A Generic Communication Scheduler for Distributed DNN Training Acceleration

In cooperation with

Venue:

The 27th ACM Symposium on Operating Systems Principles, 2019

Authors:

Yanghua Peng
Yibo Zhu
Yangrui Chen
Yixin Bao
Bairen Yi
Chang Lan
Chuan Wu
Chuanxiong Guo

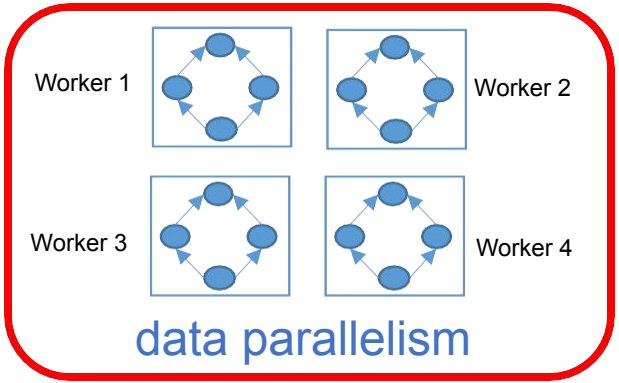
DNN Training

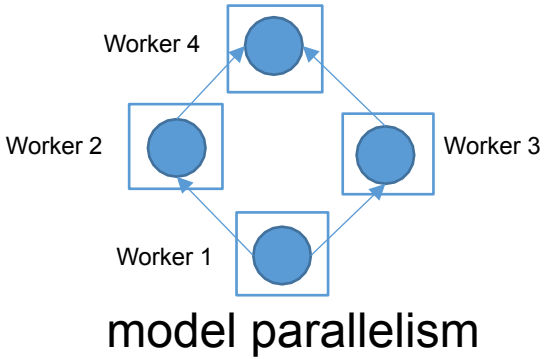
Training of DNNs is both compute-hungry and takes longer training time

ResNet50	Training Time	BERT	Training Time
1 TPUv3	10 hours	16 TPUv3	81 hours
1024 TPUv3	1.28 minutes	1024 TPUv3	76.19 minutes

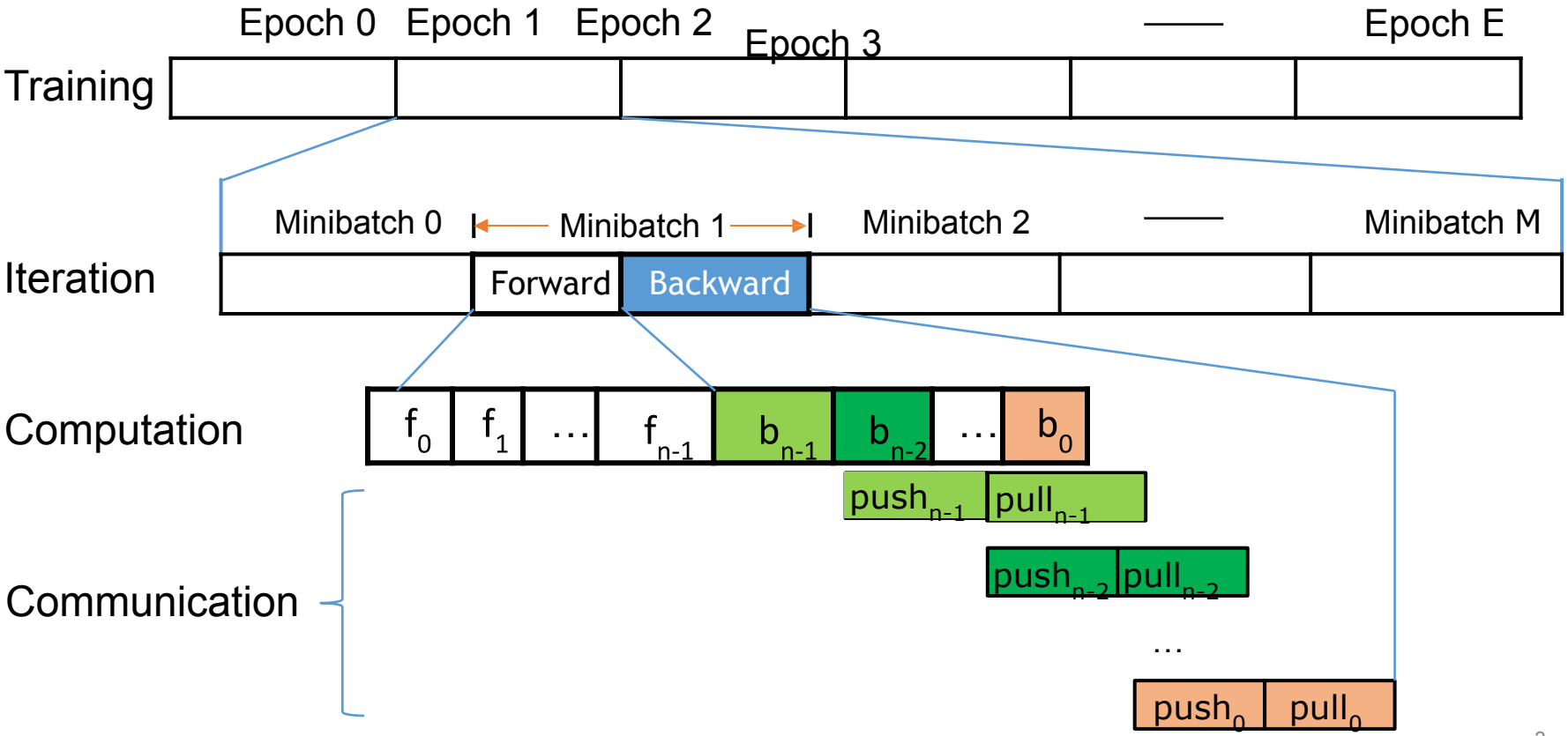
https://mlperf.org/training-results-0-6 https://arxiv.org/pdf/1904.00962.pdf

