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

The University of Hong Kong ByteDance Inc.

DNN Training

DNN training is compute-hungry and time-consuming

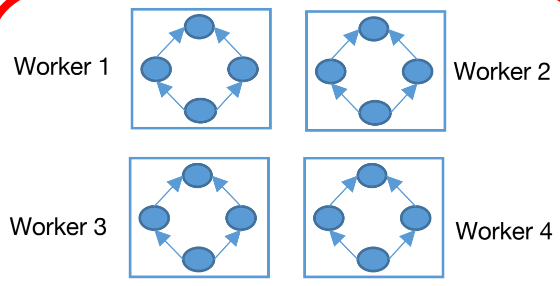
ResNet50	Training Time
1 TPUv3	10 hours
1024 TPUv3	1.28 minutes

https://mlperf.org/training-results-0-6

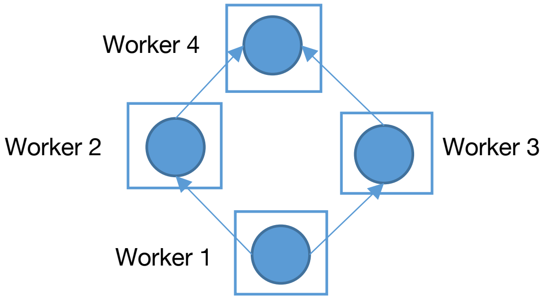
BERT	Training Time
16 TPUv3	81 hours
1024 TPUv3	76.19 minutes

