

Distributed Deep Learning

Wei Zhang, Giacomo Domeniconi, Ulrich Finkler

CSCI-GA.3033-022 HPML

Performance Factors

Algorithms Performance

- **Communication patterns**
- **Distributed DL Algorithms**

Hyperparameters Performance

- Hyperparameters choice

Implementation Performance

- Implementation of the algorithms on top of a framework

Framework Performance

- Python, **PyTorch Distributed DL**

Libraries Performance

- CUDA, cuDNN, cuBLAS, **Communication Libraries (MPI, Gloo)**

Hardware Performance

- CPU, DRAM, GPU, HBM, Tensor Units, Disk/Filesystem, **Network**

Parallel and Distributed Deep Learning Algorithms

DL Parallelism Approaches

- **Model Parallelism**
 - split model across multiple compute engines
- **Pipelining**
 - use layers as pipeline to keep all compute engines busy
- **Data Parallelism**
 - split data across multiple compute engines
- **Hybrid Parallelism**
 - use multiple approaches together

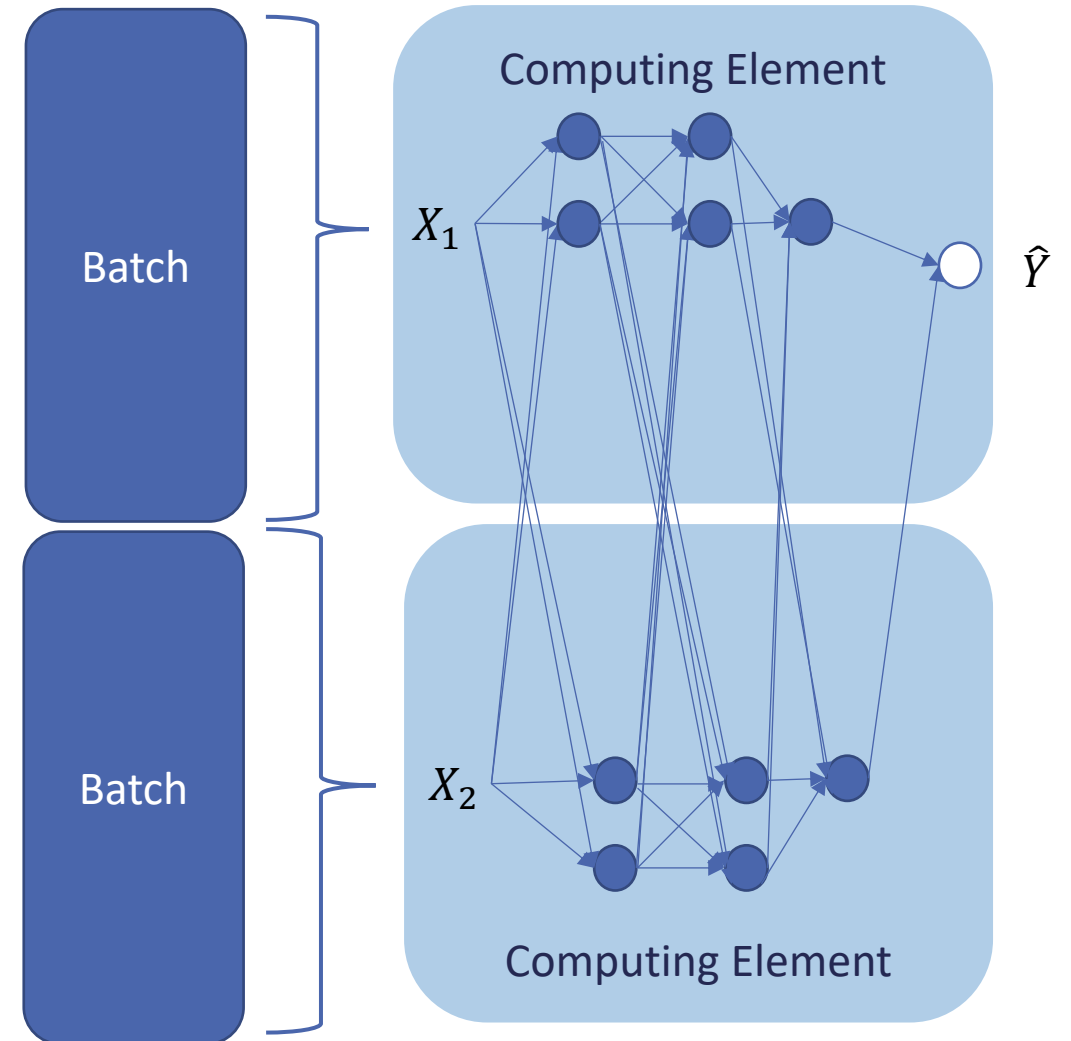
ML Model Parallelism

- **Model Parallelism:**

1. Divide the model in parts
2. Each computing element (CPU core or GPU thread) receives its part of the model
3. Send same batch to the computing elements
4. Model merges at the end

- **Characteristics:**

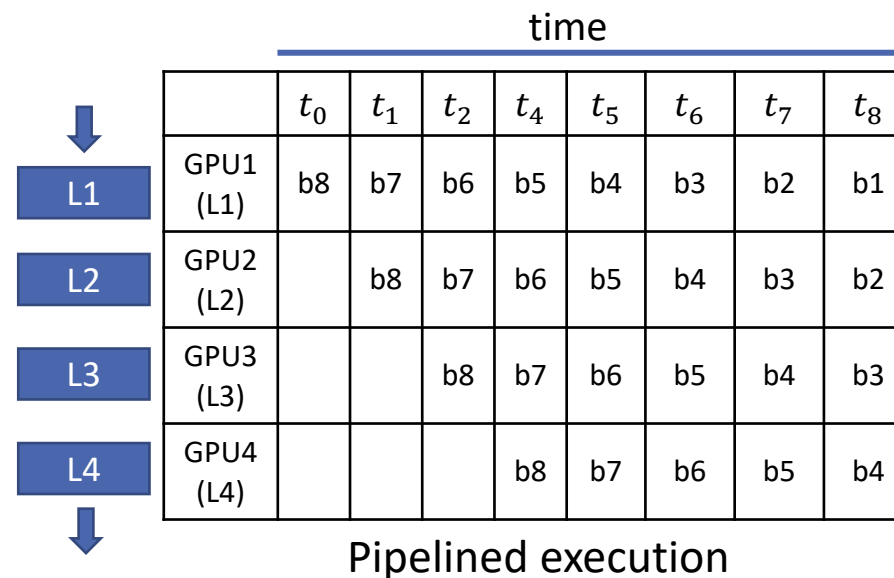
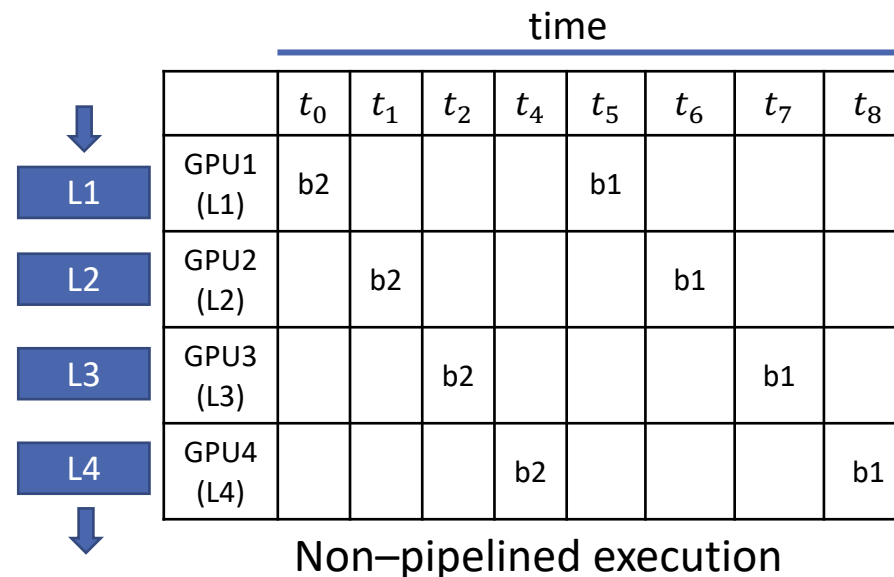
- **Latency of communication between the computing elements degrades performance**
- Harder to obtain performance if computing elements are far apart
- Used among GPU threads or CPU cores/threads



DL Pipelining

- Pipelining approach:
 - Split layers among compute engines
 - Each minibatch b (or sample s) goes from one compute engine to the next one: *no need to wait for next one to exit the pipeline*
- Is a form of **Model Parallelism**
- Pipelining performance
 - Ideal pipelining speedup:

$$S = \frac{\text{time without pipeline}}{\text{number of pipeline stages}}$$
 - Speedup is higher for deeper networks
 - Ideal pipelining never reached because of “bubbles” that cause idle CPUs
 - SGD pipeline bubble:
 - Before weights update, all batches need to have completed forward (otherwise accept **staleness**)



