ECE-GY 9143
Introduction to High Performance Machine Learning

Lecture 8 03/14/2024

Parijat Dube
Kaoutar El Maghraoui

Distributed Deep Learning

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

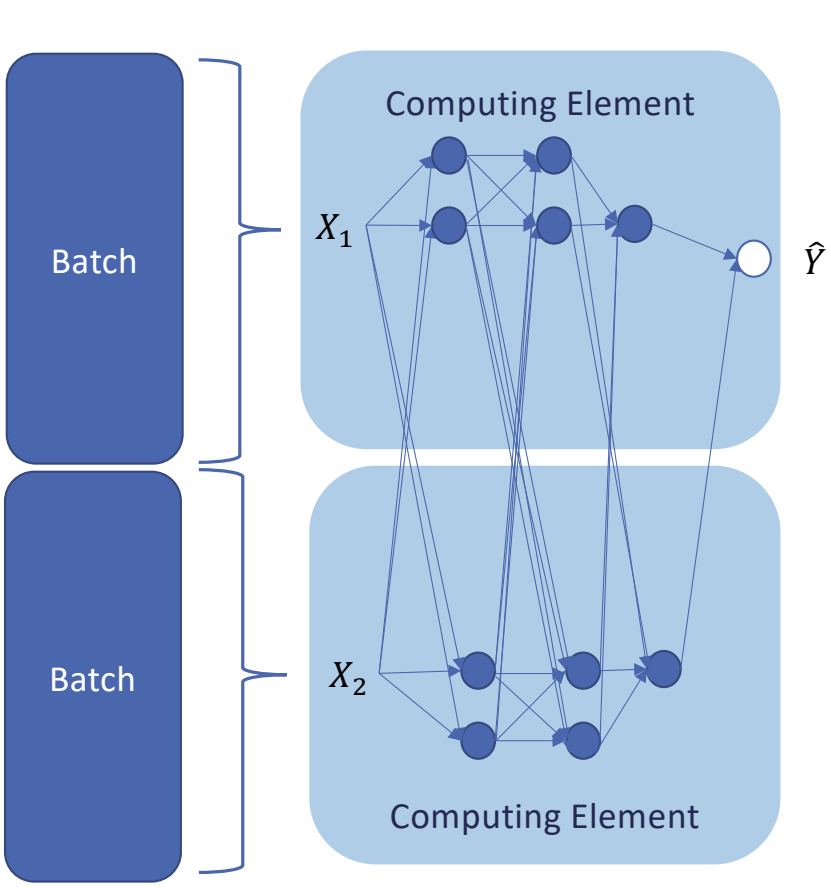
DL Parallelism Approaches

- **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

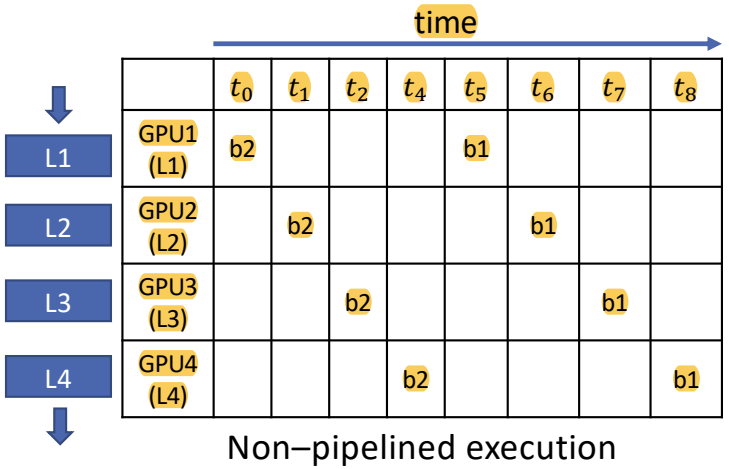


DL Pipelining

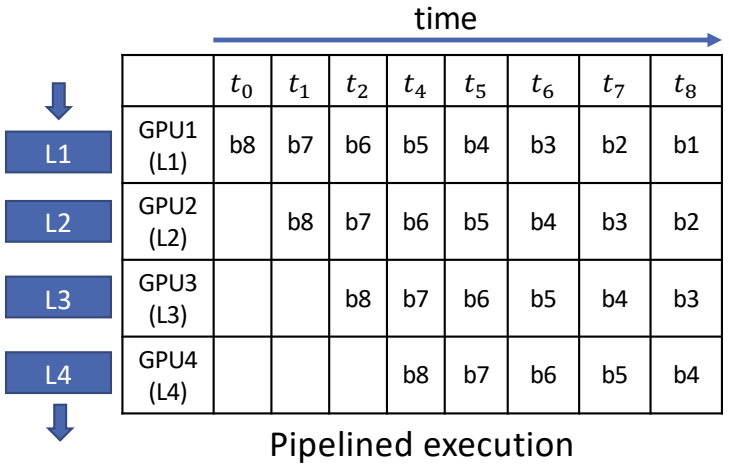
- Pipelining approach:
  - Split layers among compute engines
  - Each minibatch *b (or sample s)* goes from one compute engine to the next one: *no need to wait for next one to exit the pipeline*
- Is a form of **Model Parallelism**
- Pipelining performance
  - Ideal pipelining speedup *(number of pipeline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

$$S\_time(stage\ time) = \frac{time\ without\ pipeline}{number\ of\ pipeline\ stages}$$

  - 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**)



Non–pipelined execution

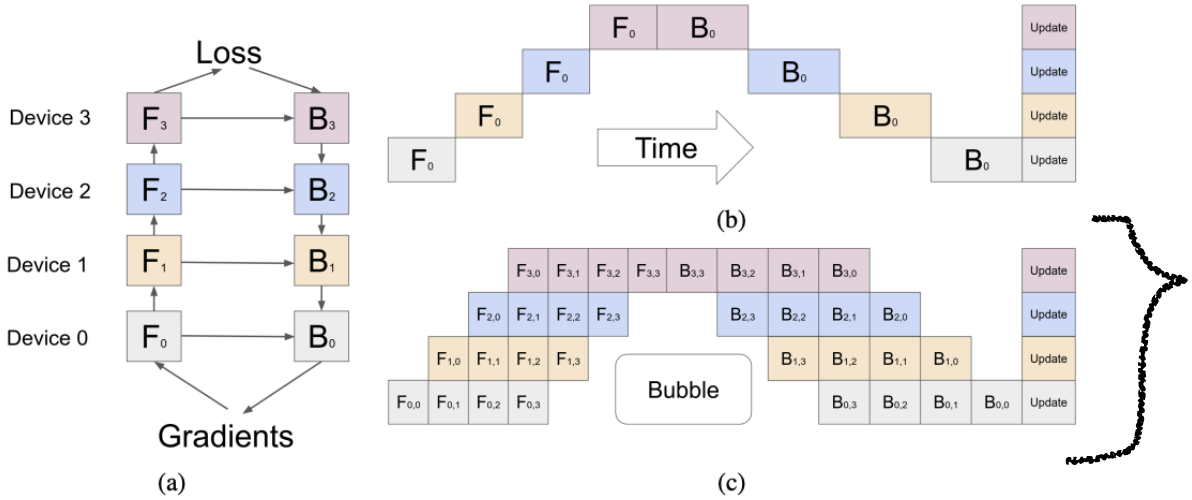


Pipelined execution

Pipeline Parallelism

- Main Steps:
  1. Split a mini-batch into microbatches, which are fed into the training pipeline one-by-one
     - A *microbatch* is a smaller subset of a given training mini-batch.
  2. Process the microbatches according to the pipeline execution schedule across all the devices in the pipeline