https://arxiv.org/pdf/1904.00962.pdf

Training can be scaled out by data parallelism or model parallelism

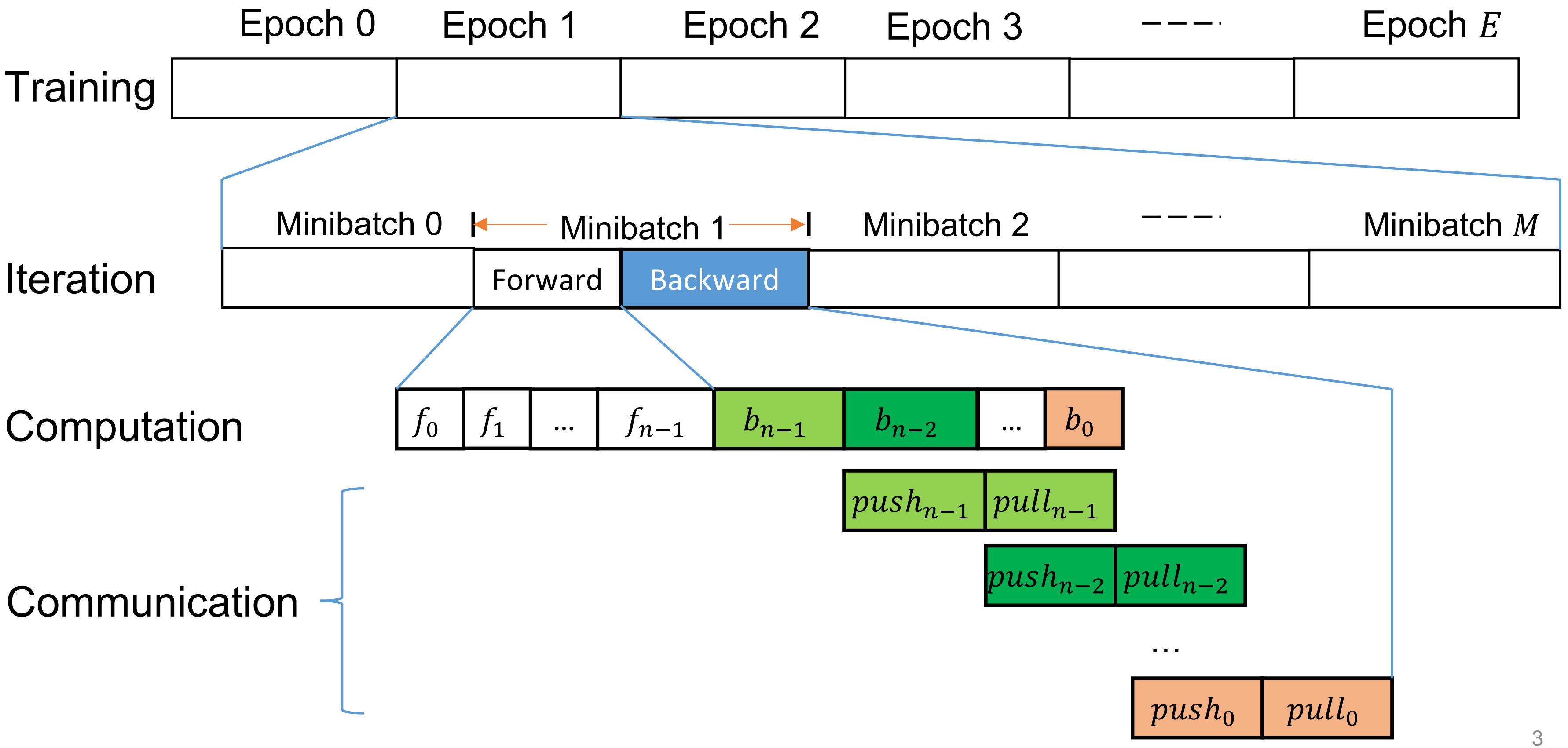


data parallelism

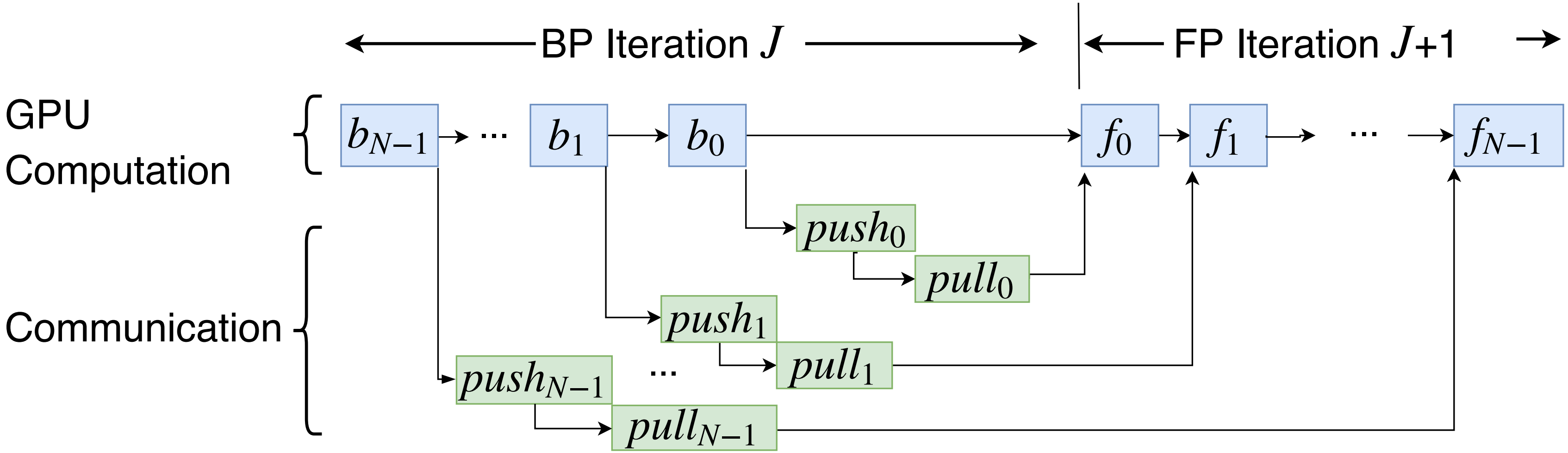


model parallelism

Data Parallel DNN Training



PS Dependency Graph

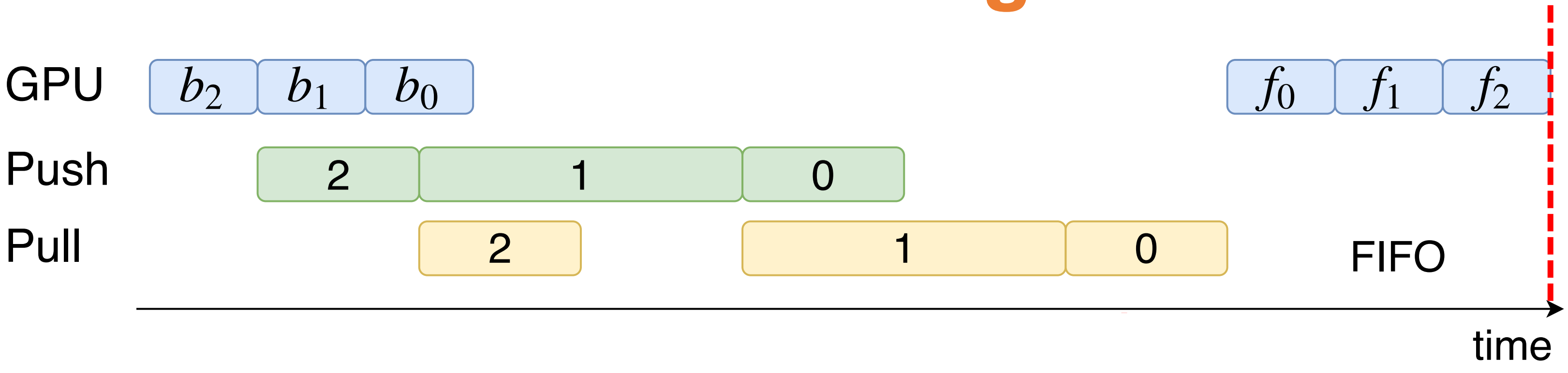


Dependency:

- Backward depends on forward
- Push depends on backward
- Pull depends on push
- Forward depends on pull

Framework engines execute the graph according to the dependencies

Communication Scheduling

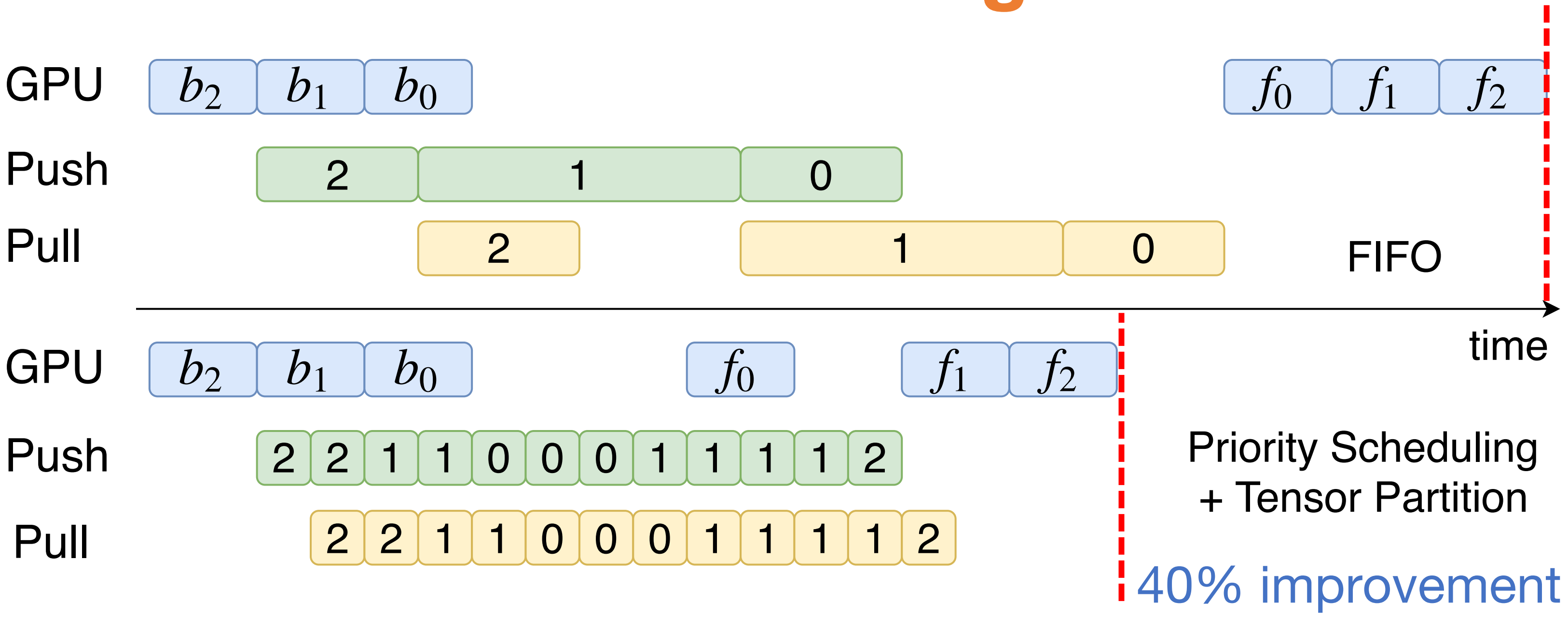


40% improvement

Problem: FIFO strategy does not overlap communication with computation well

P3, TicTac: partition tensors and change tensor transmission order

Communication Scheduling



Problem: FIFO strategy does not overlap communication with computation well

P3, TicTac: partition tensors and change tensor transmission order

Limitations of Existing Work

P3 and TicTac:

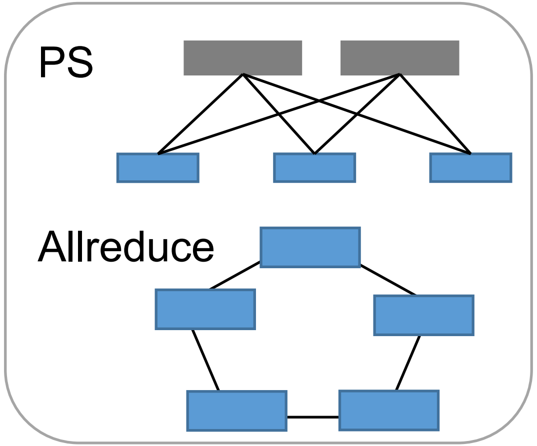
- Coupled with specific framework implementations, e.g., P3 for MXNet PS and TicTac for TensorFlow PS.
- Heuristic scheduling with empirical results

Many different setups in distributed DNN training:

TensorFlow mxnet

PYTORCH

ML frameworks



Communication architectures

TCP

RDMA

Network protocols

Limitations of Existing Work

P3 and TicTac:

- Coupled with specific framework implementations, e.g., P3 for MXNet PS and TicTac for TensorFlow PS.
- Heuristic scheduling with empirical results

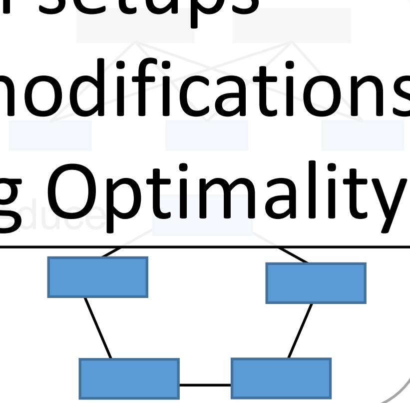
Many different setups in distributed DNN training:

How to do communication scheduling:

1. Work in all setups
2. Minimal modifications
3. Scheduling Optimality

PYTORCH

ML frameworks



Communication architectures

TCP

RDMA

Network protocols

One Unified Scheduling System for ALL

Observation: The dependency graph structure is intrinsic for DNN training, regardless of training frameworks, communication architectures, or network protocols

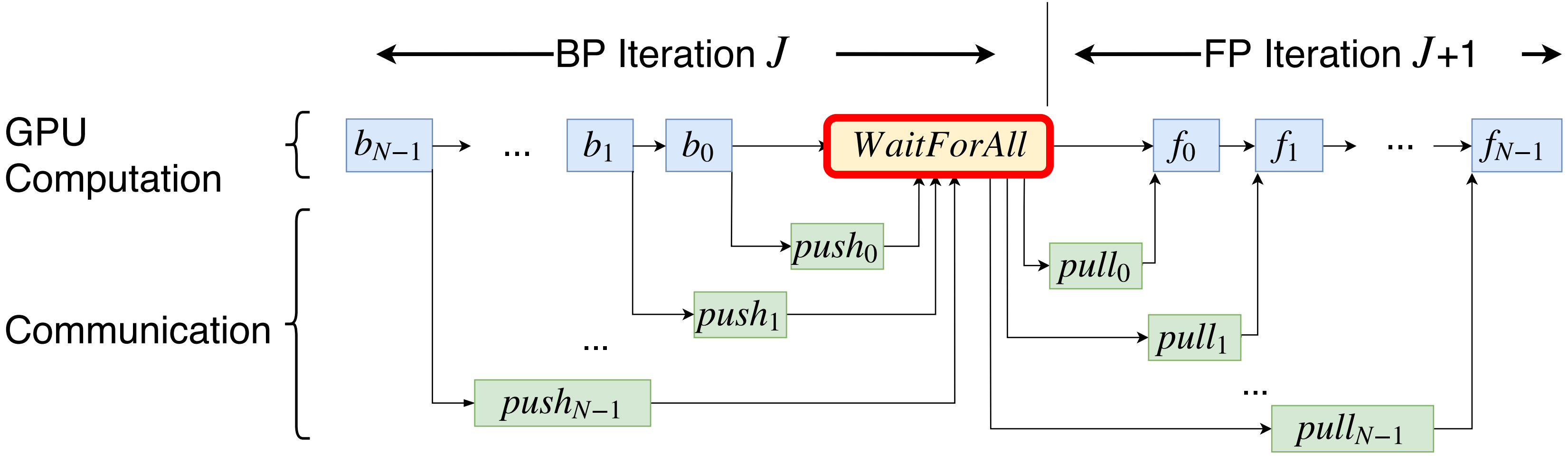
ByteScheduler: A generic tensor scheduling framework

- One unified scheduler framework that abstracts tensor scheduling from various frameworks, communication architectures and network protocols

- One principled scheduling algorithm that is guided by theory and works in real-world

Challenge 1: Different ML Frameworks

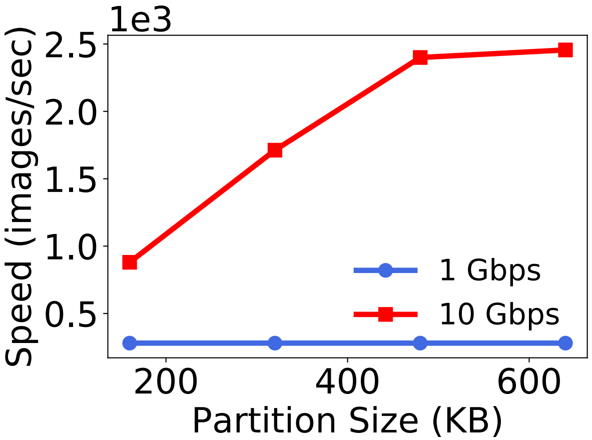
- Imperative framework (e.g., PyTorch) and declarative framework (e.g., TensorFlow)
- Global barrier between iterations (e.g., TensorFlow, PyTorch), causing any scheduling of push/all-reduce ineffective