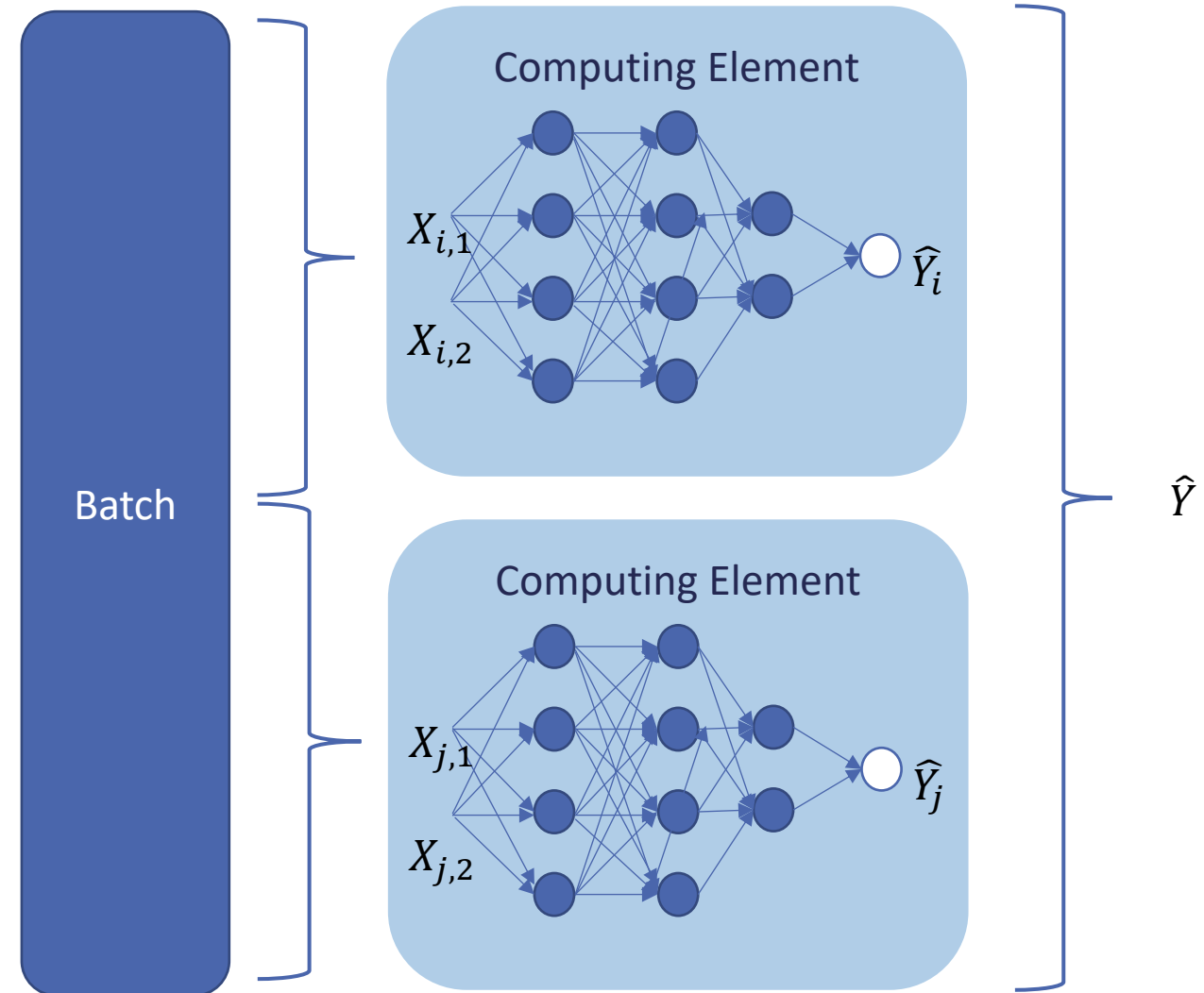
ML Data Parallelism

- **Data Parallelism:**

1. replicate model on each computing element (CPU core, GPU thread, or Cluster node)
2. Each computing element gets a portion of the batch
3. Each computing element trains/infers in parallel on the model
4. Gather the results at the end.

- **Characteristics:**

- Used among GPUs devices or among cluster nodes
- In PyTorch: DataParallel
- Can be also implemented using different batches instead of parts of one batch (typically among nodes in a cluster)



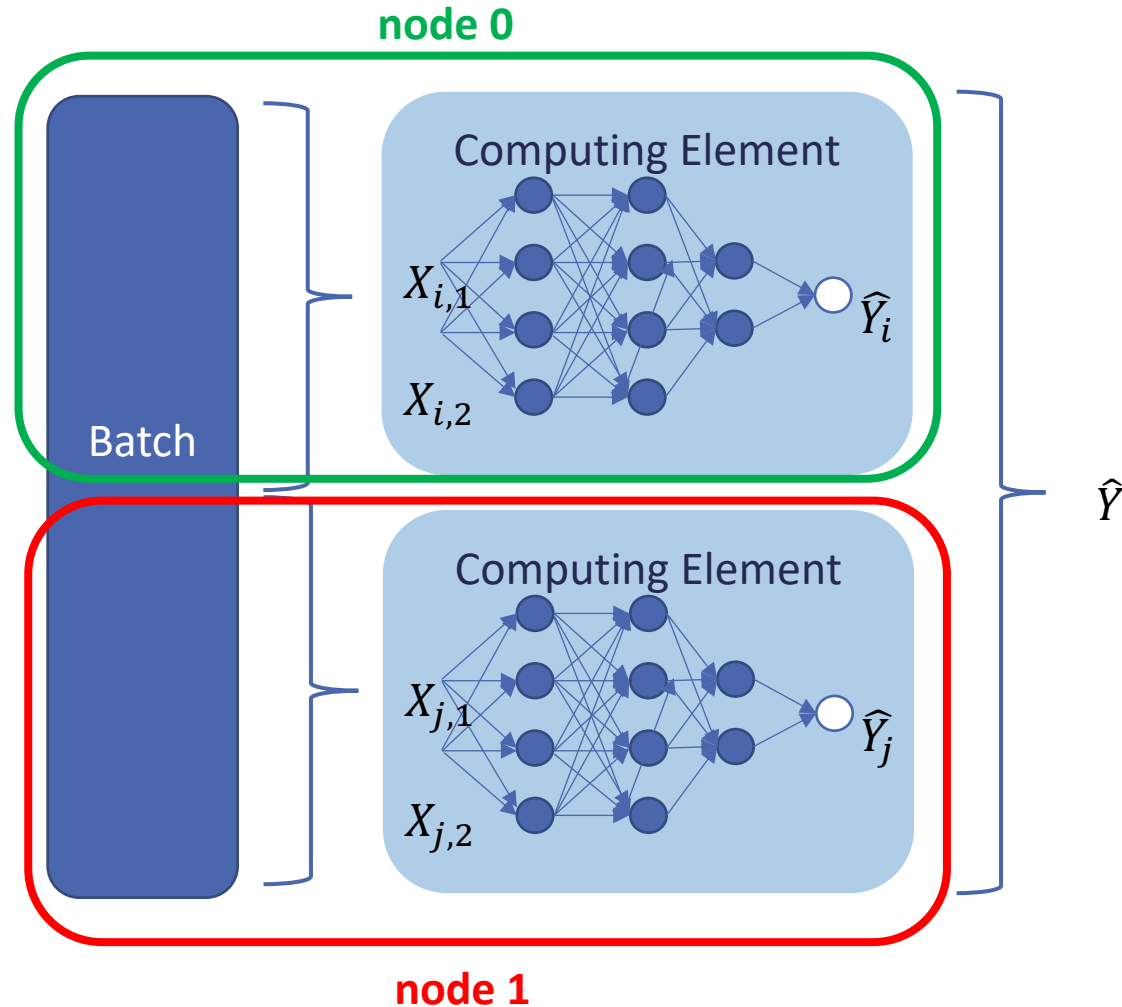
Distributed Data Parallelism

- **Distributed Data Parallelism**

- Computing elements belong to different nodes and processes

- **Requirements:**

- Parallel-Distributed SGD algorithm
- Communication infrastructure
- In Pytorch: `DistributedDataParallel`



Synchronous Distributed SGD

- In the synchronous setting, all replicas average all of their gradients at every timestep (minibatch)
 - Doing so, we're effectively multiplying the batch size M by the number of replicas R
- This has several advantages:
 - The computation is completely deterministic
 - We can work with fairly large models and large batch sizes even on memory-limited GPUs
 - It's very simple to implement, and easy to debug and analyze

Synchronous Distributed SGD

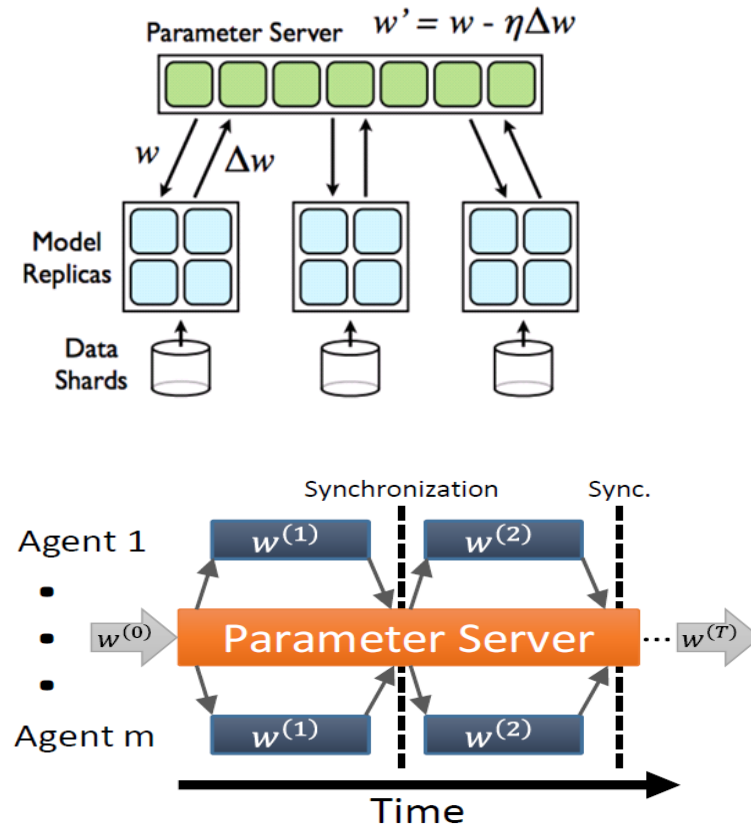
- Distributed data parallel SGD is basically the classic minibatch SGD with lines 3 and 7 modified to read/write weights from/to a parameter store which may be centralized or decentralized
- This incurs an overhead on the overall system

Algorithm 2 Minibatch Stochastic Gradient Descent with Backpropagation

```
1: for  $t = 0$  to  $\frac{|S|}{B} \cdot epochs$  do
2:    $\vec{z} \leftarrow$  Sample  $B$  elements from  $S$                                  $\triangleright$  Obtain samples from dataset
3:    $w_{mb} \leftarrow w^{(t)}$                                                $\triangleright$  Load parameters
4:    $f \leftarrow \ell(w_{mb}, \vec{z}, h(\vec{z}))$                                  $\triangleright$  Compute forward evaluation
5:    $g_{mb} \leftarrow \nabla \ell(w_{mb}, f)$                                  $\triangleright$  Compute gradient using backpropagation
6:    $\Delta w \leftarrow u(g_{mb}, w^{(0, \dots, t)}, t)$                      $\triangleright$  Weight update rule
7:    $w^{(t+1)} \leftarrow w_{mb} + \Delta w$                                  $\triangleright$  Store parameters
8: end for
```

Synchronous Distributed SGD – (Parameter Server)

1. MiniBatch is partitioned among workers and model is replicated on each worker
2. A **Parameter server** contains copy of the model
3. Each worker executes forward and backward propagation on its minibatch portion
4. Gradients are sent to the parameter server that computes the update
5. The workers receive updated models from the parameter server