GPipe Pipelining

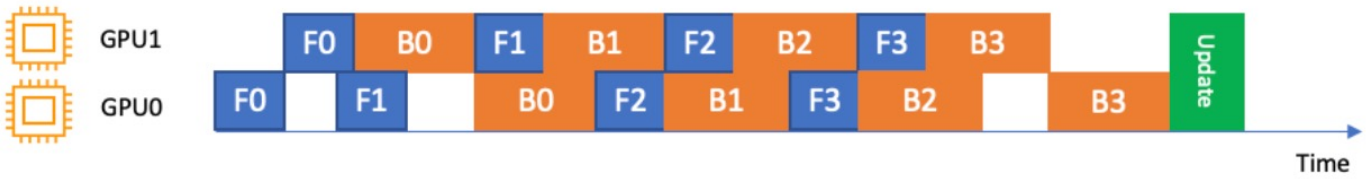


- Gpipe: a pipeline parallelism open-source library that allows scaling any network that can be expressed as a sequence of layers.
- Split global batch into multiple micro-batches and injects them into the pipeline concurrently
- Not memory-friendly and will not scale well with large batch. The activations produced by forward tasks have to be kept for all micro-batches until corresponding backward tasks start, thus leads to the memory demand to be proportional (O(M)) to the number of concurrently scheduled micro-batches (M).

GPipe: Easy Scaling with Micro-Batch Pipeline Parallelism

Interleaved Pipeline

an attempt to reduce the number of bubbles



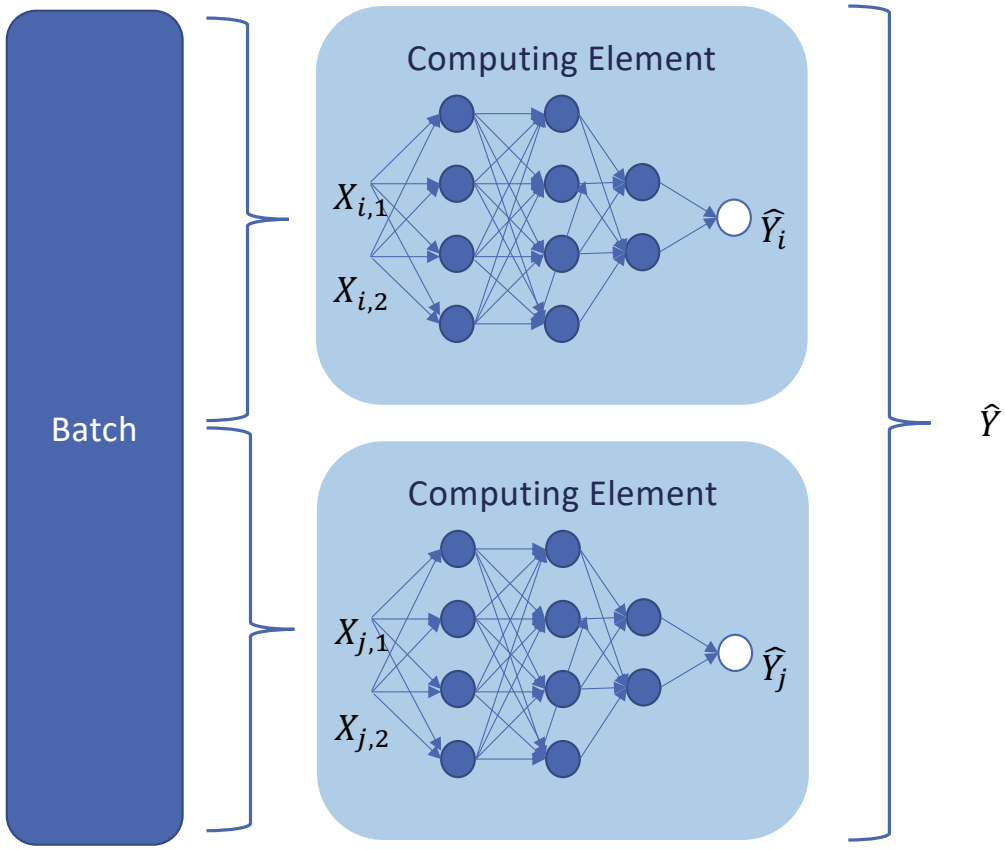
- Backward execution of the microbatches is prioritized whenever possible.
- Allows quicker release of the memory used for activations, using memory more efficiently
- At steady-state, each device alternates between running forward and backward passes
  - Backward pass of one microbatch may run before the forward pass of another microbatch finishes

https://docs.aws.amazon.com/sagemaker/latest/dg/model-parallel-core-features-pipieline-parallelism.html

ML Data Parallelism

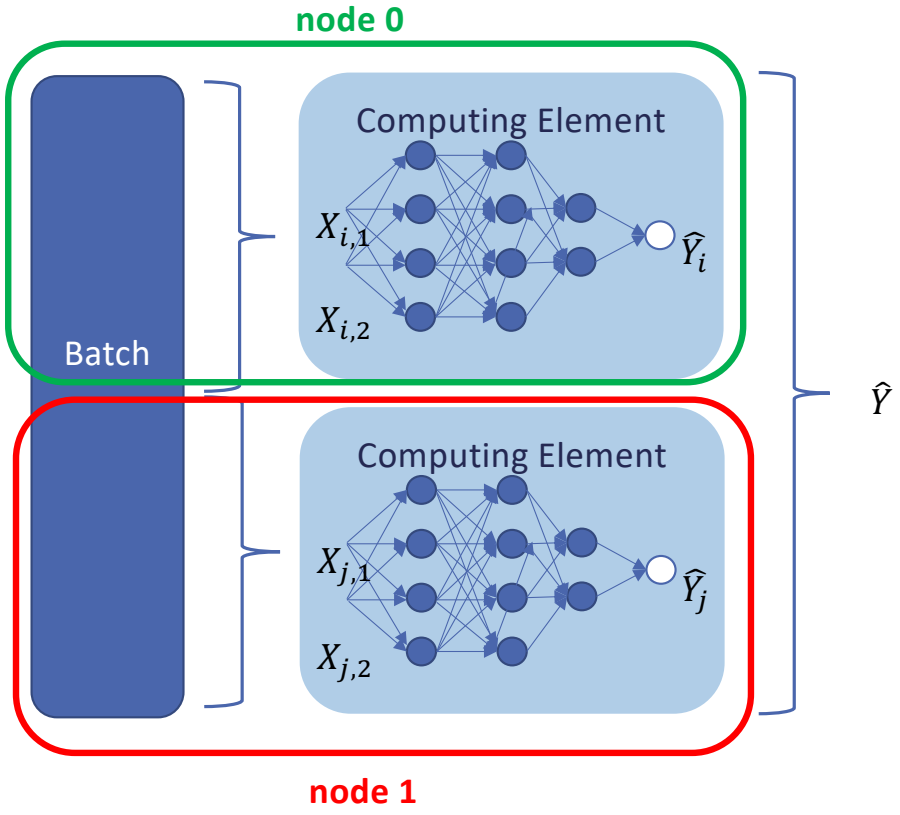
- **Data Parallelism:**
  1. replicate model on each computing element (CPU core, GPU thread, or Cluster node)
  2. Each computing element gets a portion of the batch
  3. Each computing element trains/infers in parallel on the model
  4. Gather the results at the end.

