, Jinliang Wei, Garth Gibson, Alexandra Fedorova, and Gennady Pekhimenko. 2019. Priority-Based Parameter Propagation for Distributed DNN Training. In *Proceedings of Systems and Machine Learning (SysML)*.

[22] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. 2014. Caffe: Convolutional Architecture for Fast Feature Embedding. In *Proceedings of the 22nd ACM International Conference on Multimedia*.

[23] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. ImageNet Classification with Deep Convolutional Neural Networks. In *Proceedings of Advances in Neural Information Processing Systems (NIPS)*.

[24] Mu Li, David G Andersen, Jun Woo Park, Alexander J Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J Shekita, and Bor-Yiing Su. 2014. Scaling Distributed Machine Learning with the Parameter Server. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

[25] Jiuxing Liu, Jiesheng Wu, and Dhabaleswar K Panda. 2004. High Performance RDMA-based MPI Implementation over InfiniBand. *International Journal of Parallel Programming* (2004).

[26] Luo Mai, Chuntao Hong, and Paolo Costa. 2015. Optimizing Network Performance in Distributed Machine Learning. In *Proceedings of USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*.

[27] Xiangrui Meng, Joseph Bradley, Burak Yavuz, Evan Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, DB Tsai, Manish Amde, Sean Owen, et al. 2016. Mllib: Machine Learning in Apache Spark. *Journal of Machine Learning Research* (2016).

[28] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. 2017. Automatic Differentiation in PyTorch. In *Proceedings of NIPS Autodiff Workshop*.

[29] Yanghua Peng, Yixin Bao, Yangrui Chen, Chuan Wu, and Chuanxiong Guo. 2018. Optimus: An Efficient Dynamic Resource Scheduler for Deep Learning Clusters. In *Proceedings of the 13th ACM European Conference on Computer Systems (EuroSys)*.

[30] Sebastian Ruder. 2016. An Overview of Gradient Descent Optimization Algorithms. *arXiv preprint arXiv:1609.04747* (2016).

[31] Frank Seide and Amit Agarwal. 2016. CNTK: Microsoft's Open-Source Deep-Learning Toolkit. In *Proceedings of ACM International Conference on Knowledge Discovery and Data Mining (KDD)*.

[32] Alexander Sergeev and Mike Del Balso. 2018. Horovod: Fast and Easy Distributed Deep Learning in TensorFlow. *arXiv preprint arXiv:1802.05799* (2018).

[33] Karen Simonyan and Andrew Zisserman. 2014. Very Deep Convolutional Networks for Large-Scale Image Recognition. *arXiv preprint arXiv:1409.1556* (2014).

[34] Jasper Snoek, Hugo Larochelle, and Ryan P Adams. 2012. Practical Bayesian Optimization of Machine Learning Algorithms. In *Proceedings of Advances in Neural Information Processing Systems (NIPS)*.

[35] Evan R Sparks, Ameet Talwalkar, Virginia Smith, Jey Kottalam, Xinghao Pan, Joseph Gonzalez, Michael J Franklin, Michael I Jordan, and Tim Kraska. 2013. MLI: An API for Distributed Machine Learning. In *Proceedings of IEEE International Conference on Data Mining (ICDM)*.

[36] Minjie Wang, Chien-chin Huang, and Jinyang Li. 2019. Supporting Very Large Models Using Automatic Dataflow Graph Partitioning. In *Proceedings of the 14th ACM European Conference on Computer Systems (EuroSys)*.

[37] Wei Wen, Cong Xu, Feng Yan, Chunpeng Wu, Yandan Wang, Yiran Chen, and Hai Li. 2017. Terngrad: Ternary Gradients to Reduce Communication in Distributed Deep Learning. In *Proceedings of Advances in Neural Information Processing Systems (NIPS)*.

[38] Wikipedia. 2019. Monkey Patch. https://en.wikipedia.org/wiki/Monkey_patch.

[39] Hao Zhang, Zeyu Zheng, Shizhen Xu, Wei Dai, Qirong Ho, Xiaodan Liang, Zhiting Hu, Jinliang Wei, Pengtao Xie, and Eric P Xing. 2017. Poseidon: An Efficient Communication Architecture for Distributed Deep Learning on GPU Clusters. In *Proceedings of USENIX Annual Technical Conference (USENIX ATC)*.