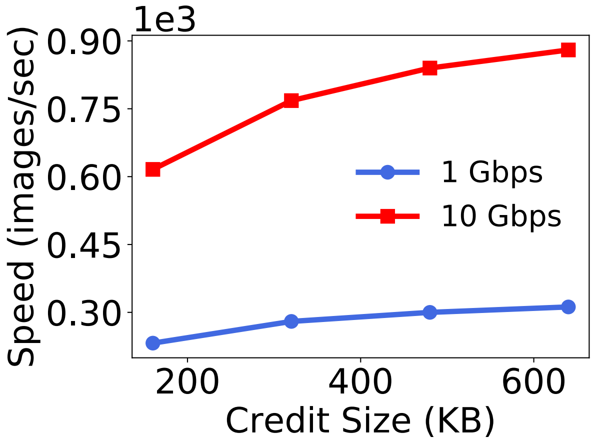


Challenge 2: Different Runtime Environments

The overhead of scheduling & tensor partitioning is different for different system setups and network conditions

How to balance the performance gain with scheduling overhead? The system parameters (e.g., partition size) are likely to be affected by different runtime configurations, e.g., bandwidths, DNNs





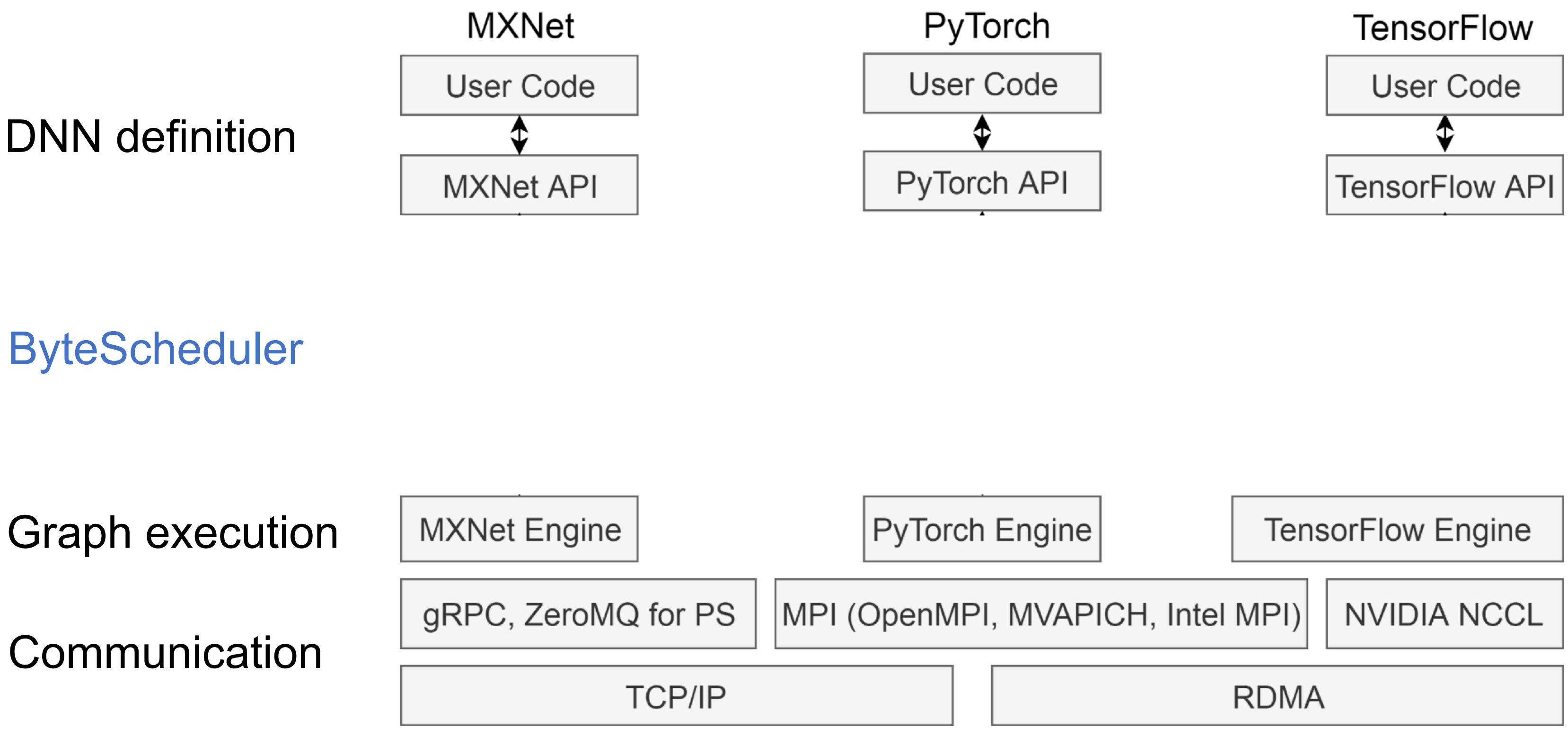
Outline

1. Background and Motivation

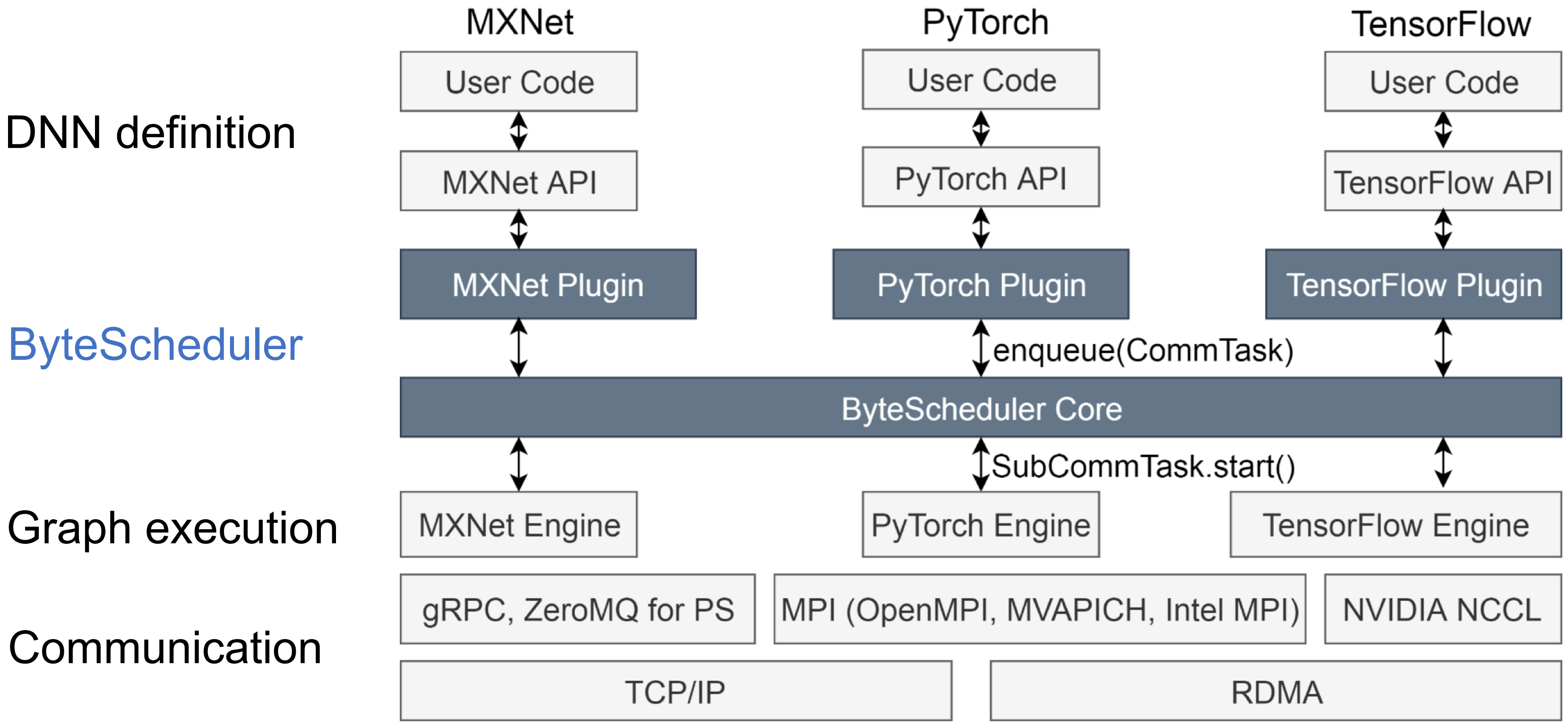
2. ByteScheduler Design

3. Evaluation

Unified Scheduler Across Frameworks



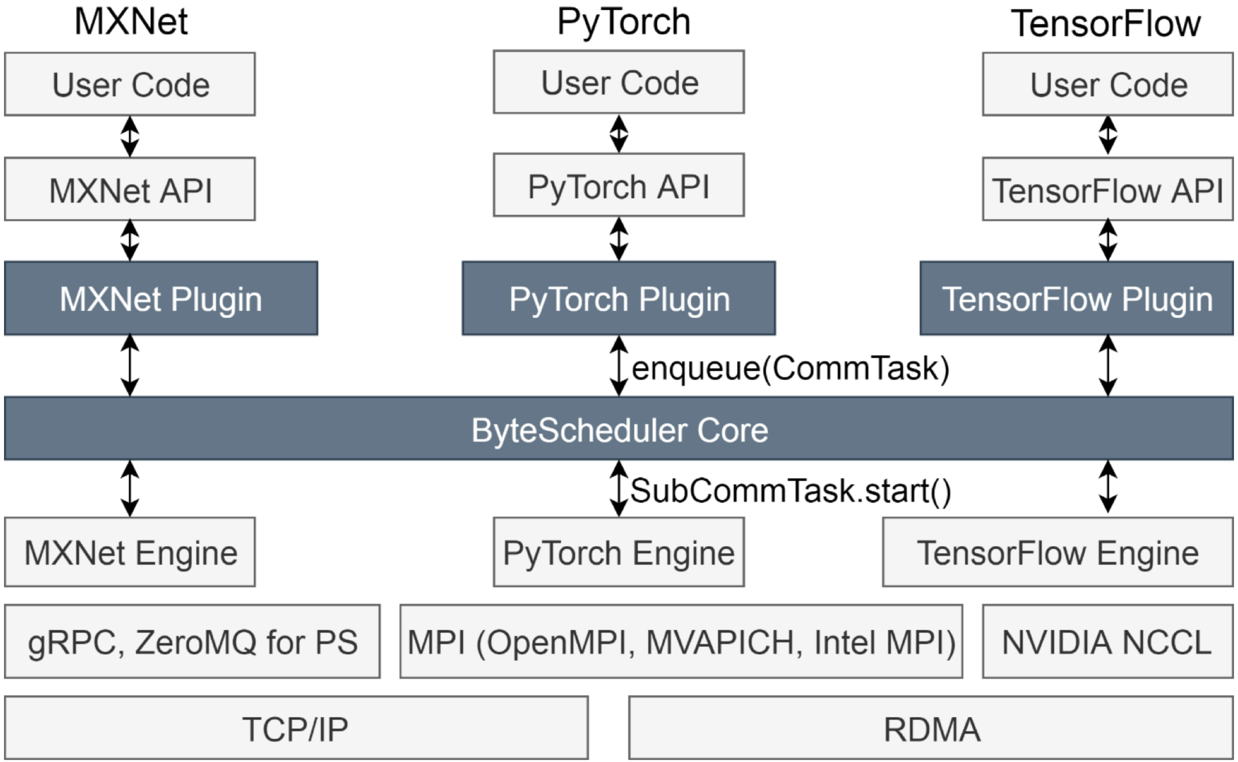
Unified Scheduler Across Frameworks



ByteScheduler Architecture

Plugin: Wrap each communication operation as a CommTask

Core: Partition and schedule CommTasks



CommTask: A Unified Abstraction

CommTask: A wrapped communication operation, e.g., push one tensor, all-reduce one tensor

CommTask APIs implemented in framework plugins:

- partition(size): partition a CommTask into SubCommTasks with tensors no larger than a threshold *size*
- notify_ready(): notify Core about the readiness of a CommTask
- start(): start a CommTask by calling the underlying push/pull/all-reduce
- notify_finish(): notify Core about the completion of a CommTask