- **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

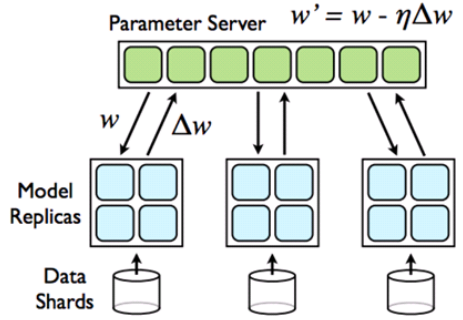
- Distributed data parallel SGD is basically the classic minibatch SGD with lines 3 and 7 modified to read/write weights from/to a parameter store which may be centralized or decentralized
- This incurs an overhead on the overall system

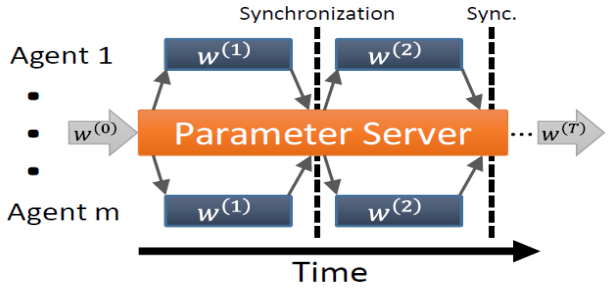
Algorithm 2 Minibatch Stochastic Gradient Descent with Backpropagation

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





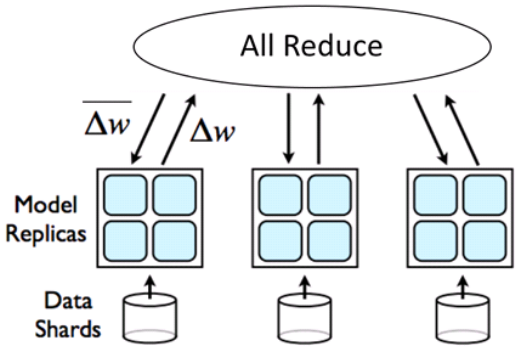
Synchronous Centralized Distributed SGD

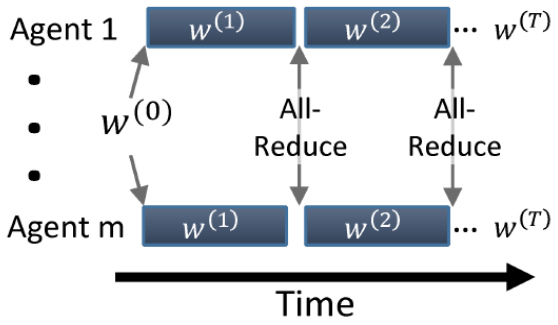
from: https://arxiv.org/abs/1802.09941

No parameter server

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

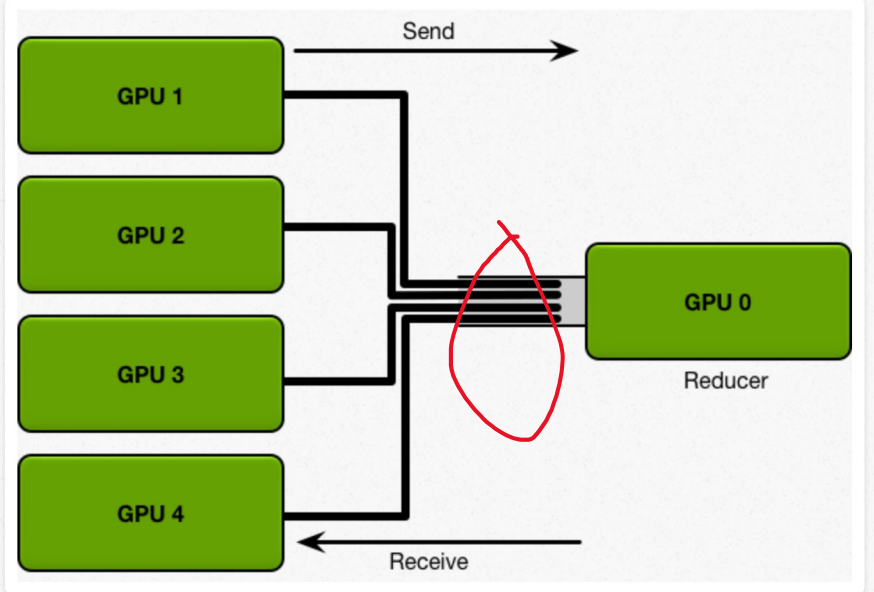
Reduction over Gradients

- To synchronize gradients of N learners, a reduction operation needs to be performed