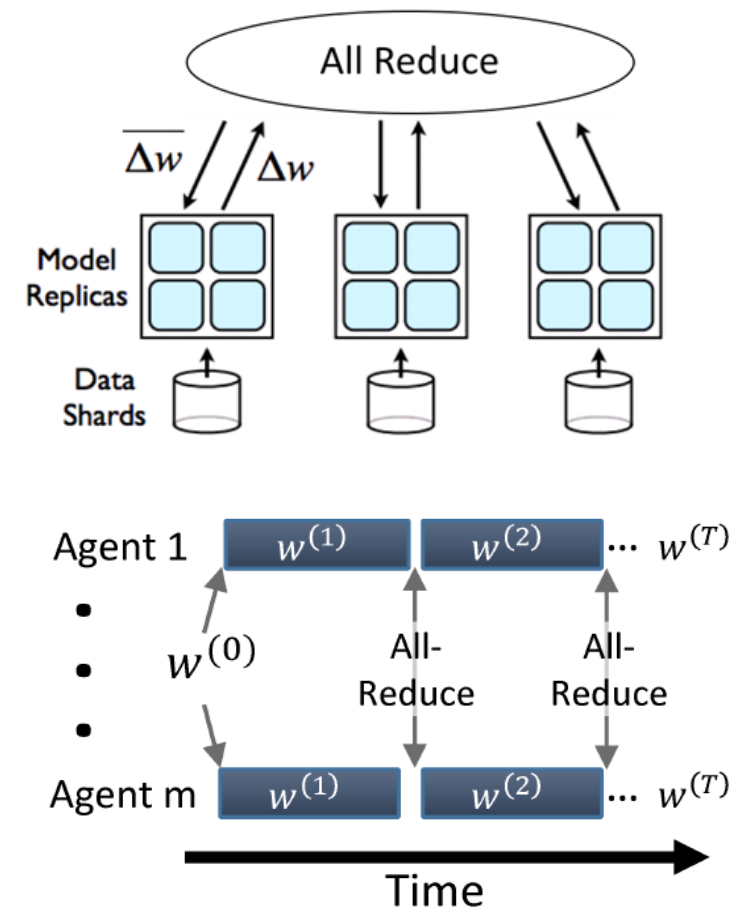


Synchronous Centralized Distributed SGD

from: <https://arxiv.org/abs/1802.09941>

Synchronous Distributed SGD – (All Reduce)

1. MiniBatch is partitioned among workers and model is replicated on each worker
2. Each worker executes forward and backward propagation on its minibatch portion
3. Gradients are averaged with all-reduce operation
4. Models are updated with the same average gradients



Synchronous Decentralized Distributed SGD
from: <https://arxiv.org/abs/1802.09941>

Asynchronous Distributed SGD

- SGD can be made **asynchronous (ASGD)**
 - Relax timing of updates
 - Relax the synchronization restriction creating an **inconsistent model**
 - Each worker updates when it completes computing gradients
 - **Staleness (or lag)** of a model:
 - interval of time between the last update to the parameter server and the current model
 - **Stragglers problem:**
 - some workers may be very late (stragglers) and use a model with high staleness

Asynchronous stochastic gradient descent (ASGD)

- Each learner asynchronously repeats the following:
 - **Pull:** Get the parameters from the server
 - **Compute:** Compute the gradient with respect to randomly selected mini-batch (i.e., a certain number of samples from the dataset)
 - **Push and update:** Communicate the gradient to the server. The server then updates the parameters by subtracting this newly communicated gradient multiplied by the learning rate
- Reduces synchronization and communication overhead by tolerating stale gradient updates
- Recent analyses show ASGD converges with linear asymptotic speedup over SGD
- Examples: Downpour and EAMSGD

Some history of the Distributed asynchronous SGD

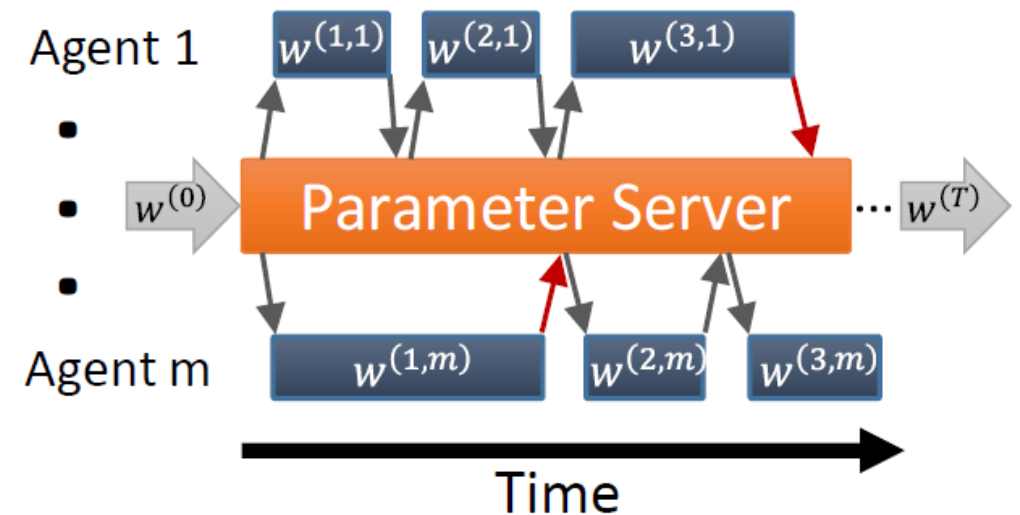
- Distributed asynchronous SGD method was introduced by [Tsitsiklis1986].
- [Zinkevich2010] proposed the first synchronous parallel SGD algorithm by averaging the individual parameter after each local update

$$\tilde{\mathbf{w}}_N = \frac{1}{N} \sum_{j=1}^N \mathbf{w}_N^j, \mathbf{w}_N^j = \tilde{\mathbf{w}}_{N-1} - \gamma \nabla F(\mathbf{w}_{N-1}; \xi_N^j), j = 1, \dots, P.$$

- [Recht2011] proposed HOGWILD, a lock free (asynchronous) parallel SGD (ASGD) method, and gained great success in convex optimization realm.
- [Dean2012] proposed Downpour, a popular ASGD method to train deep networks.

HOGWILD - lock free asynchronous parallel SGD

- **HOGWILD** is a well known shared memory algorithm to run SGD in parallel without locks for problems with sparse separable cost functions
- A training agent i at time t contains a copy of the weights, denoted as $w(\tau, i)$ for $\tau \leq t$
 - $t - \tau$ is the staleness (or lag)
- It was originally designed for shared-memory architectures but has been extended to distributed-memory systems, in which it still attains convergence for deep learning problems



Asynchronous Centralized Distributed SGD

from: <https://arxiv.org/abs/1802.09941>

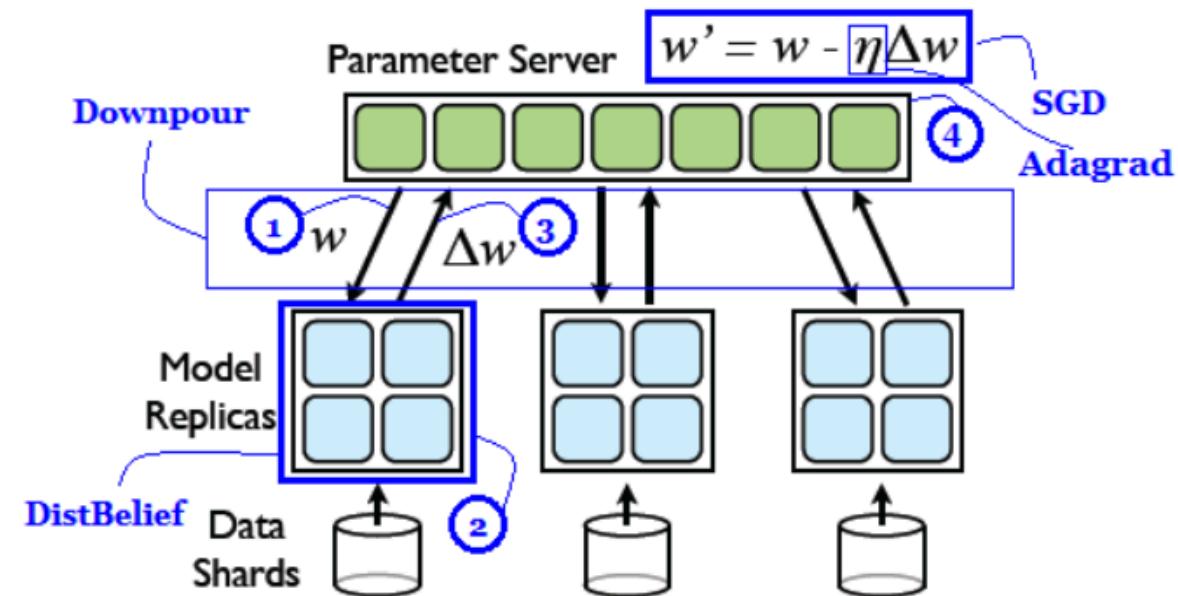
Paper: B. Recht, C. Re, S. Wright, and F. Niu. 2011. Hogwild: A Gradient lock-Free Approach to Parallelizing Stochastic Descent. In Advances in Neural Information Processing Systems.

HOGWILD - lock free asynchronous parallel SGD

- HOGWILD allows training agents to read parameters and update gradients at will, overwriting existing progress
- HOGWILD has been proven to converge for sparse learning problems, where updates only modify small subsets of w
- Based on foundations of distributed asynchronous SGD, the proofs impose that
 - a) write accesses (adding gradients) are always atomic
 - b) Lipschitz continuous differentiability and strong convexity on f_w
 - c) the staleness, i.e., the maximal number of iterations between reading w and writing ∇w , is bounded
- To mitigate the interference effect of overwriting w at each step, the actual implementation transfers the gradient ∇w instead of w from the training agents

Downpour

- Implementation of inconsistent SGD (called Downpour SGD, based on HOGWILD) proposed in DistBelief
 - J. Dean et al. 2012. *Large Scale Distributed Deep Networks*. In Proc. 25th International Conference on Neural Information Processing Systems - Volume 1 (NIPS'12). 1223–1231.
- In Downpour, whenever a worker computes a gradient (or a sequence of gradients), the gradient is sent to the parameter server
- When the parameter server receives the gradient update from a worker, it will incorporate the update in the center variable.



Downpour – Pseudocode

Algorithm 1.1: DOWNPOURSGDCLIENT($\alpha, n_{\text{fetch}}, n_{\text{push}}$)

procedure STARTASYNCHRONOUSLYFETCHINGPARAMETERS(*parameters*)
 parameters \leftarrow GETPARAMETERSFROMPARAMSERVER()

procedure STARTASYNCHRONOUSLYPUSHINGGRADIENTS(*accruedgradients*)
 SENDGRADIENSTOPARAMSERVER(*accruedgradients*)
 accruedgradients \leftarrow 0

main

global *parameters, accruedgradients*

step \leftarrow 0

accruedgradients \leftarrow 0

while true

do {
 if (*step* mod n_{fetch}) == 0
 then STARTASYNCHRONOUSLYFETCHINGPARAMETERS(*parameters*)
 data \leftarrow GETNEXTMINIBATCH()
 gradient \leftarrow COMPUTEGRADIENT(*parameters, data*)
 accruedgradients \leftarrow *accruedgradients* + *gradient*
 parameters \leftarrow *parameters* - α * *gradient*
 if (*step* mod n_{push}) == 0
 then STARTASYNCHRONOUSLYPUSHINGGRADIENTS(*accruedgradients*)
 step \leftarrow *step* + 1
 }

1

2

3

From:

http://admis.fudan.edu.cn/~yfhuang/files/LSDDN_slide.pdf

Downpour – Drawbacks of frequent communication

- Downpour does not assume any communication constraints (as opposite to EASGD), and even more, if frequent communication with the parameter server does not take place (in order to reduce worker variance), Downpour will not converge (this is also related to the *asynchrony induces momentum* issue)
 - If we allow the workers to explore "too much" of the parameter space, then the workers will not work together on finding a good solution for the center variable
- Downpour does not have any intrinsic mechanisms to remain in the neighborhood of the center variable
- As a result, if you would increase the communication window, you would *proportionally* increase the length of the gradient which is sent to the parameter server, thus, the center variable is updated more aggressively in order to keep the variance of the workers in the parameter space “small”

Model Accuracy and Runtime Tradeoff in Distributed Deep Learning: A Systematic Study

{Suyog Gupta, **Wei Zhang**}@IBM T.J.Watson

{Fei Wang}@Cornell

Executive Summary

- Motivation
- Experiment methodology
- Several lessons we've learnt
 - Staleness-aware ASGD
 - Staleness, Mini-batch, #Learners interplay
 - Novel parameter server design
- Conclusion and future work

Motivation

- We want to understand the interplay between hyper parameter tuning (e.g., mini-batch size, learning rate) and system design (e.g., number of learners, communication patterns) for large scale deep learning.
- In this paper , we focus on how to build ASGD based deep learning framework that runs fast while maintaining good model accuracy.