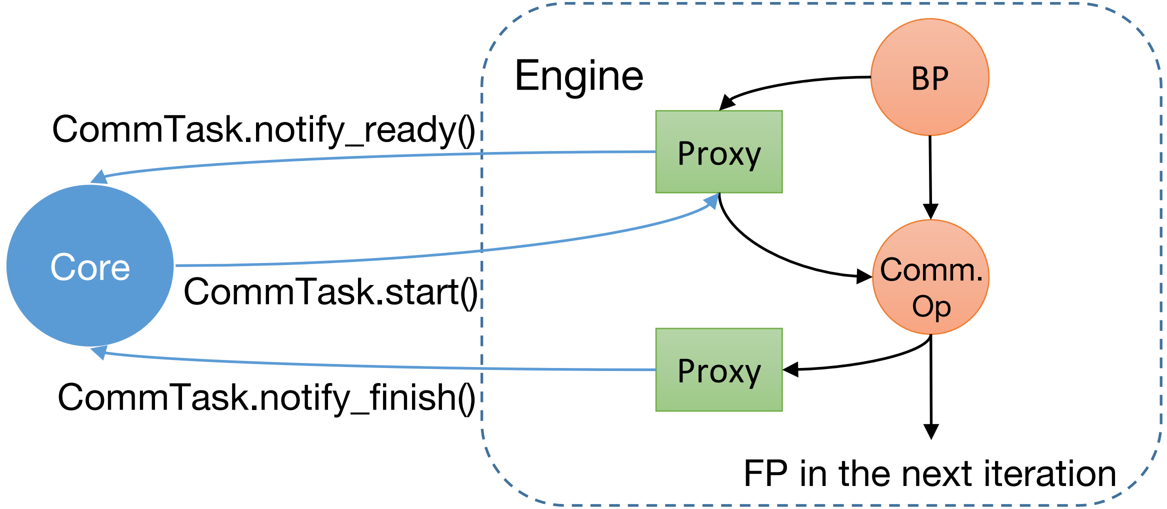
Dependency Proxy: Get the Scheduling Control

A Dependency Proxy is an operator to get the scheduling control from the frameworks to the Core

Implementation differs for imperative and declarative engines

Dependency Proxy:

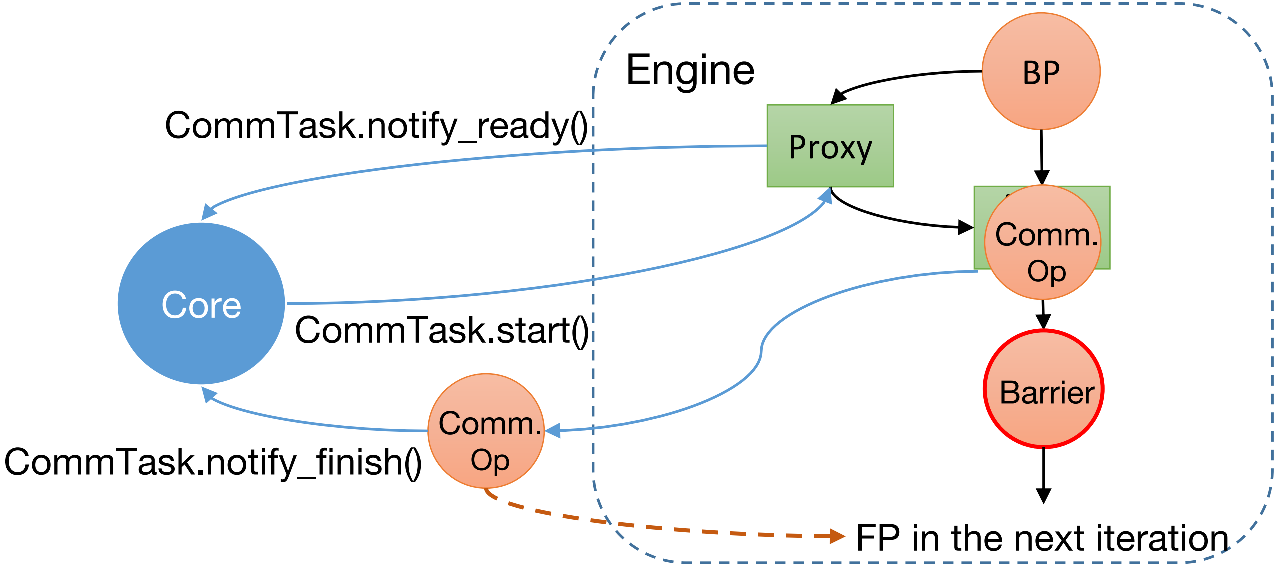
- Trigger CommTask.notify_ready() via a callback
- Wait to finish until Core calls CommTask.start()
- Generate completion signal using CommTask.notify_finish()



Dependency Proxy: Crossing the Global Barrier

Out-of-engine communication: Start the actual communication outside engine

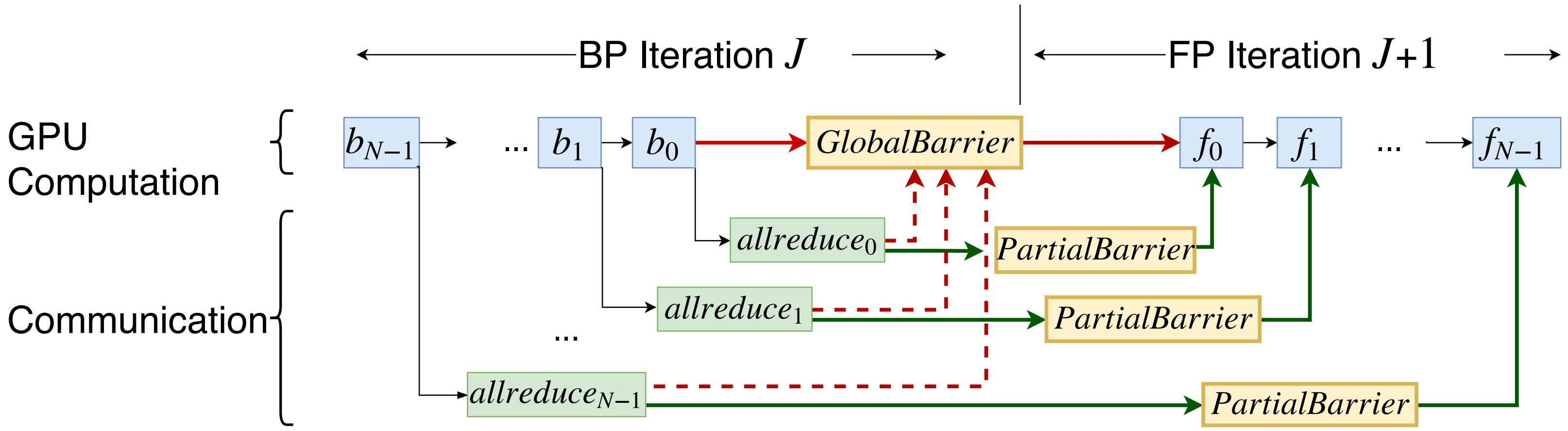
Layer-wise out-of-engine dependencies: Build correct dependency for each layer by adding a Proxy to block forward computation