$$\sum_{j=1}^{N} \Delta w_j$$

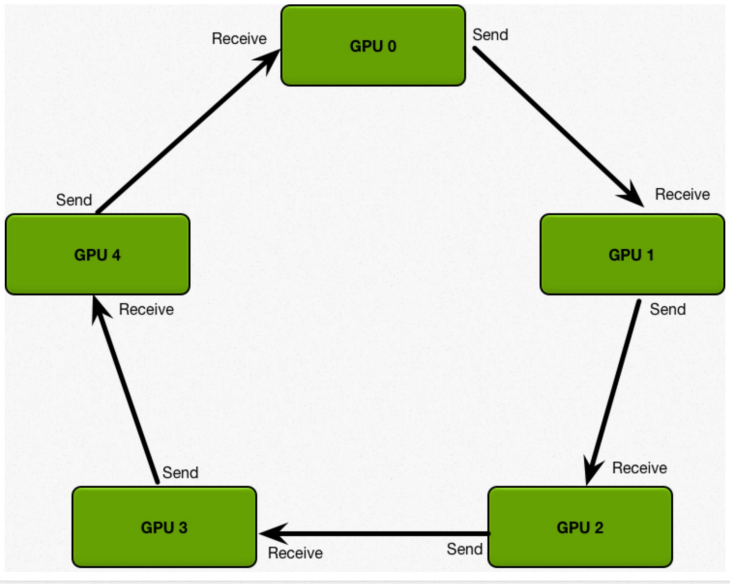
Reduction Topologies

Parameter server: single reducer



SUM (Reduce operation) performed at PS

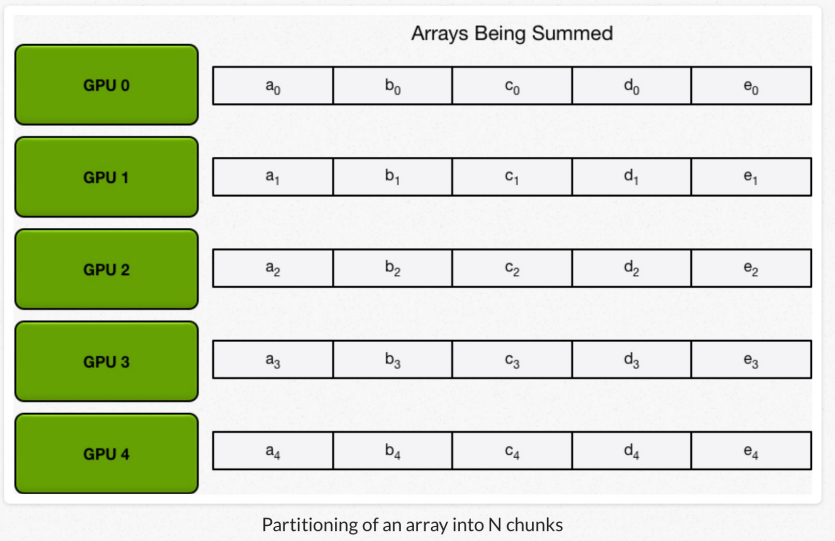
GPUs arranged in a logical Ring (aka bucket) : *all are reducers*



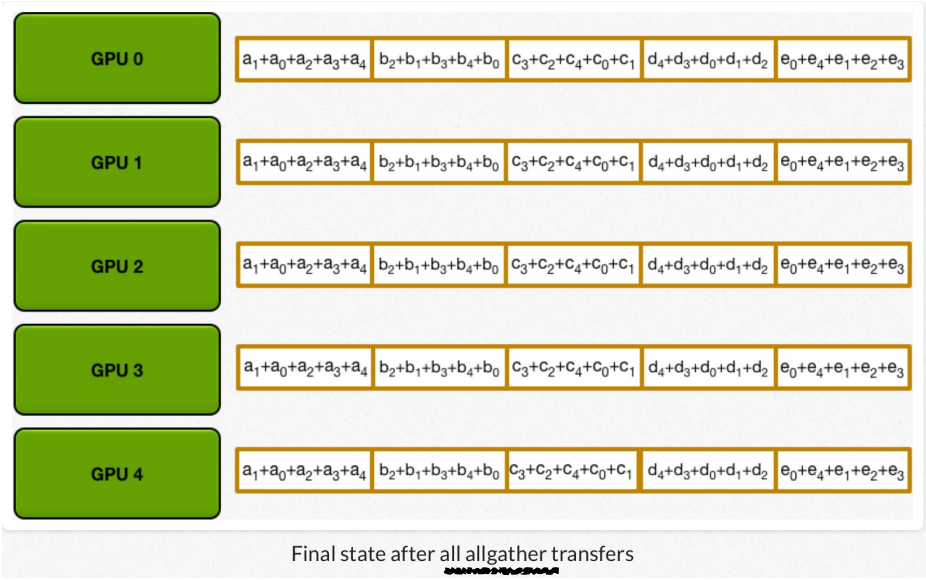
SUM (Reduce operation) performed at all nodes

- Each node has a left neighbor and a right neighbor
- Node only sends data to its right neighbor, and only receives data from its left neighbor

Example Problem



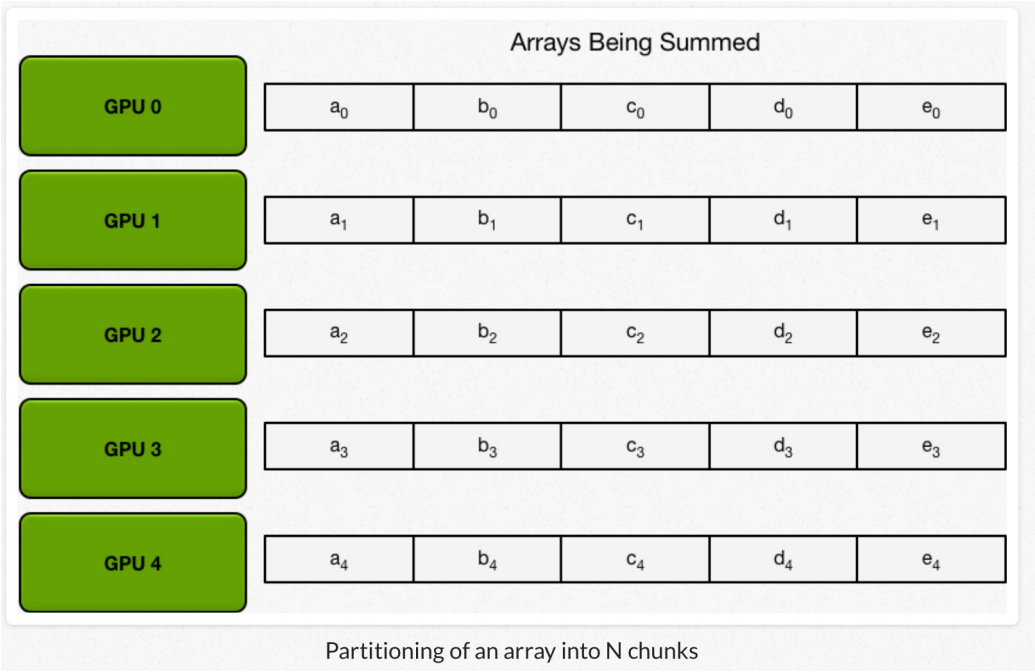
Partitioning of an array into N chunks



Final state after all allgather transfers

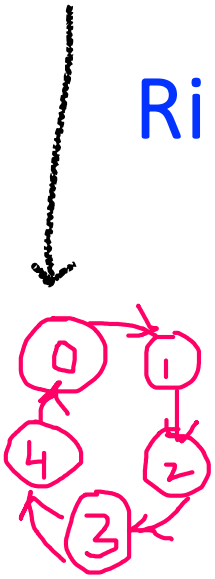
Ring All-Reduce

- Two step algorithm:
  - *Scatter-reduce*
    - GPUs exchange data such that every GPU ends up **with a chunk** of the final result
  - *Allgather*
    - GPUs exchange chunks from scatter-reduce such that all GPUs end up with the complete final result.