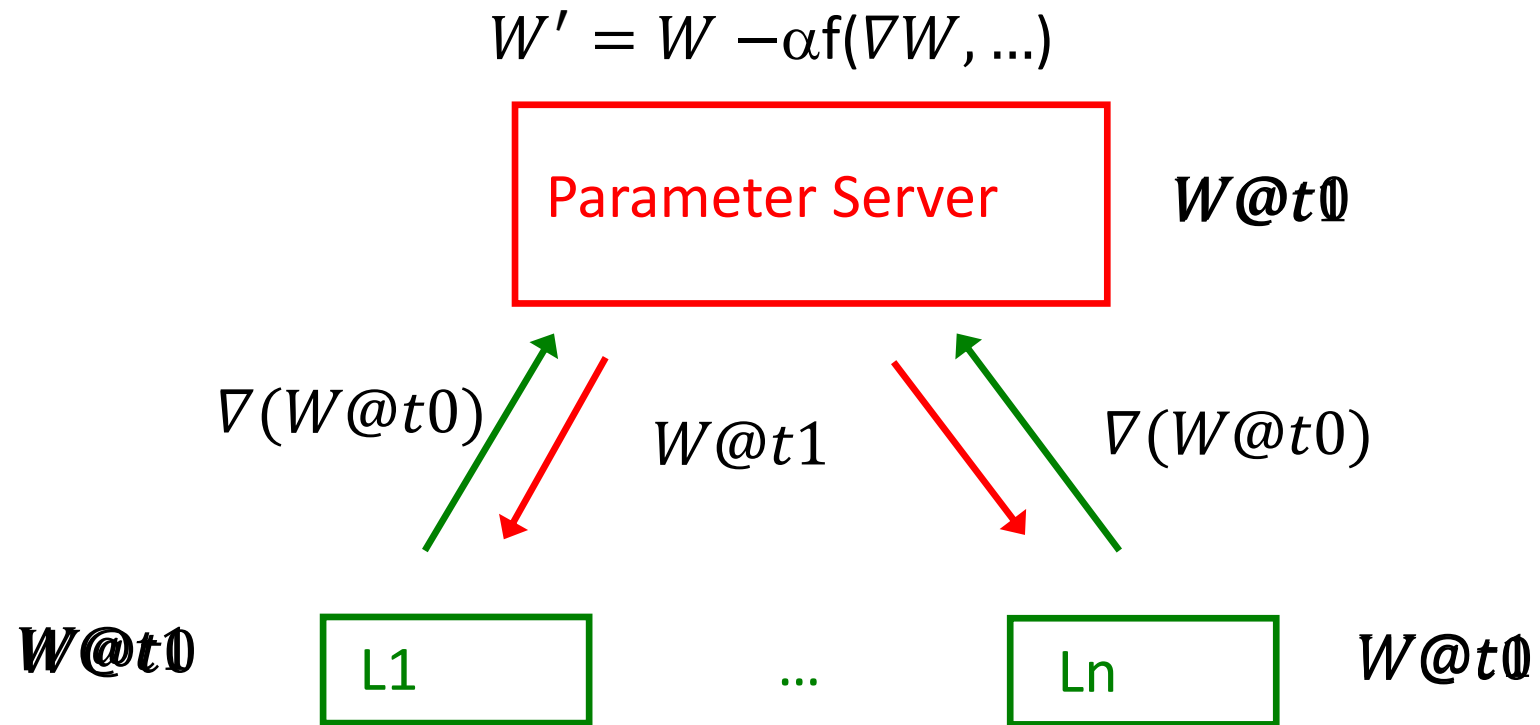
Methodology

- Rudra, in-house implementation of NN (Power/Z/x86/GPU)
- MPI-based distributed implementation runs on P775 super computer
 - Each node contains 4 8-core 3.84GHz P7 processor (92 such nodes)
 - 128GB memory per node
 - 64GB/s main memory bandwidth, 24GB/s bi-directional p7ih inter-connect.
- CIFAR10(convnet)/ImageNet (AlexNet)

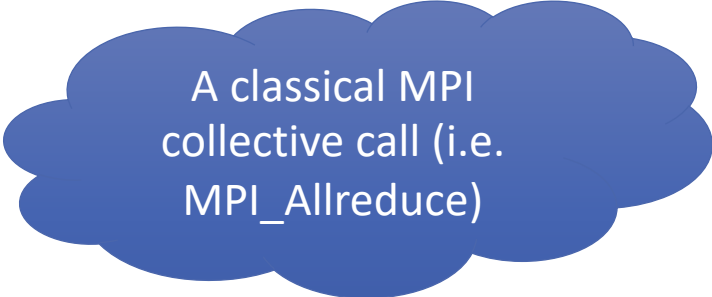
Terminologies

- σ , staleness of the gradients. (def is given in 2 slides)
- μ , minibatch size.
- λ , number of learners.

Hard-sync



Hard-sync



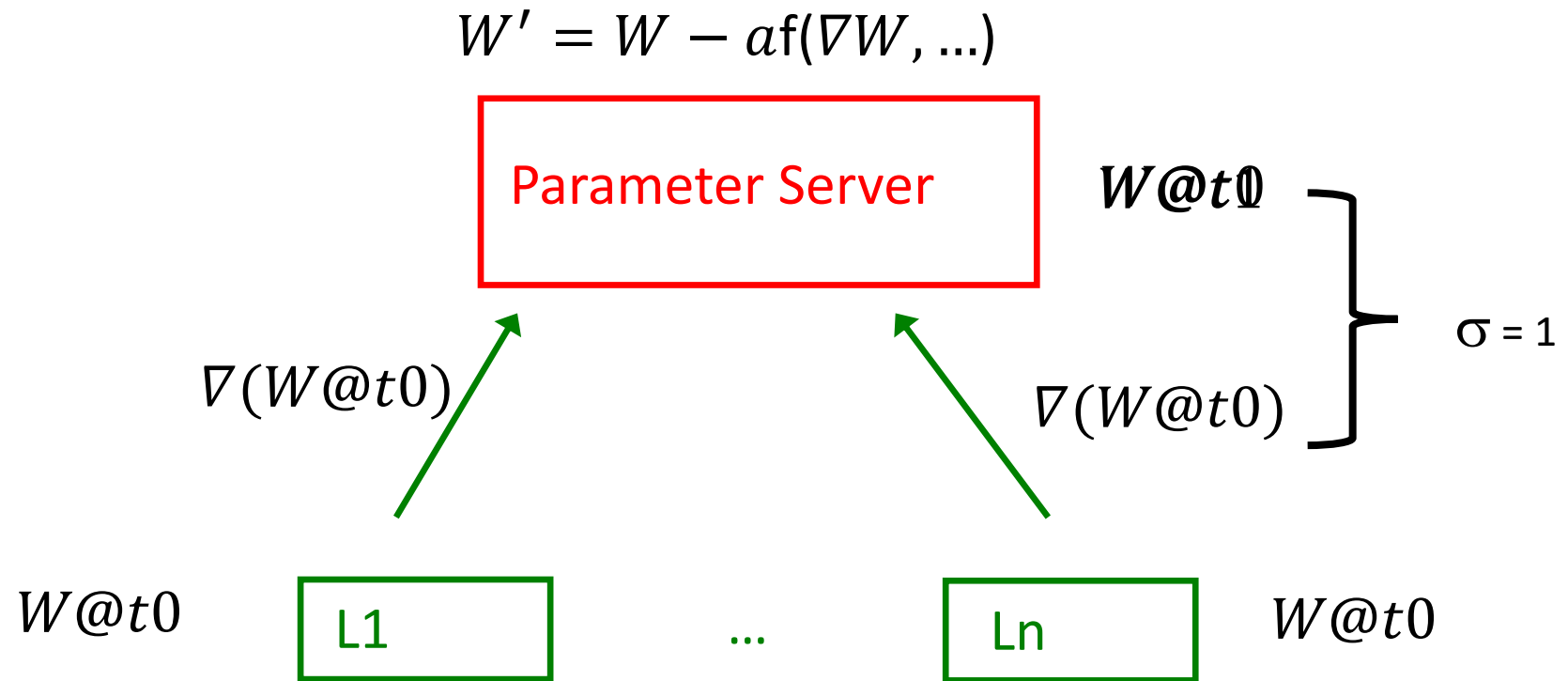
A classical MPI
collective call (i.e.
MPI_Allreduce)

$$\nabla W(i) = \frac{1}{\lambda} \sum_{l=1}^{l=\lambda} \nabla W_l(i)$$

$$W(i+1) = W(i) - \alpha \nabla W(i)$$

Equivalent to single-learner SGD w/ batchsize of $\lambda * \mu$

Soft-sync



n-softsync

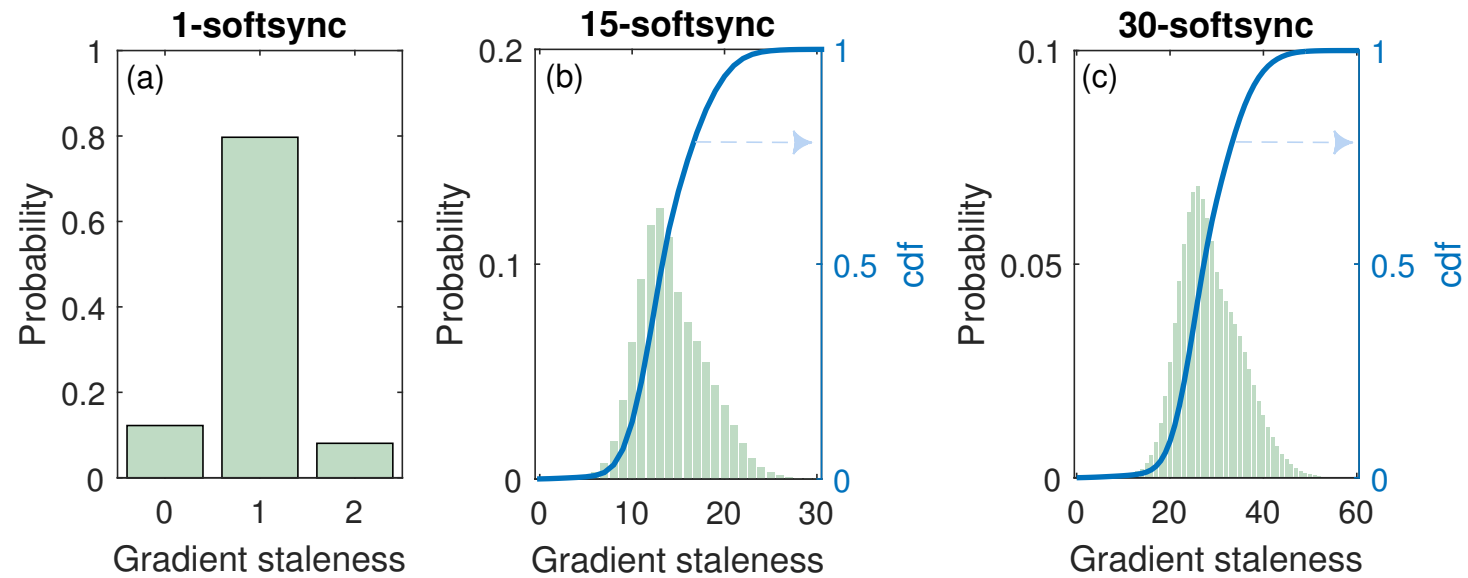
$n=1$, similar to hardsync
 $n = \lambda$, equivalent to Downpour
 n dictates system staleness