Dependency Proxy: Crossing the Global Barrier

Out-of-engine communication: Start the actual communication outside engine

Layer-wise out-of-engine dependencies: Build correct dependency for each layer by adding a Proxy to block forward computation



Optimal Scheduling Theorem

Optimal scheduling for minimizing the time for each iteration:

- For PS, prioritize $push_i$ over $push_j$, and $pull_i$ over $pull_j$, $\forall i < j$
- For all-reduce, prioritize $allreduce_i$ over $allreduce_j$, $\forall i < j$
- Assuming infinitely small partition size and immediate preemption without overhead

In practice, partitioning and preemption have overhead

Credit-based Preemption

Stop-and-wait approach in previous work can not fully utilize network bandwidth

- Send a single tensor and wait for its ACK

Credit-based Preemption

- Work like a sliding window and the credit is the window size
- Allow multiple tensors in a sliding window to be sent concurrently

Credit size is an important system parameter

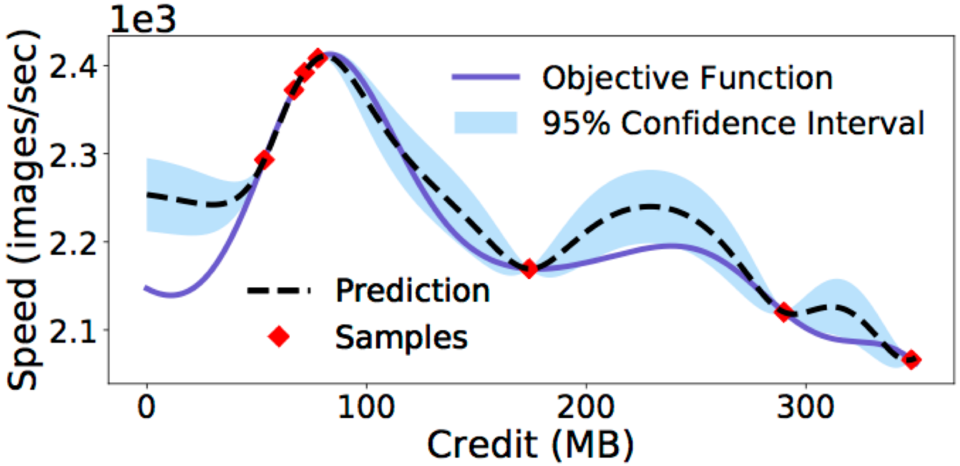
- Pro: higher bandwidth utilization
- Con: less timely preemption due to FIFO communication stack

Auto-tuning Partition Size and Credit Size

Optimal partition size and credit size are affected by many factors, e.g., network bandwidths, number of workers, DNN models, CPU and GPU types

We use Bayesian Optimization for auto-tuning

- Work with general objective function
- Minimize the overhead, i.e., the number of sampled points



Outline

1. Background and Motivation

2. ByteScheduler Design

3. Evaluation

Evaluation

Implementation: MXNet PS and all-reduce (based on Horovod), PyTorch (based on Horovod), TensorFlow PS

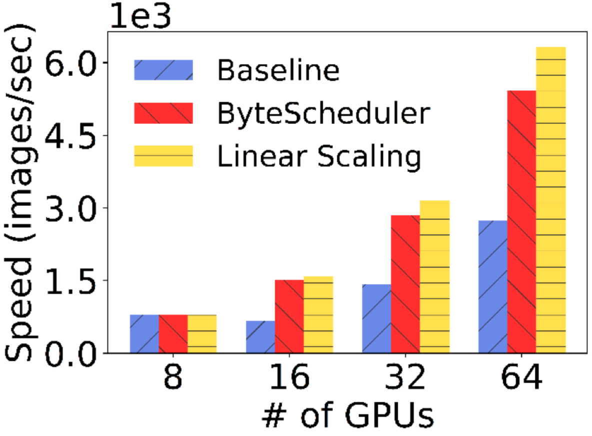
```python
# After user created an MXNet KVStore object kvs
from bytescheduler.mxnet.kvstore import ScheduledKVS
kvs = ScheduledKVS(kvs)
# Continue using kvs without any further modification
```

Testbed: 16 machines, each with 8 Tesla V100 GPUs and 100Gbps bandwidth

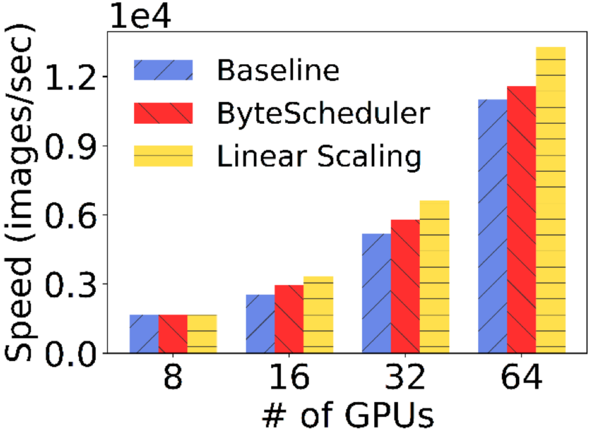
Comparison:

- Baseline: vanilla ML frameworks
- Linear scaling: vanilla training speed on 1 machine multiplied by the number of machines

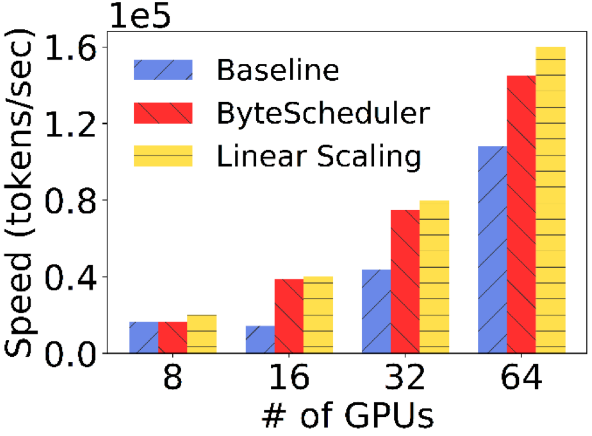
Scalability of ByteScheduler



VGG16 (97%-125%)