Scaling approaches to training of DNNs: data parallelism and model parallelism

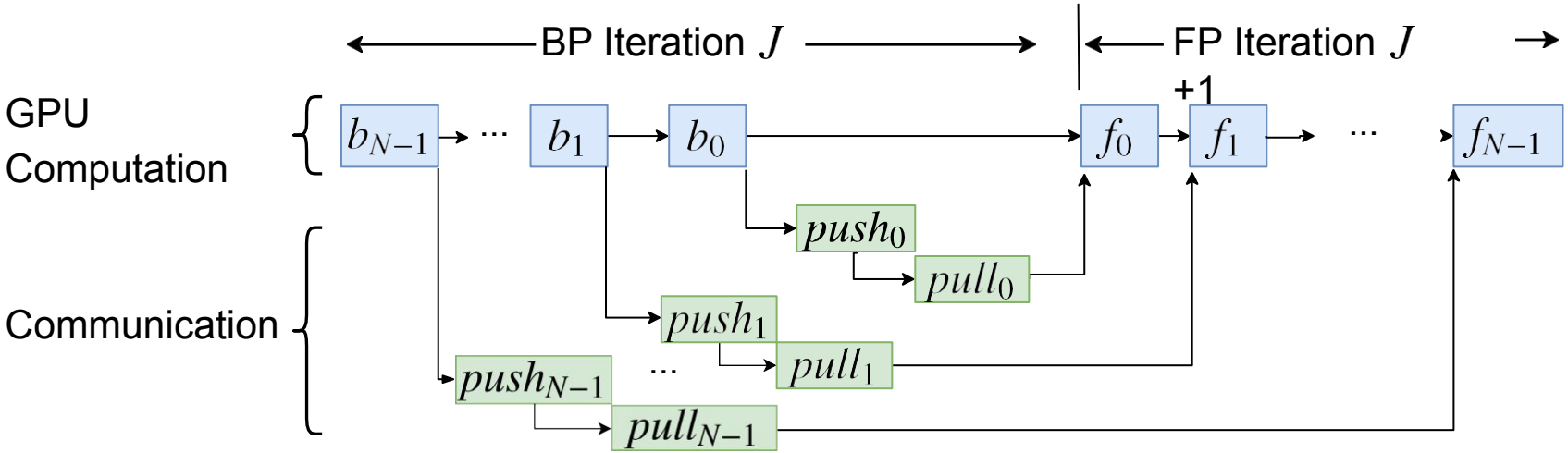




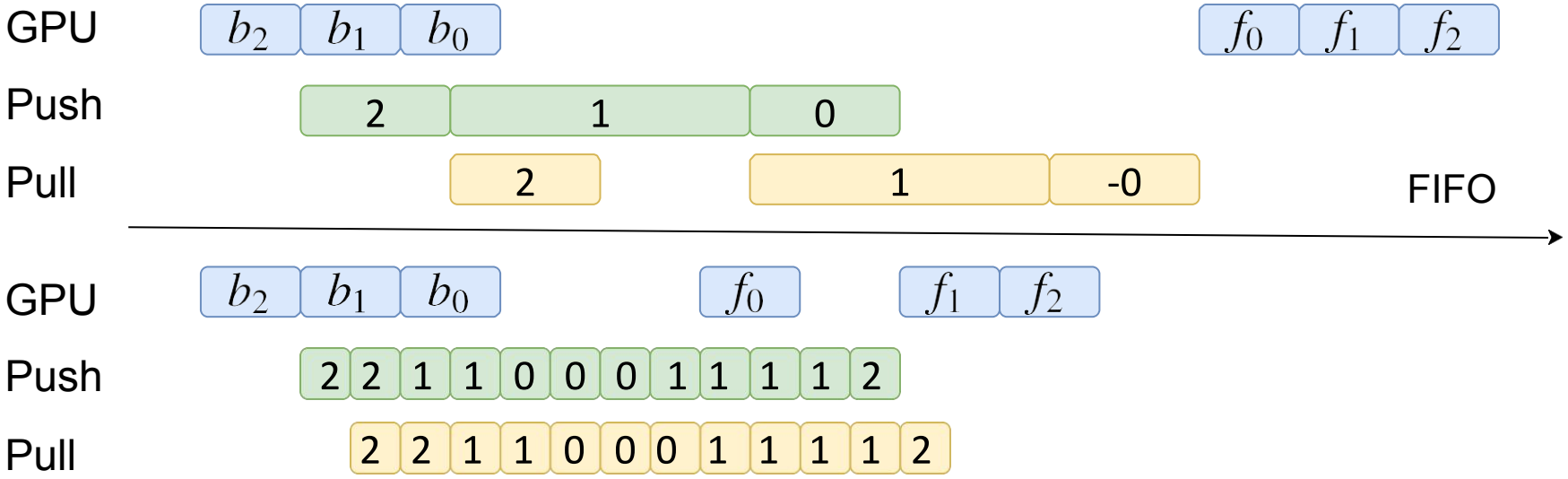
Data Parallel DNN Training



Dependency Graphs (Parameter-Server)



Communication Scheduling



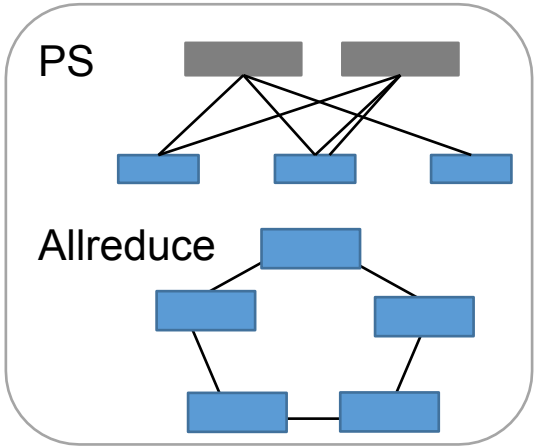
- Authors have independently found 40% speedup just by leveraging priority scheduling & tensor partitioning.