c

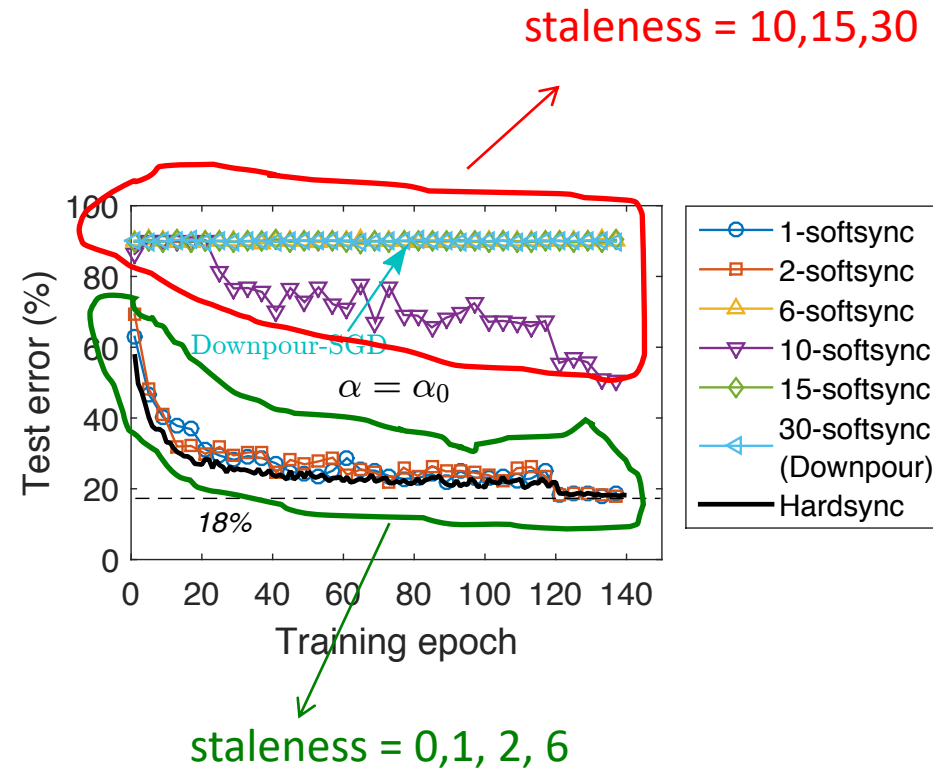
$$\nabla W(i) = \frac{1}{c} \sum_{l=1}^{l=c} \nabla W_l(i)$$

$$W(i+1) = W(i) - \alpha \nabla W(i)$$

Staleness distribution



Model Accuracy (baseline)

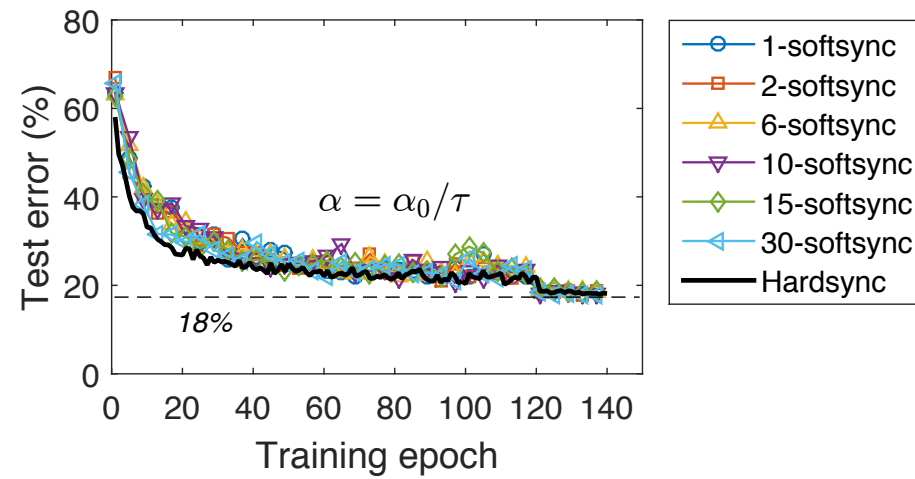


CIFAR10

Staleness-aware learning rate modulation

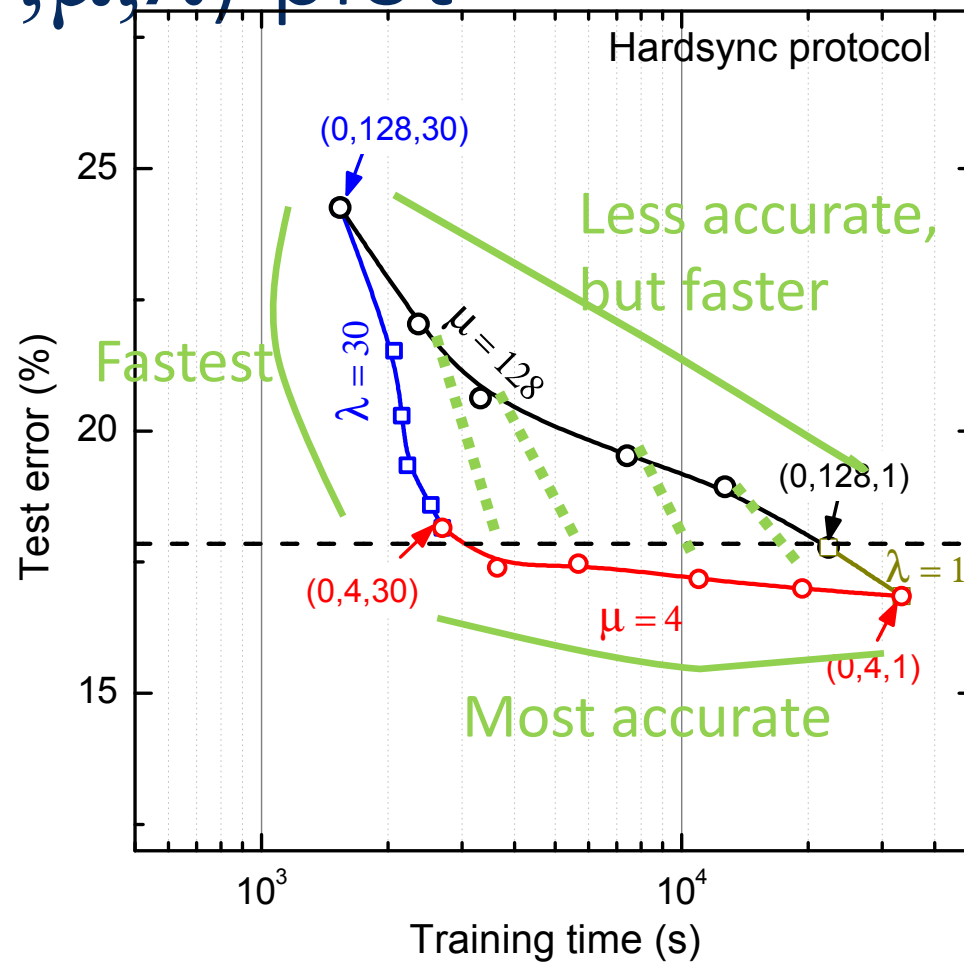
$$\alpha = \frac{\alpha_0}{\langle \delta \rangle}$$

Model Accuracy (our scheme)