ResNet50 (9%-15%)

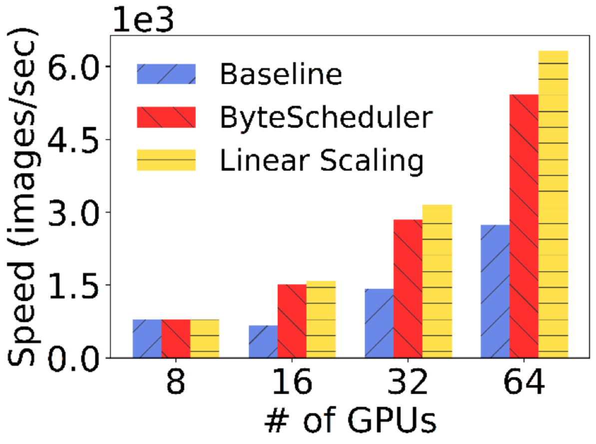


Transformer (70%-171%)

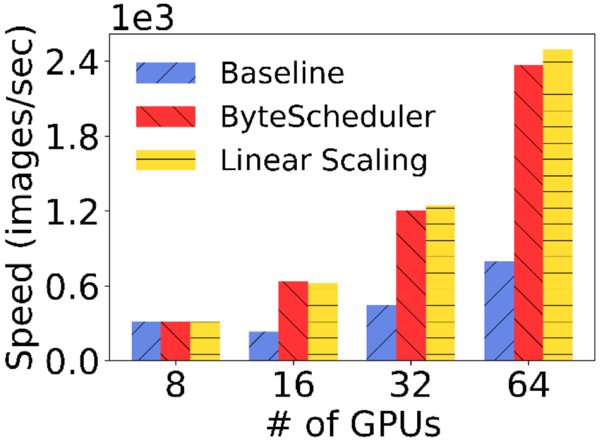
MXNet PS RDMA

- Up to 171% improvement and close to linear scaling

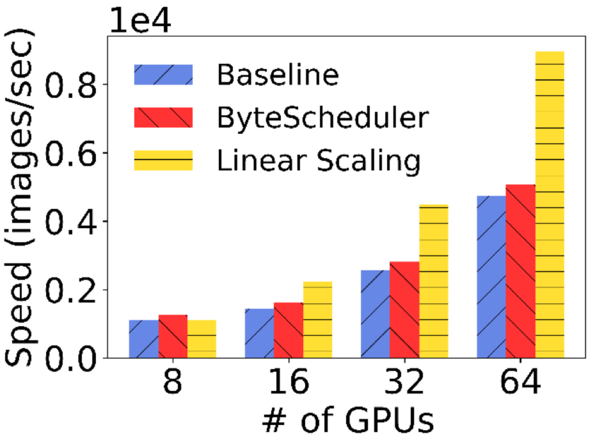
ByteScheduler for Multiple Frameworks



MXNet PS RDMA
(97%-125%)



TensorFlow PS RDMA
(170%-196%)

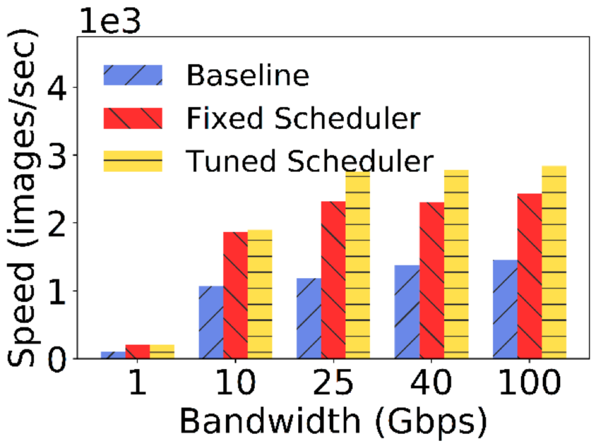


PyTorch NCCL TCP
(7%-13%)

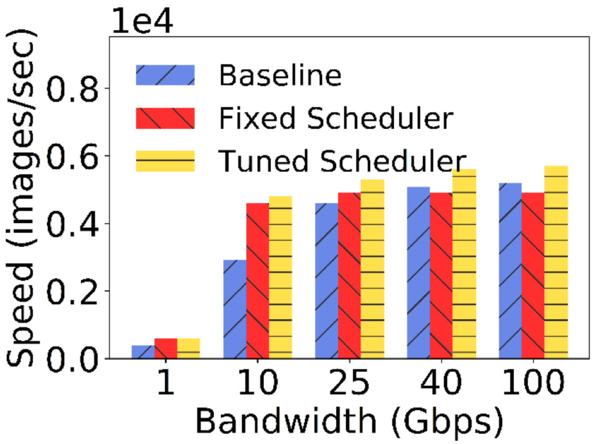
VGG16

- Up to 196% improvement compared to the baseline

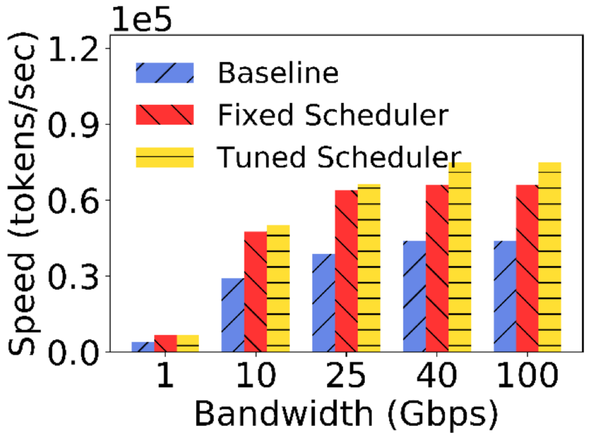
ByteScheduler Adapts to Different Bandwidths



VGG16 (79%-132%)



ResNet50 (10%-64%)



Transformer (67%-70%)

MXNet PS RDMA

- Consistent speedup in all bandwidth settings
- Without auto-tuning, the training speed is lower

Conclusion

ByteScheduler: A generic communication scheduler for distributed DNN training acceleration

- Unified abstraction for tensor scheduling
- Multiple training framework support, with minimal code change to existing frameworks
- Principled tensor scheduling design with parameter autotuning

Q&A

Source code:

https://github.com/bytedance/byteps/tree/bytescheduler/bytescheduler