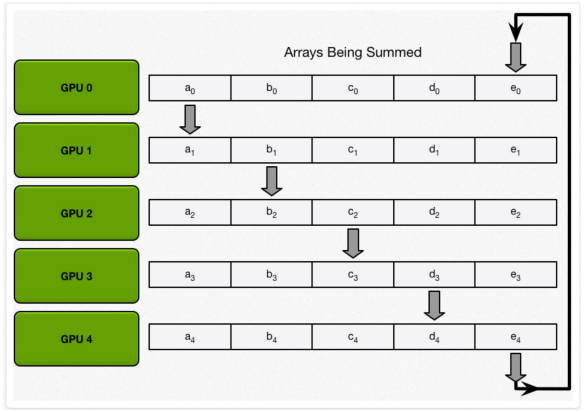


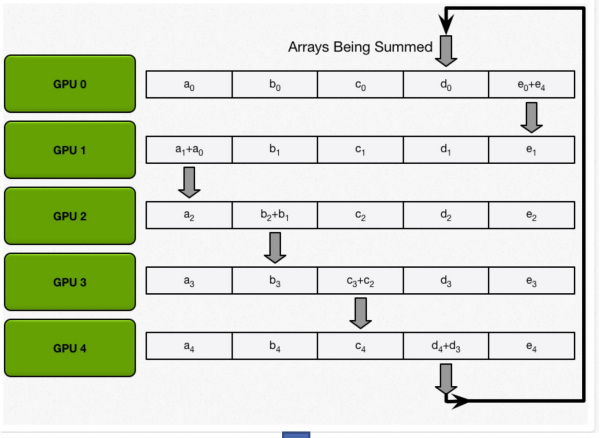
Partitioning of an array into N chunks

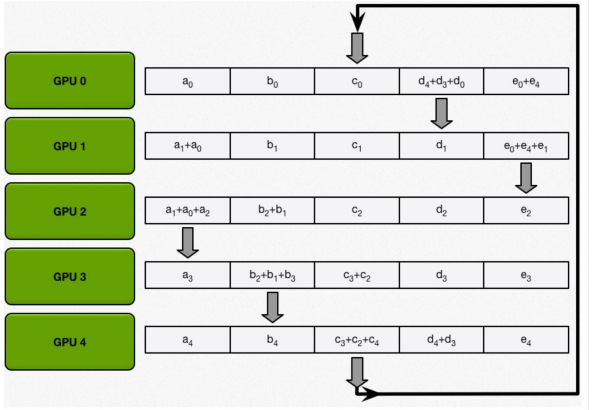
GPUs are arranged in a cyclic fashion

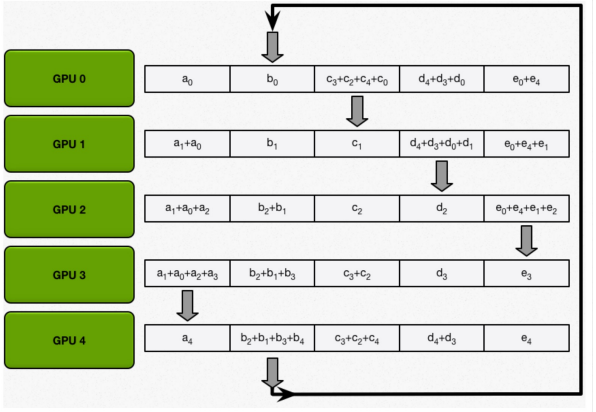


Ring All-Reduce: Scatter-Reduce Step

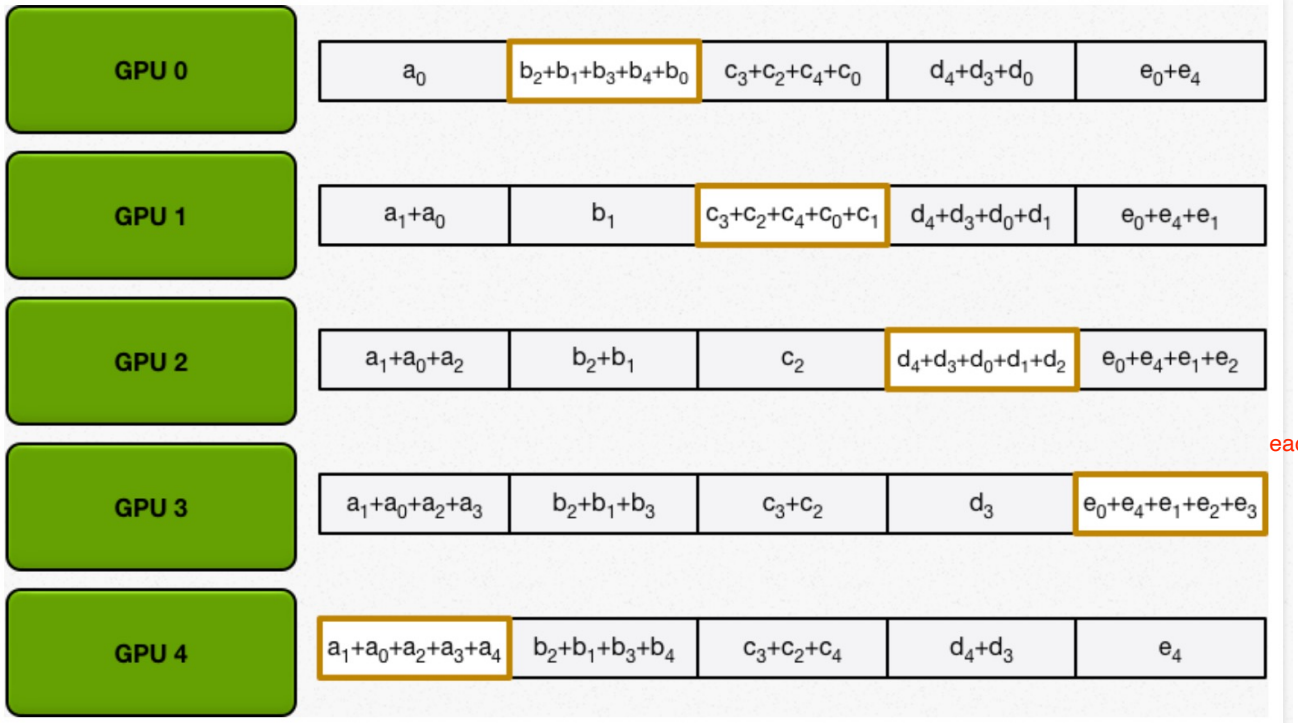








Ring All-Reduce: End of Scatter-Reduce Step



Final state after all scatter-reduce transfers

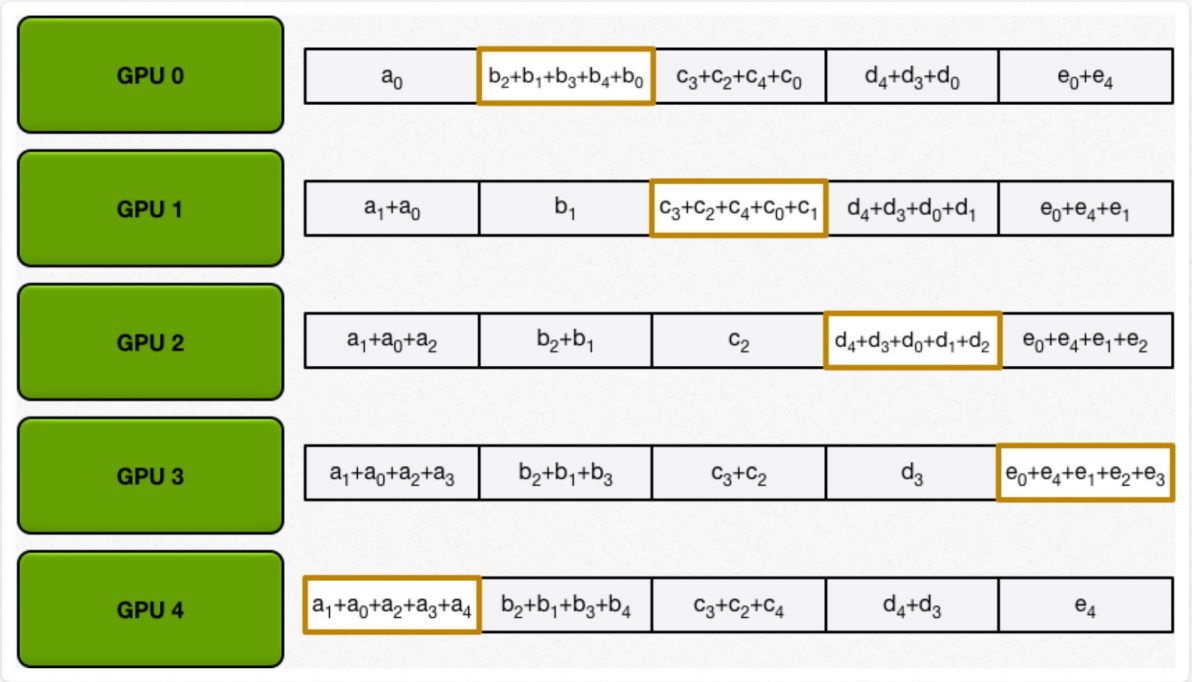
How many iterations in scatter-reduce step with N GPUs ?