Limitations of Existing Work

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

Out-of-sync with the demands of the real-world settings:

ML frameworks



Communication architectures

Network protocols

Key questions motivating this research include:

- ❑ Proposed solution should work in all setups.
- ❑ Necessitates minimal modifications.
- ❑ Achieves elegant scheduling optimality.

Single Unified Scheduling

Key Finding:

Inherent relationship between the underlying dependency graph for DNN training.

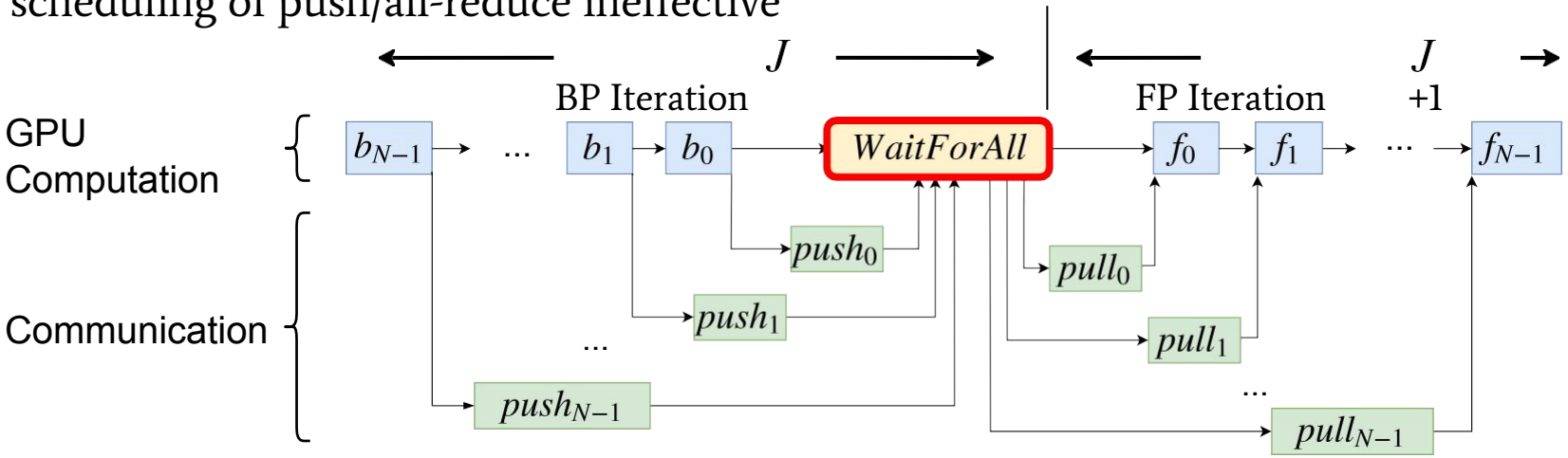
Solution:

ByteScheduler: A generic tensor scheduling framework

- One unified scheduler framework that abstracts tensor scheduling from various frameworks, communication architectures and network protocols
- One principled practical scheduling algorithm.
- Supports Tensorflow, Keras, Pytorch, and MXNet, and compatible with both TCP and RDMA networks.

Problem 1: Choice of Framework?

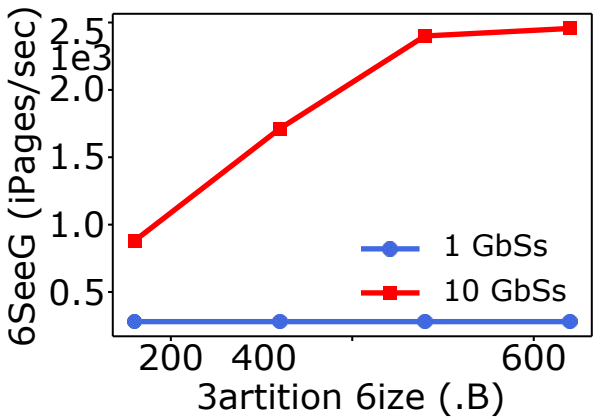
- Imperative framework (e.g., PyTorch) and declarative framework (e.g., TensorFlow)
- Global barrier between iterations (e.g., TensorFlow, PyTorch), causing any scheduling of push/all-reduce ineffective