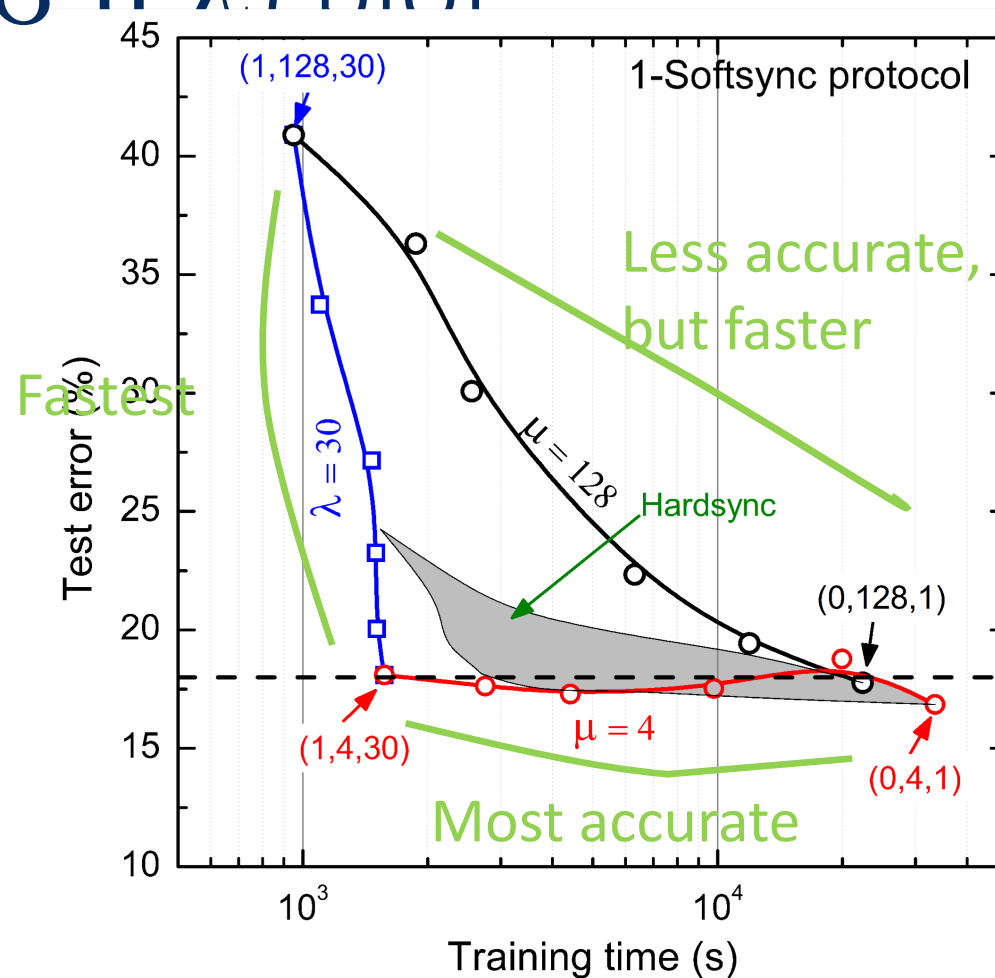


CIFAR10

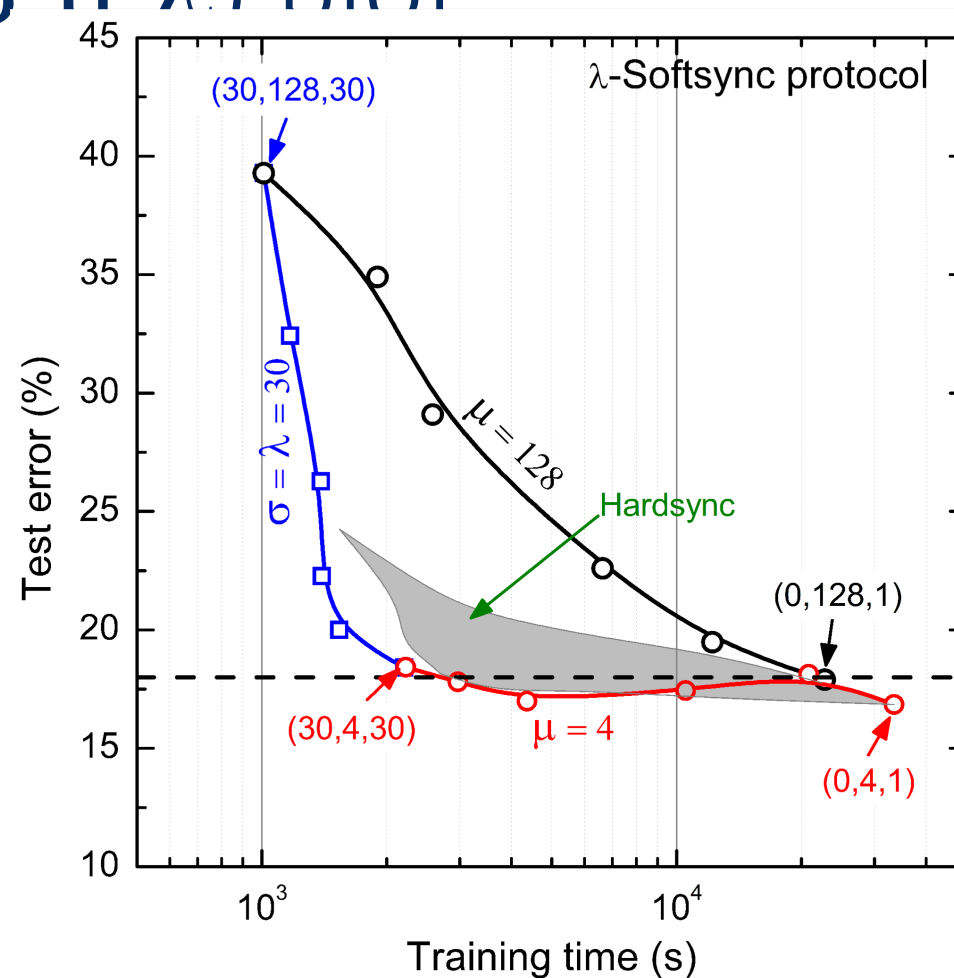
Hard-sync (σ, μ, λ) plot



1-softsync (σ μ λ) plot



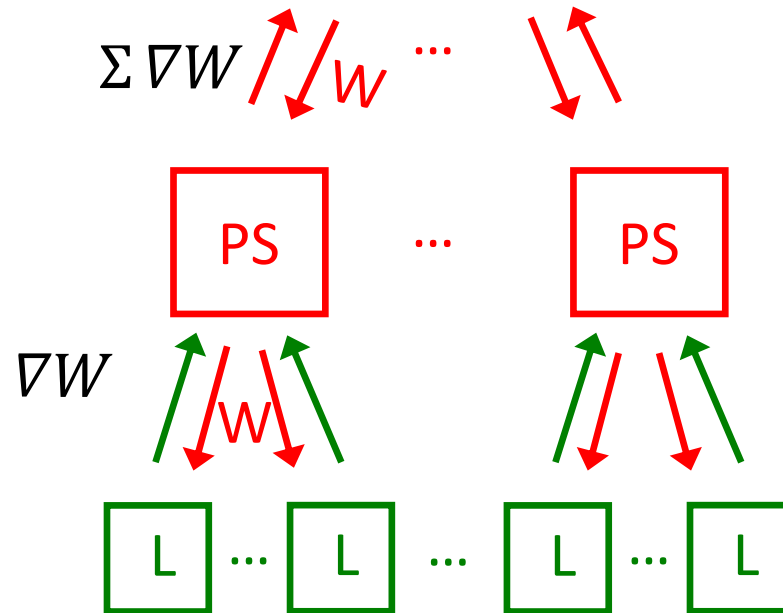
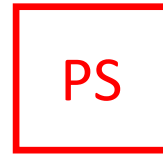
λ -softsync ($\sigma \parallel \lambda$) plot



$\mu * \lambda$ constant

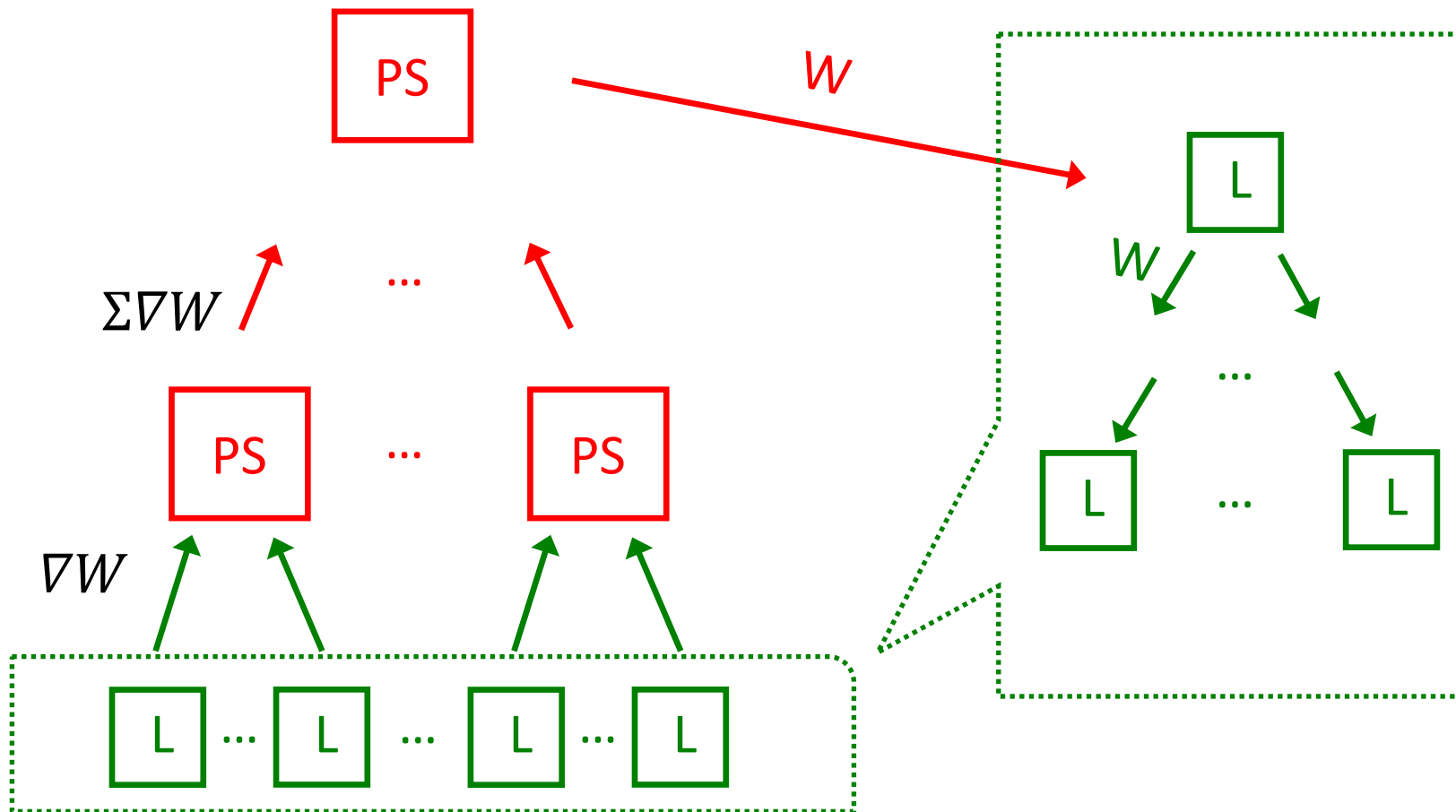
	σ	μ	λ	Test error	Training time(s)	
$\mu\lambda \approx 128$	1	4	30	18.09%	1573	$\sim 18\%$
	30	4	30	18.41%	2073	
	18	8	18	18.92%	2488	
	10	16	10	18.79%	3396	
	4	32	4	18.82%	7776	
	2	64	2	17.96%	13449	
$\mu\lambda \approx 256$	1	8	30	20.04%	1478	$\sim 20\%$
	30	8	30	19.65%	1509	
	18	16	18	20.33%	2938	
	10	32	10	20.82%	3518	
	4	64	4	20.70%	6631	
	2	128	2	19.52%	11797	
$\mu\lambda \approx 512$	1	16	30	23.25%	1469	$\sim 23\%$
	30	16	30	22.14%	1502	
	18	32	18	23.63%	2255	
	10	64	10	24.08%	2683	
	4	128	4	23.01%	7089	
$\mu\lambda \approx 1024$	1	32	30	27.16%	1299	$\sim 28\%$
	30	32	30	27.27%	1420	
	18	64	18	28.31%	1713	
	1	128	10	29.83%	2551	
	10	128	10	29.90%	2626	