N - 1

for N number of GPUs

each node has a chunk of final result. now we need allgather step

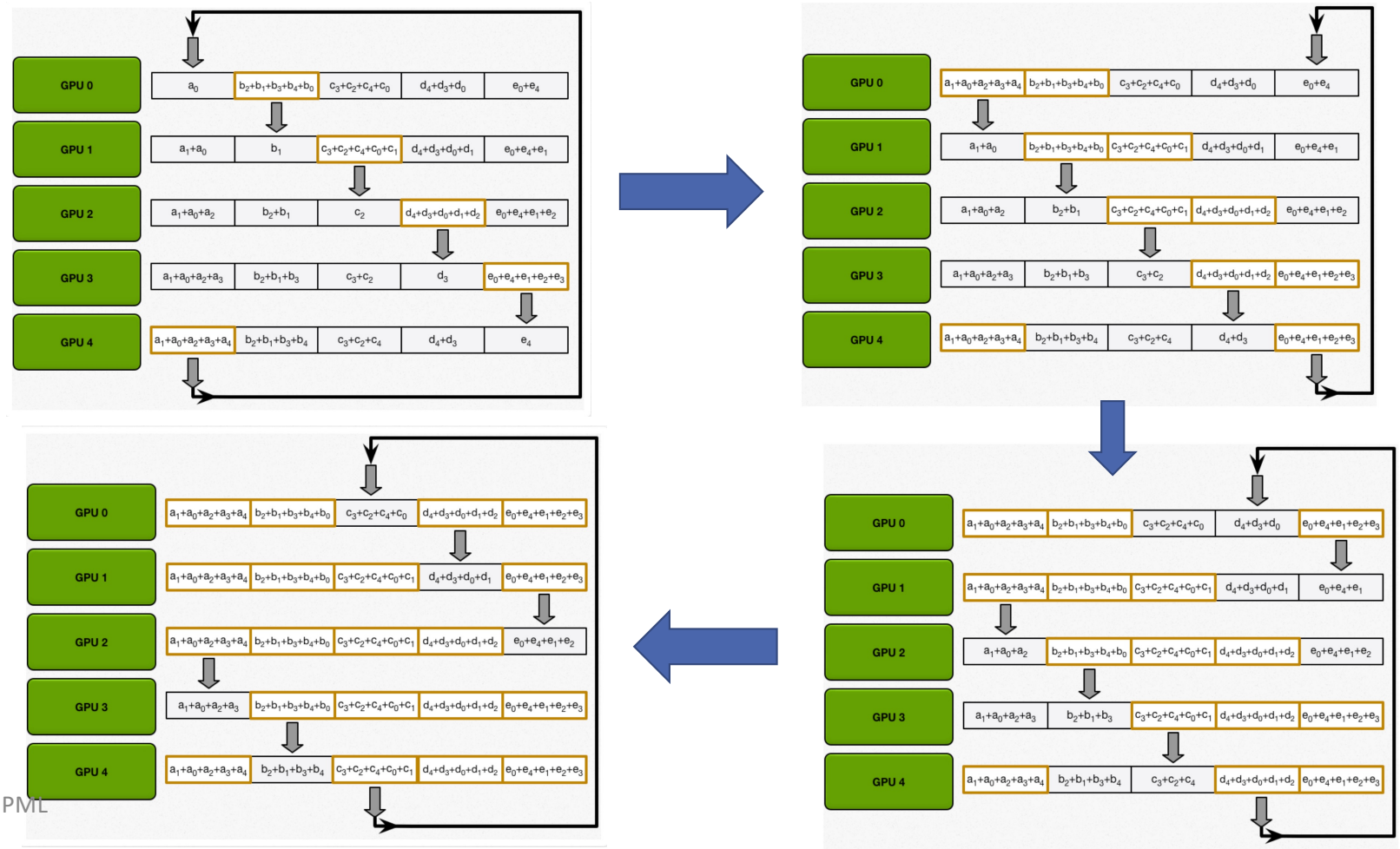
Ring All-Reduce: What to do next ?



Final state after all scatter-reduce transfers

GPU	Send	Receive
0	Chunk 1	Chunk 0
1	Chunk 2	Chunk 1
2	Chunk 3	Chunk 2
3	Chunk 4	Chunk 3
4	Chunk 0	Chunk 4

Ring All-Reduce: AllGather Step



Ring All-Reduce: End of AllGather Step

Final state after all allgather transfers

How many iterations in allgather step with N GPUs ?