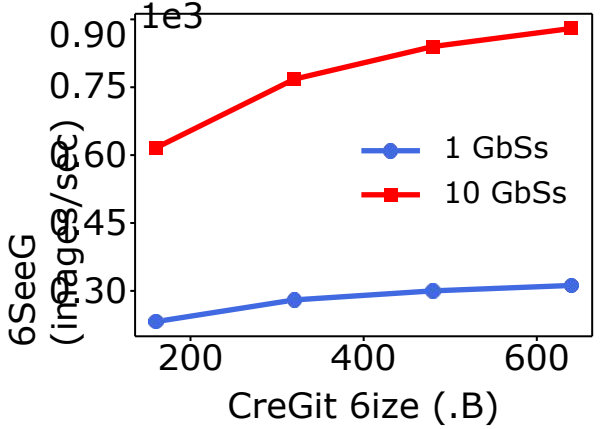


Problem 2: Choice of Runtime Environments

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

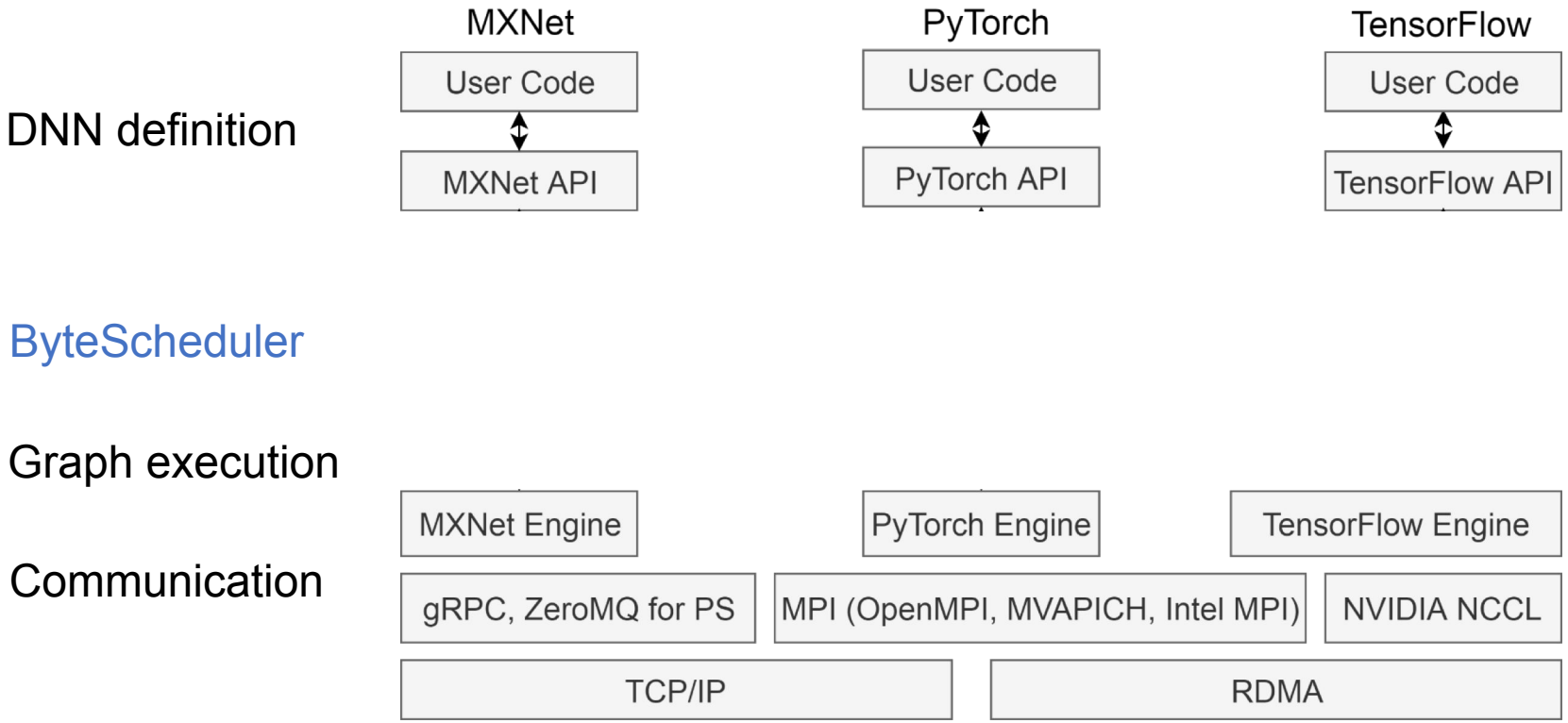
How to balance the performance gain with scheduling overhead? The system parameters (e.g., partition size) are likely to be affected by different runtime configurations, e.g., bandwidths, DNNs