Rudra-adv

$$W' = W - \alpha f(\nabla W, \dots)$$


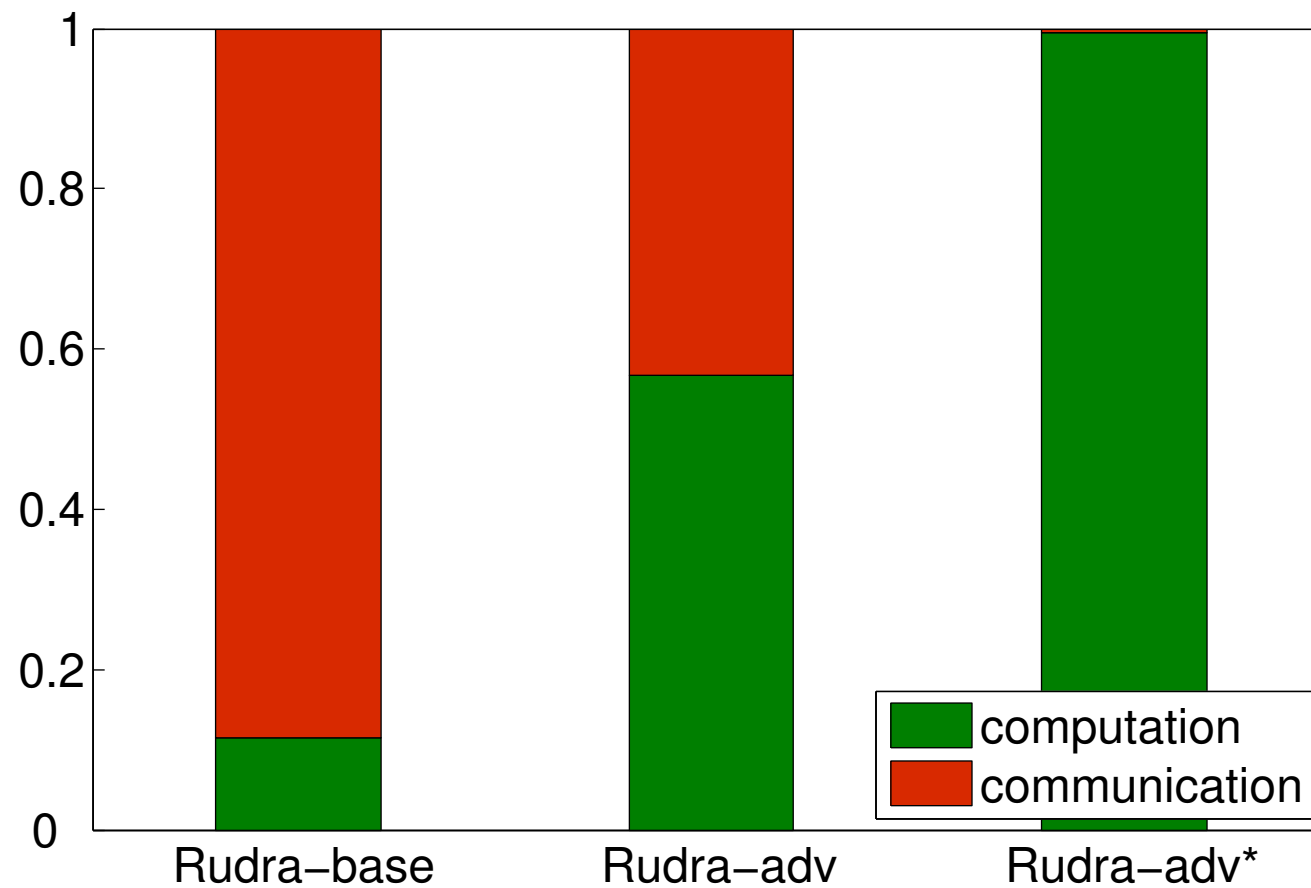
Rudra-adv*

$$W' = W - \alpha f(\nabla W, \dots)$$

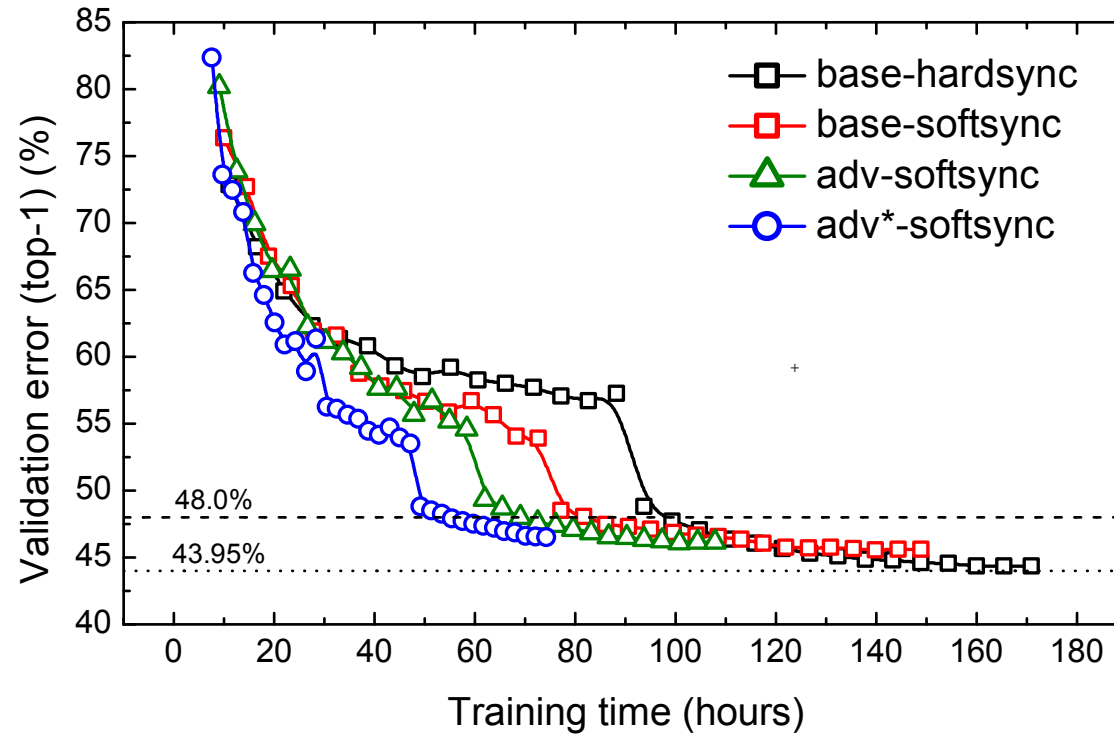


Communication vs Computation

$\lambda=54, \mu=4$, ImageNet



ImageNet model accuracy vs runtime



26X End-to-End speedup using 54 learners.

Conclusion

- We study the trade off of model accuracy and runtime in large-scale deep learning systems
- We made the following contributions
 - Proposed the staleness-aware learning rate modulation
 - Quantified the tradeoff of model accuracy vs runtime by studying the (σ, μ, λ) plots
 - Empirically observed the model accuracy dictated by $\mu * \lambda$
 - Proposed novel parameter server architecture

PyTorch Distributed DL

Data Parallelism: Multiple GPU configurations

- Single node:
 - **Multi-GPU with 1 Process (single node - preferred)**
 - Use `nn.DataParallel` for standard data-parallelism
 - Multi-GPU with Multiple Processes (single node)
 - Manual Handling of data/kernels with multiple devices/multiple processes: prone to errors
- Multi-node:
 - **Multi-GPUs with multiple-processes (multi-node - preferred)**
 - Use `nn.DistributedDataParallel` for standard data-parallelism

DataParallel Example

torch.nn.DataParallel(module, device_ids=None, output_device=None, dim=0)[source]

- Replicates models on each GPU
- Forward pass: **divides each batch into #devices and does it in parallel**
 - Batch size should be: larger than #devices and multiple of #devices
- Backward pass: **computes gradients separately and sums them into the original module**

```
class DataParallelModel(nn.Module):
```

```
    def __init__(self):
```

```
        super().__init__()
```

```
        self.block1 = nn.Linear(10, 20)
```

```
        # wrap block2 in DataParallel
```

```
        self.block2 = nn.Linear(20, 20)
```

```
        self.block2 = nn.DataParallel(self.block2)
```

```
        self.block3 = nn.Linear(20, 20)
```

```
    def forward(self, x):
```

```
        x = self.block1(x)
```

```
        # This will execute in parallel on the GPUs (if any)
```

```
        x = self.block2(x)
```

```
        x = self.block3(x)
```

```
        return x
```

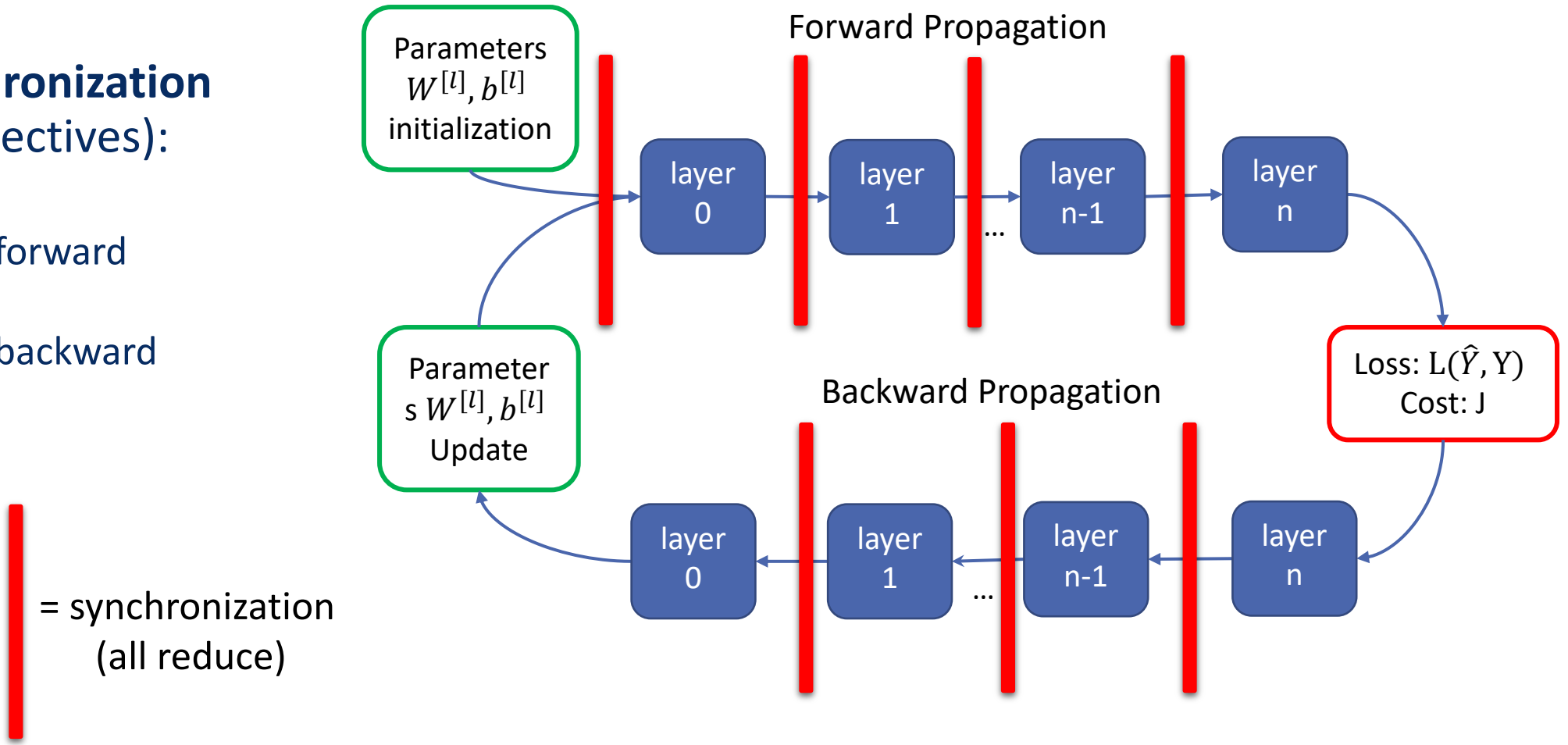
torch.nn.DistributedDataParallel

- torch.nn.DataParallel implements data parallelism at the “layer” (module in PyTorch) level:
 1. Splitting the input across the specified devices by chunking in the batch dimension
 2. The module is replicated on each machine and each device, and each replica handles a portion of the input.
 3. During the backward pass, gradients from each node are **averaged**
- <http://pytorch.org/docs/master/nn.html#torch.nn.parallel.DistributedDataParallel>