N - 1

Parameter Server (PS) vs Ring All-Reduce: Communication Cost

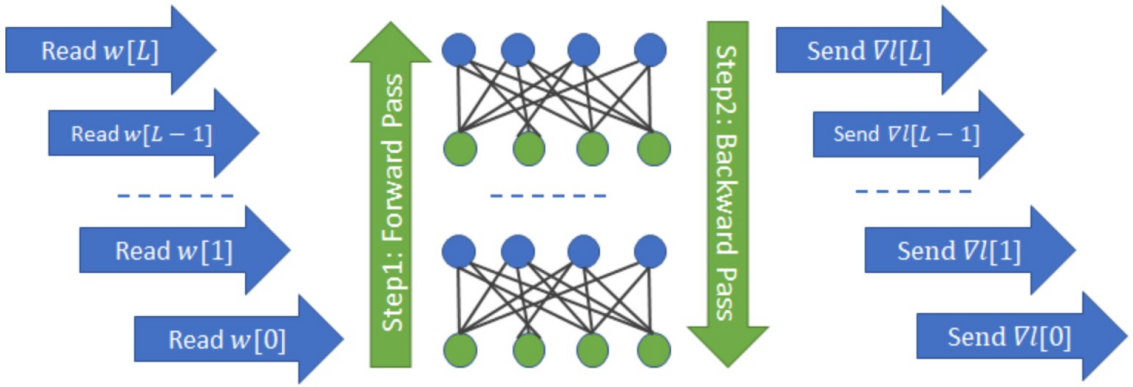
without parameter server

- P: number of processes N: total number of model parameters
- PS (centralized reduce)
  - Amount of data sent to PS by (P-1) learner processes: N(P-1)
  - After reduce, PS sends back updated parameters to each learner
  - Amount of data sent by PS to learners: N(P-1)
  - Total communication cost at PS process is proportional to 2N(P-1)
- Ring All-Reduce (decentralized reduce)
  - Scatter-reduce: Each process sends N/P amount of data to (P-1) learners
    - Total amount sent (per process): N(P-1)/P
  - AllGather: Each process again sends N/P amount of data to (P-1) learners
  - Total communication cost per process is 2N(P-1)/P
- PS communication cost is proportional to P whereas ring all-reduce cost is practically independent of P for large P (ratio (P-1)/P tends to 1 for large P)
- Which scheme is more bandwidth efficient ?
- Note that both PS and Ring all-reduce involve synchronous parameter updates

not bandwidth efficient

why?

All-Reduce applied to Deep Learning



- Backpropagation computes gradients starting from the output layer and moving towards in the input layer
- Gradients for output layers are available earlier than inner layers
- Start all reduce on the output layer parameters while other gradients are being computed
- Overlay of communication and local compute; Pipelining

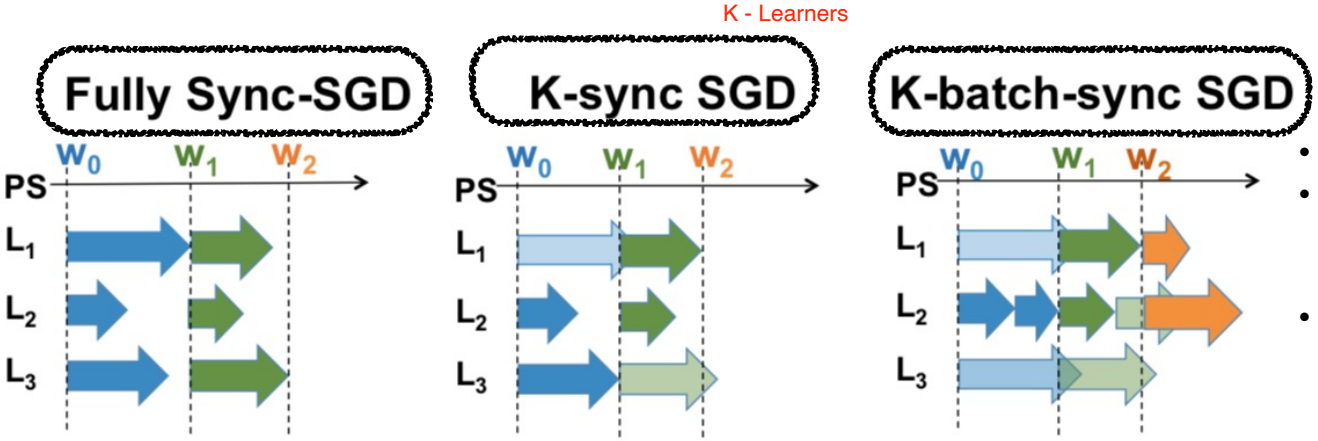
Scaling using compute-communication overlap in all-reduce

- https://andrew.gibiansky.com/blog/machine-learning/baidu-allreduce/

Synchronous SGD and the Straggler problem

- PS needs to wait for updated gradients from all the learners before calculating the model parameters
- Even though size of mini-batch processed by each learner is same, updates from different learners may be available at different times at the PS
  - Randomness in compute time at learners
  - Randomness in communication time between learners and PS
- Waiting for slow and straggling learners diminishes the speed-up offered by parallelizing the training

Synchronous SGD Variants



- P: total number of learners
- K: number of learners/mini-batches the PS waits for before updating parameters
- Lightly shaded arrows indicate straggling gradient computations that are canceled.

- K-sync SGD: PS waits for gradients from K *learners* before updating parameters; the remaining learners are canceled
- When K = P , K-sync SGD is same as Fully Sync-SGD
- K-batch sync: PS waits for gradients from K *mini-batches* before updating parameters; **the remaining (unfinished) learners are canceled**
  - Irrespective of which learner the gradients come from
  - Wherever any learner finishes, it pushes its gradient to the PS, fetches current parameter at PS and starts computing gradient on the next mini-batch based on the same local value of the parameters
- Runtime per iteration reduces with K-batch sync; error convergence is same as K-sync

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

Some history of the Distributed asynchronous SGD

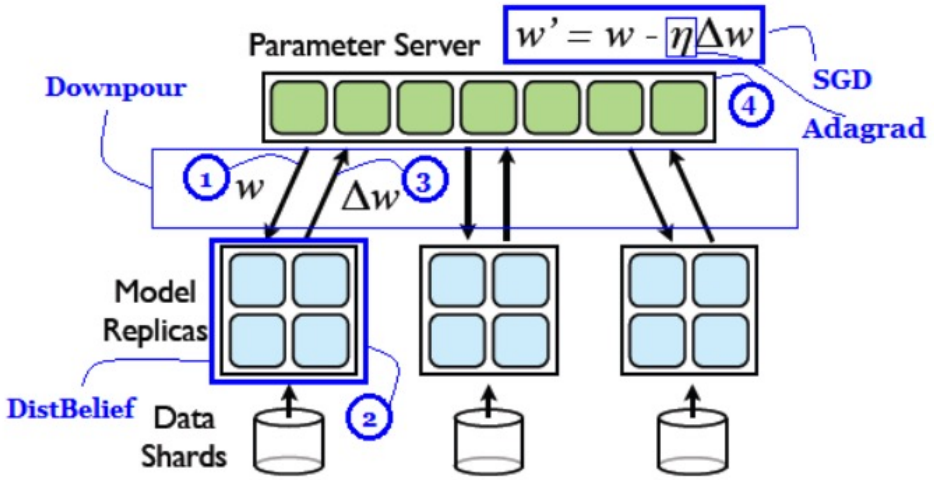
- Distributed asynchronous SGD method was introduced by [Tsitsiklis1986].
- [Zinkevich2010] proposed the first synchronous parallel SGD algorithm by averaging the individual parameter after each local update

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

- [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.