Design of ByteScheduler

Frameworks-Agnostic Unified Scheduler



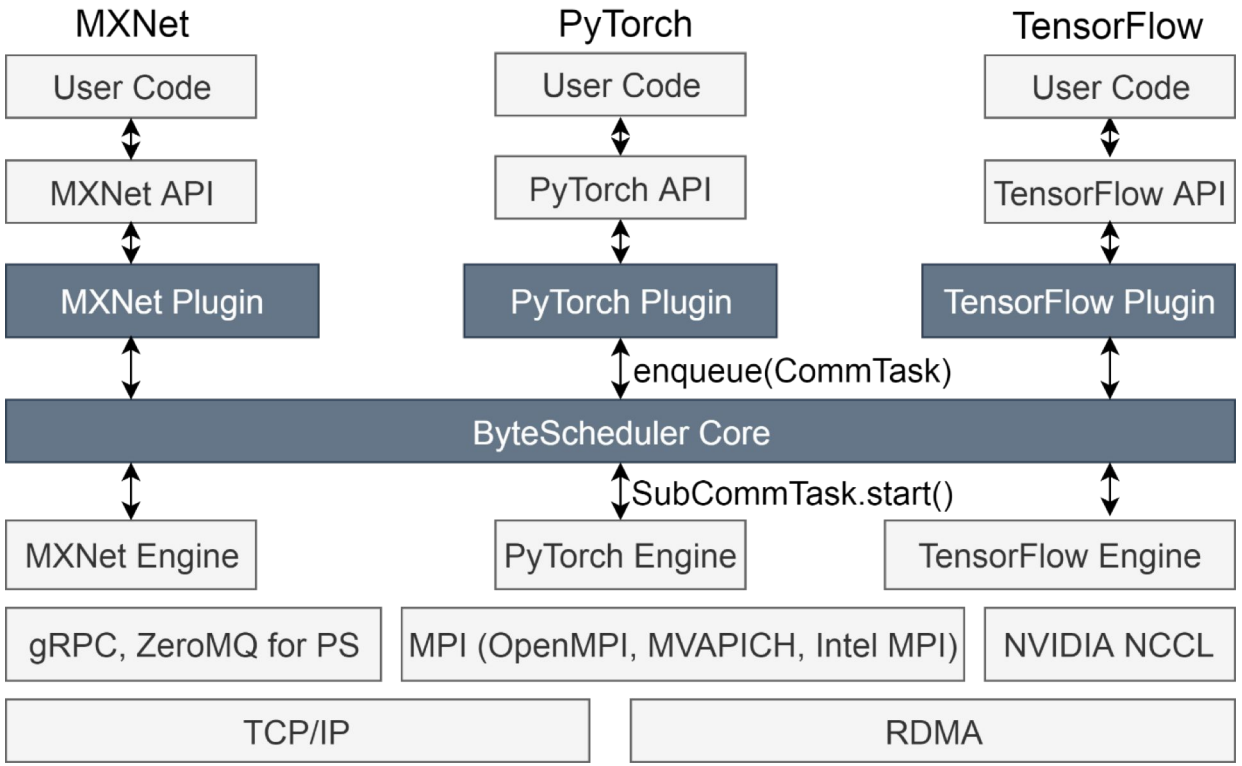
Frameworks-Agnostic Unified Scheduler

DNN definition

ByteScheduler

Graph execution

Communication



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

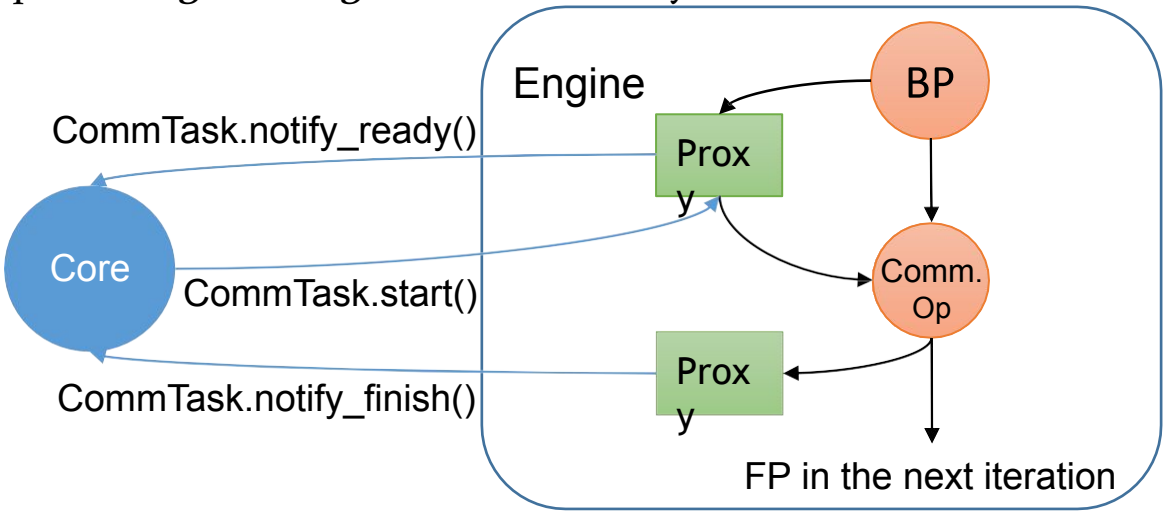
Dependency Proxy: Administers Scheduling Control

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

Dependency Proxy:

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

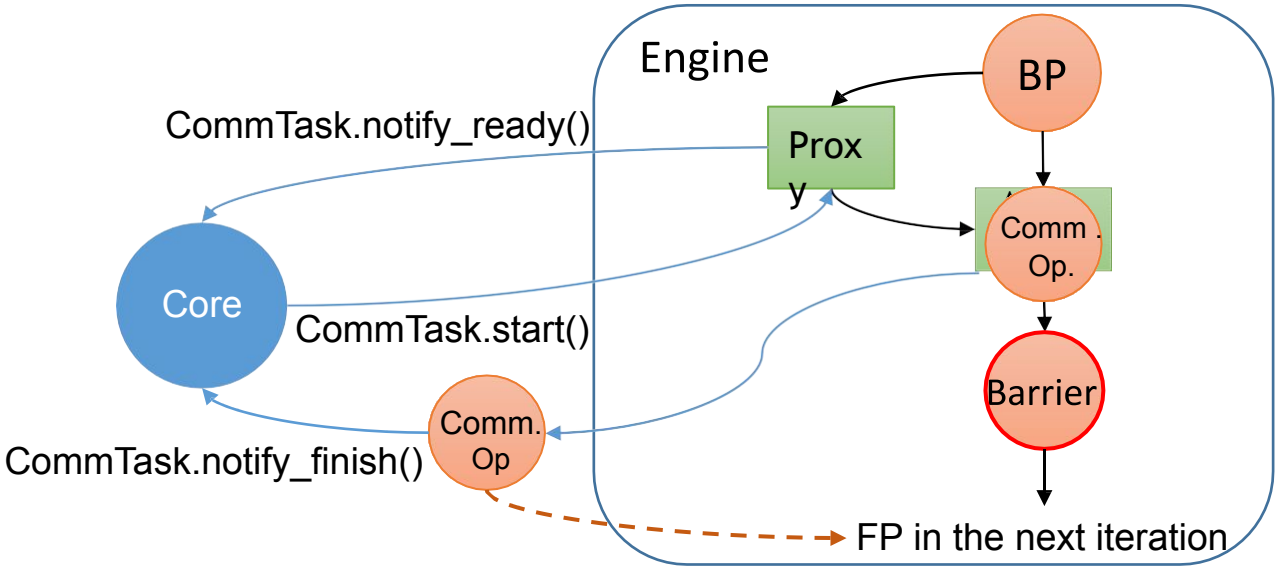
Different implementation for imperative, declarative engines