torch.nn.DistributedDataParallel

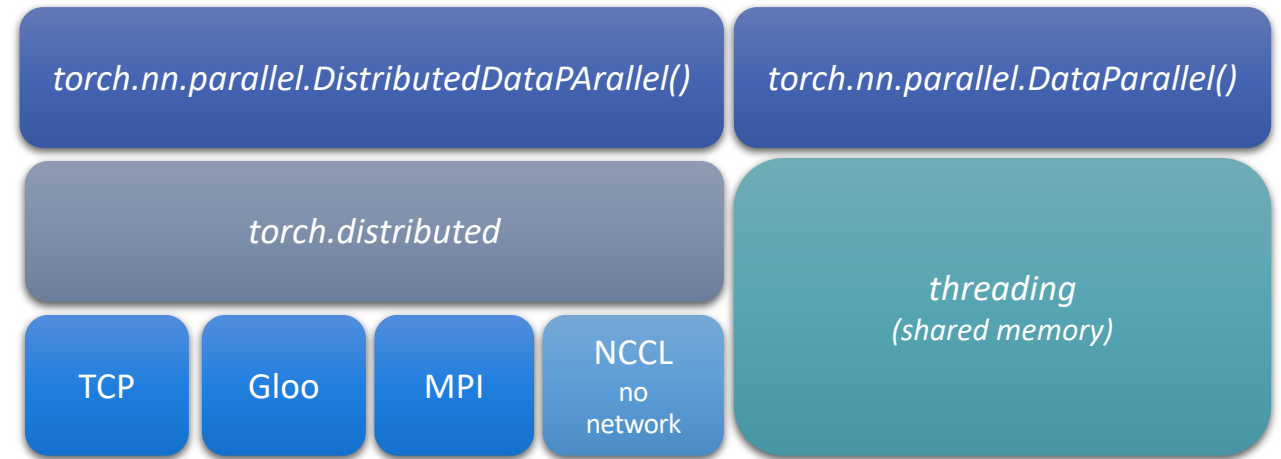
- **Points of synchronization**
(barriers or collectives):

- Constructor
- Each module forward operation
- Each module backward operation



PyTorch – *Data Parallel Support*

- ***torch.nn.parallel.DistributedDataParallel()***
builds on *torch.distributed* to provide synchronous distributed training as a wrapper around any PyTorch model
- ***torch.distributed***: provides an MPI-like interface for exchanging tensor data across multi-machine networks. It supports a few different backends and initialization methods
 - **Backends**: TCP, gloo, MPI, NCCL
- ***torch.nn.parallel.DataParallel()*** is not distributed: valid in a single node only
 - Based on ***threading*** module



PyTorch - Distributed Deep Learning

- **Simple Approach:** use ***torch.nn.parallel.DistributedDataParallel()*** class
 - Only available with Gloo and NCCL backends
 - Need to use parallelism at layer level (all-reduce at each layer)
 - Cannot use implemented algorithms
- **Advanced Approach:** use ***torch.distributed*** package
 - Write your own distributed algorithm with **MPI** using collectives and point-to-point communication

PyTorch – *torch.distributed* package support

- PyTorch offer support for several backends: tcp, gloo, mpi, nccl (only local)
- **MPI** has the more flexible support for collectives with gpu

Backend	tcp		gloo		mpi		nccl	
Device	CPU	GPU	CPU	GPU	CPU	GPU	CPU	GPU
send	✓	X	X	X	✓	?	X	X
recv	✓	X	X	X	✓	?	X	X
broadcast	✓	X	✓	✓	✓	?	X	✓
all_reduce	✓	X	✓	✓	✓	?	X	✓
reduce	✓	X	X	X	✓	?	X	✓
all_gather	✓	X	X	X	✓	?	X	✓
gather	✓	X	X	X	✓	?	X	✓
scatter	✓	X	X	X	✓	?	X	✓
barrier	✓	X	✓	✓	✓	?	X	X

PyTorch – *torch.distributed* Launch

- Launch the same as any other MPI program:

```
mpirun -n <num_ranks> python ./application.py
```

- Each MPI rank is a separate process executing a Python interpreter
 - Ranks can be created on different nodes depending on job configuration (slurm)
 - MPI will be initialized inside the python application
 - Each MPI rank will communicate using Python MPI primitives provided by *torch.distributed*

PyTorch – *torch.distributed* Hello World

- Initialization: ranks need to be initialized similar to MPI, each rank needs to know its rank number

```
import os
import torch
import torch.distributed as dist

# Initialize MPI.
dist.init_process_group(backend='mpi')
rank = dist.get_rank()
wsize = dist.get_world_size()

# Print Hello World
print('Hello from process {} (out of {})\n'.format(rank, wsize))
```

PyTorch – *torch.distributed* Point-to-Point (1)

- Blocking Point-to-point communication primitives:
 - *dist.send()*
 - *dist.recv()*
- Support for *tensor* type:
 - serialization done automatically
- Notice that process 1 needs to allocate memory in order to store the data that it will receive

```
"""Blocking point-to-point communication."""

def run(rank, size):
    tensor = torch.zeros(1)
    if rank == 0:
        tensor += 1
        # Send the tensor to process 1
        dist.send(tensor=tensor, dst=1)
    else:
        # Receive tensor from process 0
        dist.recv(tensor=tensor, src=0)
    print('Rank ', rank, ' has data ', tensor[0])
```

PyTorch – *torch.distributed* Point-to-Point (2)

- Non Blocking Point-to-point communication primitives:
 - `dist.isend()`
 - `dist.irecv()`
- Support for *tensor* type:
 - serialization done automatically
- Sender: cannot modify tensor until the `req.wait()` is completed
- Writing to tensor after `dist.isend()` will result in undefined behaviour.
- Reading from tensor after `dist.irecv()` will result in undefined behaviour

```
"""Non-blocking point-to-point communication."""
```

```
def run(rank, size):  
    tensor = torch.zeros(1)  
    req = None  
    if rank == 0:  
        tensor += 1  
        # Send the tensor to process 1  
        req = dist.isend(tensor=tensor, dst=1)  
        print('Rank 0 started sending')  
    else:  
        # Receive tensor from process 0  
        req = dist.irecv(tensor=tensor, src=0)  
        print('Rank 1 started receiving')  
    req.wait()  
    print('Rank ', rank, ' has data ', tensor[0])
```

PyTorch – *torch.distributed* All-Reduce

- All-Reduce Operation
- Using the *dist.reduce_op.SUM* reduce operation
- Available reduce operations:
 - *dist.reduce_op.SUM*
 - *dist.reduce_op.PRODUCT*
 - *dist.reduce_op.MAX*
 - *dist.reduce_op.MIN*

```
""" All-Reduce example."""  
def run(rank, size):  
    """ Simple point-to-point communication. """  
    group = dist.new_group([0, 1])  
    tensor = torch.ones(1)  
    dist.all_reduce(tensor, op=dist.reduce_op.SUM,  
group=group)  
    print('Rank ', rank, ' has data ', tensor[0])
```

PyTorch – *torch.distributed* Available Collectives

- **dist.broadcast(tensor, src, group):**
 - Copies tensor from src to all other processes.
- **dist.reduce(tensor, dst, op, group):**
 - Applies op to all tensor and stores the result in dst.
- **dist.all_reduce(tensor, op, group):**
 - Same as reduce, but the result is stored in all processes.
- **dist.scatter(tensor, src, scatter_list, group):**
 - Copies the ith tensor scatter_list[i] to the ith process.
- **dist.gather(tensor, dst, gather_list, group):**
 - Copies tensor from all processes in dst.
- **dist.all_gather(tensor_list, tensor, group):**
 - Copies tensor from all processes to tensor_list, on all processes.

Distributed SGD with *torch.distributed*

- `run()` implements the default forward-backward training
- At each step average gradients among ranks with
 - *`average_gradients(model)`*

```
""" Gradient averaging. """
```

```
def average_gradients(model):  
    size = float(dist.get_world_size())  
    for param in model.parameters():  
        dist.all_reduce(param.grad.data,  
                        op=dist.reduce_op.SUM)  
        param.grad.data /= size
```

```
""" Distributed Synchronous SGD Example """
```

```
def run(rank, size):  
    torch.manual_seed(1234)  
    train_set, bsz = partition_dataset()  
    model = Net()  
    optimizer = optim.SGD(model.parameters(),  
                           lr=0.01, momentum=0.5)  
  
    num_batches = ceil(len(train_set.dataset) / float(bsz))  
    for epoch in range(10):  
        epoch_loss = 0.0  
        for data, target in train_set:  
            optimizer.zero_grad()  
            output = model(data)  
            loss = F.nll_loss(output, target)  
            epoch_loss += loss.data[0]  
            loss.backward()  
            average_gradients(model)  
            optimizer.step()  
        print('Rank ', dist.get_rank(), ', epoch ',  
              epoch, ': ', epoch_loss / num_batches)
```

Ring All-Reduce with *torch.distributed*

- *allreduce()* implements a custom All-Reduce algorithm based on a **ring** communication pattern:
 - Each rank sends his data to the next rank
 - The last one sends it back to the first
- Ring approach:
 - very efficient for a small number of ranks
 - not scalable for large number of ranks

```
""" Implementation of a ring-reduce with addition. """
def allreduce(send, recv):
    rank = dist.get_rank()
    size = dist.get_world_size()
    send_buff = th.zeros(send.size())
    recv_buff = th.zeros(send.size())
    accum = th.zeros(send.size())
    accum[:] = send[:]

    left = ((rank - 1) + size) % size
    right = (rank + 1) % size

    for i in range(size - 1):
        if i % 2 == 0:
            # Send send_buff
            send_req = dist.isend(send_buff, right)
            dist.recv(recv_buff, left)
            accum[:] += recv[:]
        else:
            # Send recv_buff
            send_req = dist.isend(recv_buff, right)
            dist.recv(send_buff, left)
            accum[:] += send[:]
        send_req.wait()
    recv[:] = accum[:]
```

Lesson Key Points

- Distributed DL Algorithms
 - Synchronous/Asynchronous
 - Centralized/Decentralized
 - Passing gradient/parameters
- PyTorch Distributed DL:
 - Basic: *torch.nn.parallel.DistributedDataParallel()*
 - Advanced: *torch.distributed*