Dependency Proxy: Crossing the Global Barrier

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

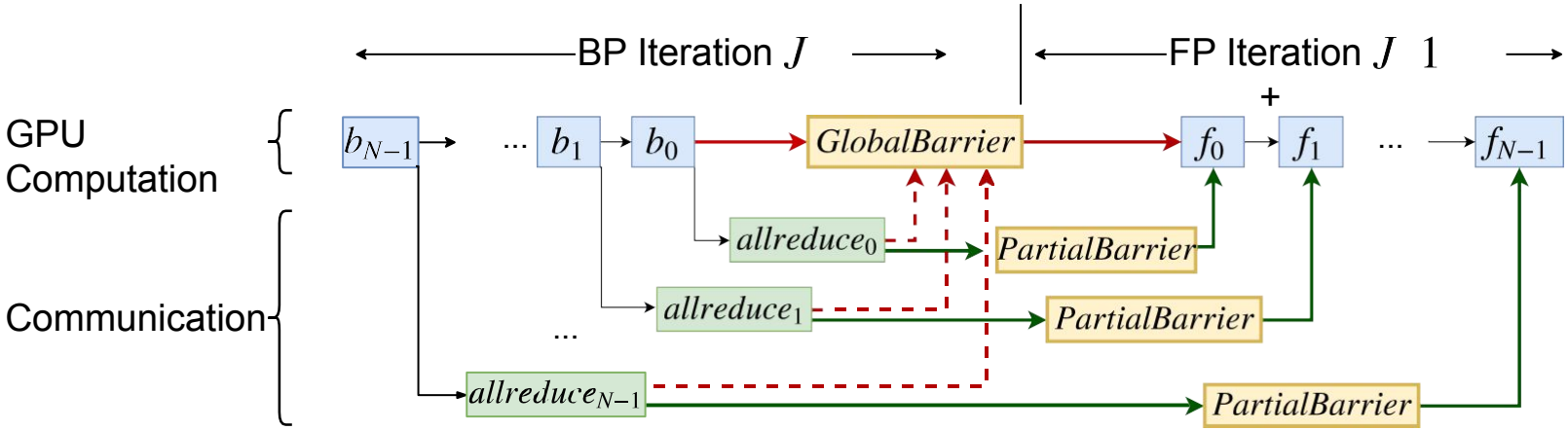
Layer-wise out-of-engine dependencies: Build correct dependency for each layer by adding a Proxy to block forward computation.



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

In practice, both partitioning and preemption may have overheads.

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

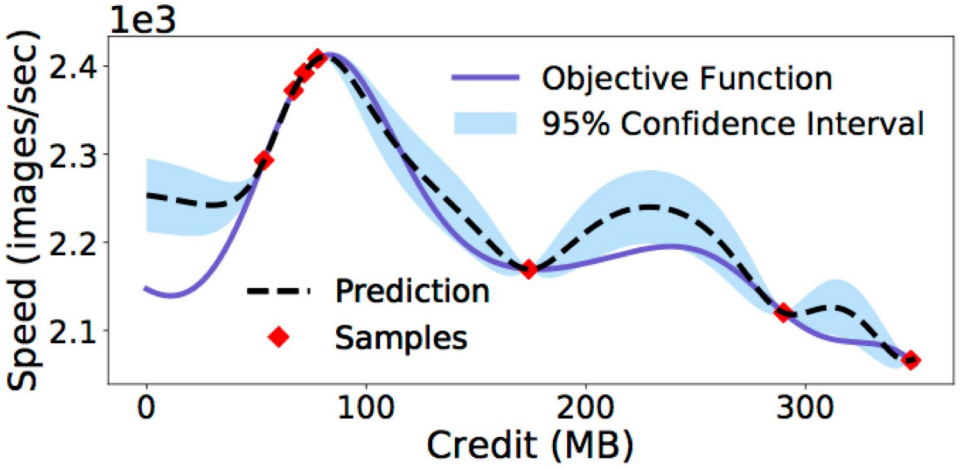
- Pro: higher bandwidth utilization
- Con: less timely preemption due to FIFO communication stack

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



Evaluation

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

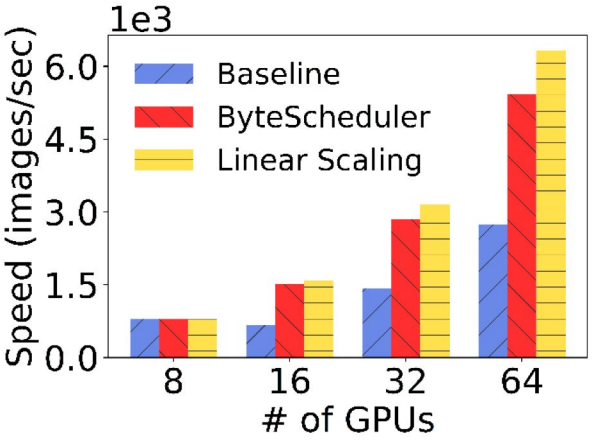
```
# After user created an MXNet KVStore object kvs
from bytescheduler.mxnet.kvstore import ScheduledKVS
kvs = ScheduledKVS(kvs)
# Continue using kvs without any further modification
```

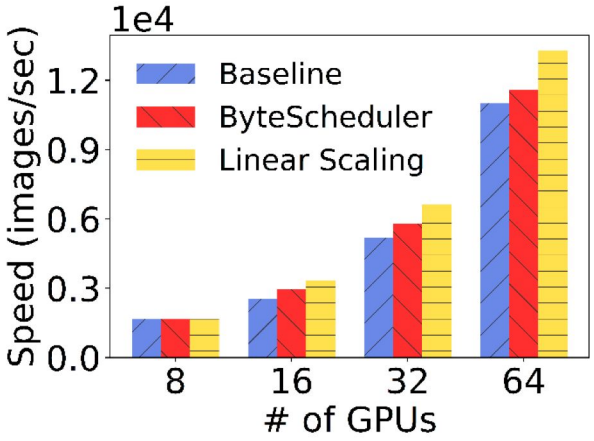
Testbed: 16 machines, each with 8 Tesla V100 GPUs and 100 Gbps bandwidth

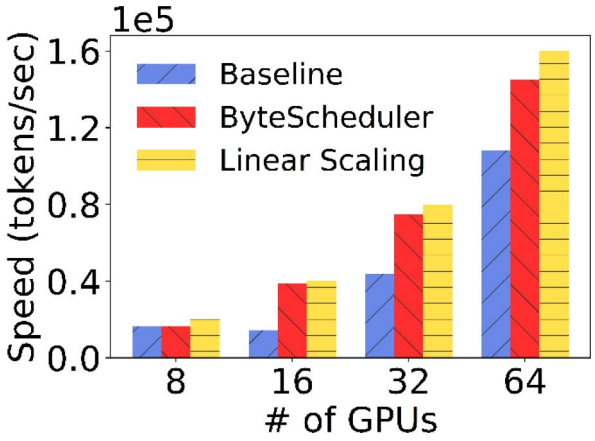
Comparison:

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

ByteScheduler Scalability







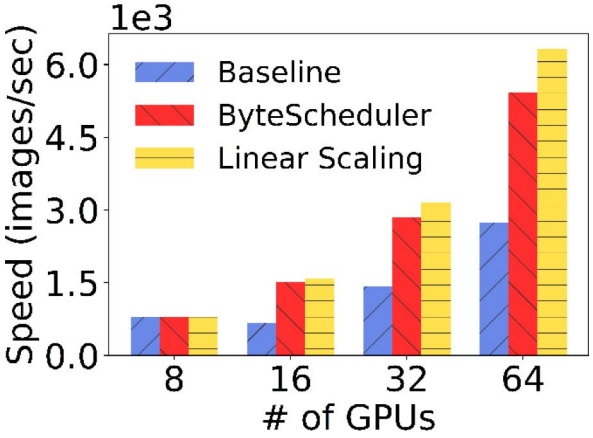
VGG16 (97%-125%)ResNet50 (9%-15%)

Transformer (70%-171%)

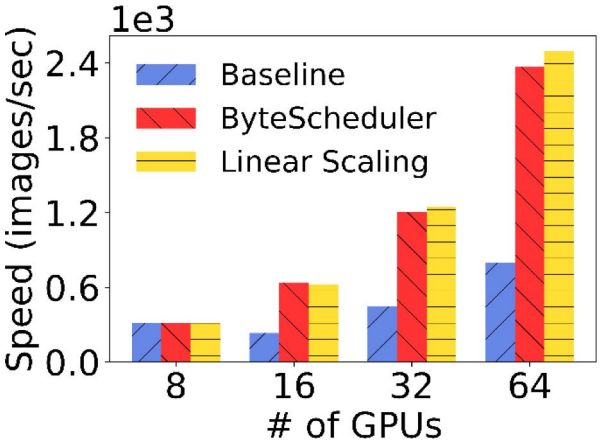
MXNet PS RDMA

- Up to 2x improvement and approximates linear scaling

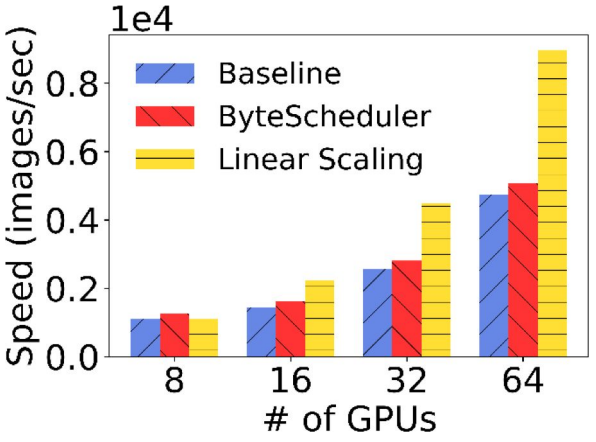
ByteScheduler for Multiple Frameworks



MXNet PS RDMA (97%-125%)



TensorFlow PS RDMA (170%-196%)

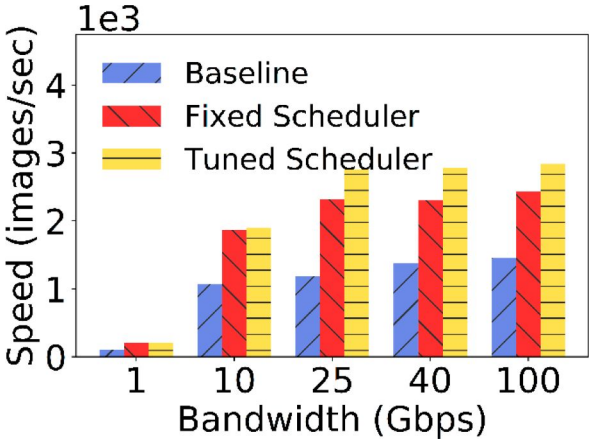


PyTorch NCCL TCP (7%-13%)

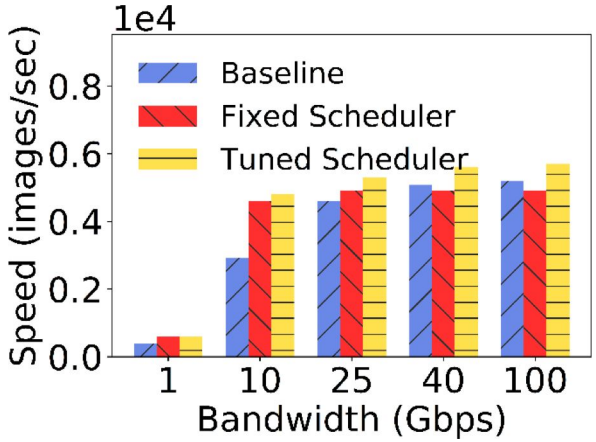
VGG16

- Up to 2x improvement compared to the baseline

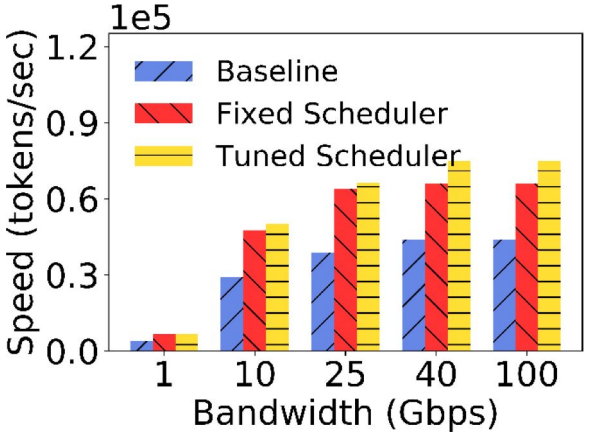
ByteScheduler Adapts to Different Bandwidths



VGG16 (79%-132%)



ResNet50 (10%-64%)

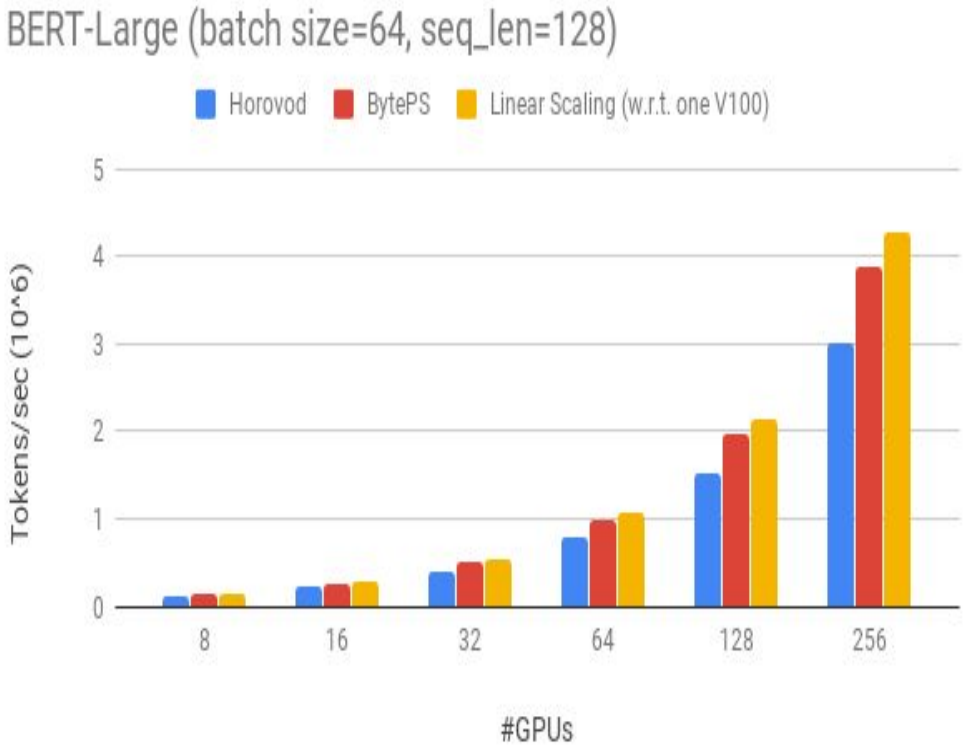


Transformer (67%-70%)

MXNet PS RDMA

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

ByteScheduler Performance



1. Achieves ~90% scaling efficiency for BERT-large with 256 GPUs.

2. Scaling efficiency with Horovod+NCCL with optimal parameter tuning is ~70%.

3. Over slower network, ByteScheduler over 2x speedup compared to Horovod+NCCL.

Conclusion

ByteScheduler: A generic communication scheduler for distributed DNN training acceleration.

- Unified abstraction for tensor scheduling
- Principled tensor scheduling design with parameter autotuning
- Generalizable across training frameworks
- Superior cloud, shared clusters speedups over traditional MPI paradigm.
- Incorporates several accelerations including hierarchical strategy, pipelining, tensor-partitioning, NUMA-aware local communications, priority-based scheduling, etc.

Limitations

- Missing support for pure CPU training.

- Lack of support for following features in ByteScheduler Architecture:
  1. Sparse model training
  2. Fault Tolerance
  3. Straggler-mitigation

- Missing support for aggressive compressions including gradient compression.