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($\alpha, n_{fetch}, n_{push}$)

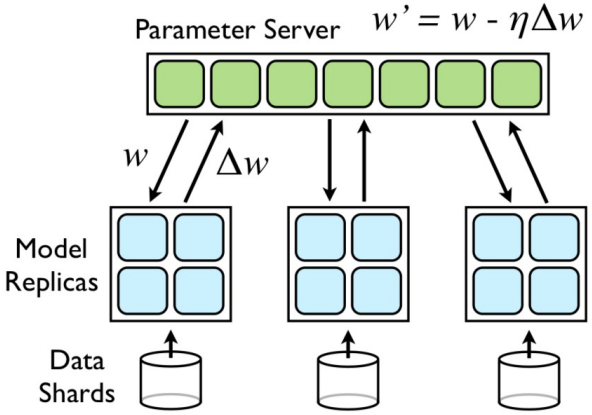
procedure STARTASYNCHRONOUSLYFETCHINGPARAMETERS($parameters$)
 $parameters \leftarrow$ GETPARAMETERSFROMPARAMSERVER()

procedure STARTASYNCHRONOUSLYPUSHINGGRADIENTS($accruedgradients$)
 SENDGRADIENTSTOPARAMSERVER($accruedgradients$)
 $accruedgradients \leftarrow 0$

main
 global $parameters, accruedgradients$
 $step \leftarrow 0$
 $accruedgradients \leftarrow 0$
 while $true$
 do
  if $(step \bmod n_{fetch}) == 0$ (1)
   then STARTASYNCHRONOUSLYFETCHINGPARAMETERS($parameters$)
  $data \leftarrow$ GETNEXTMINIBATCH()
  $gradient \leftarrow$ COMPUTEGRADIENT($parameters, data$) (2)
  $accruedgradients \leftarrow accruedgradients + gradient$
  $parameters \leftarrow parameters - \alpha * gradient$ (3)
  if $(step \bmod n_{push}) == 0$
   then STARTASYNCHRONOUSLYPUSHINGGRADIENTS($accruedgradients$)
  $step \leftarrow step + 1$

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

Downpour SGD



Downpour SGD (from Google, 2012)
Each model replica is a DistBelief model, Google's proprietory ML framework

- Model parallelism
- Automatically parallelizes computation in each machine using all available cores

Model and Data parallelism
Parameter server is sharded

- Different PS shards update their parameters **asynchronously**, they do not need to communicate among themselves. **Why ?**
- Different model replicas (Distbelief models) run independently. **Why no direct communication ?**
- Model replicas are permitted to fetch parameters and push gradients in separate threads. **What is the value of this ?**
- Tolerates variances in the processing speed of different model replicas
- Tolerates failure of model replicas; More robust to machine failures than synchronous SGD. **Correct ?**

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”

Downpour SGD: Sources of Stochasticity

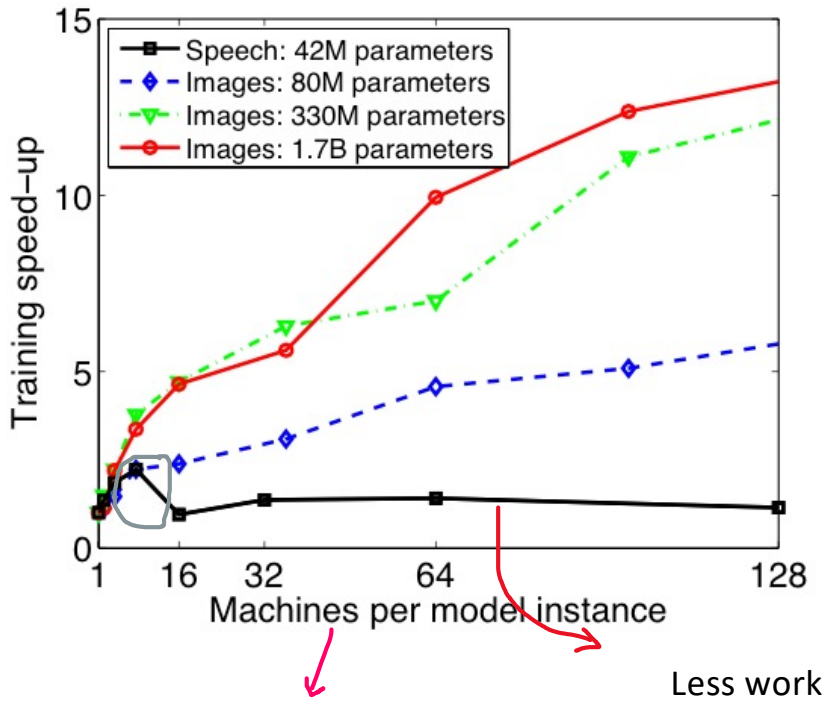
- Model replicas will most likely be using "stale" parameters when calculating gradients locally
- No guarantee that at any given moment the parameters on each shard of the parameter server have undergone the same number of updates
  - Order of updates may be different at different PS shards
- Adagrad learning rate improves robustness of Downpour SGD

$$\eta_{i,k} = \frac{\gamma}{\sqrt{\sum_{j=1}^{k} \Delta w_{i,j}^2}}$$

$\eta_{i,k}$: learning rate of the ith parameter at iteration k
$\Delta w_{i,k}$ is its gradient
γ is the constant scaling factor

- Adagrad implemented locally within each parameter server shard

Scaling with Model Parallelism



Average training speed-up: the ratio of the time taken per iteration using only a single machine to the time taken using N partitions (machines)

Benefit diminishes when adding more machines:

- Less work per machine
- Increased communication between machines

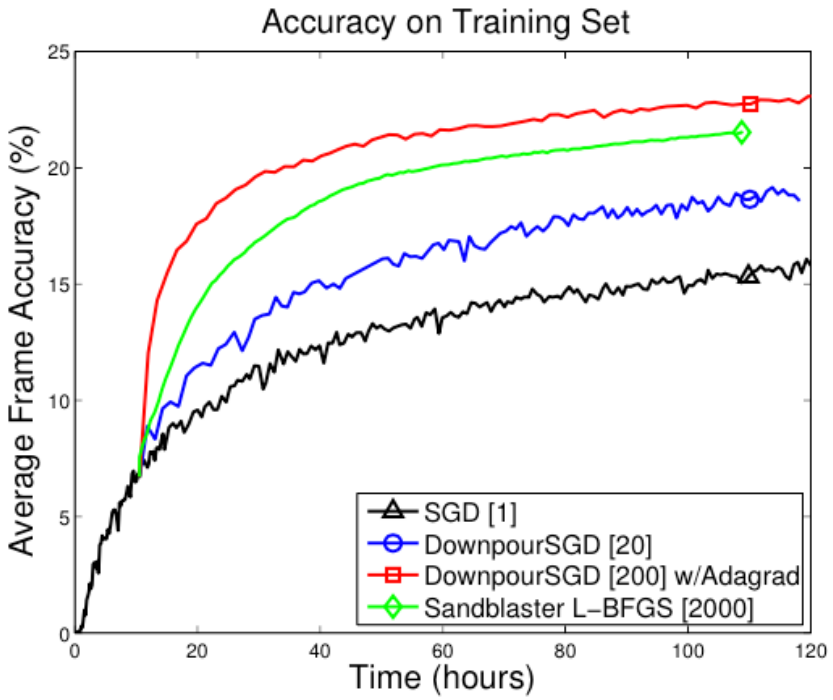
Speed-up with model parallelism

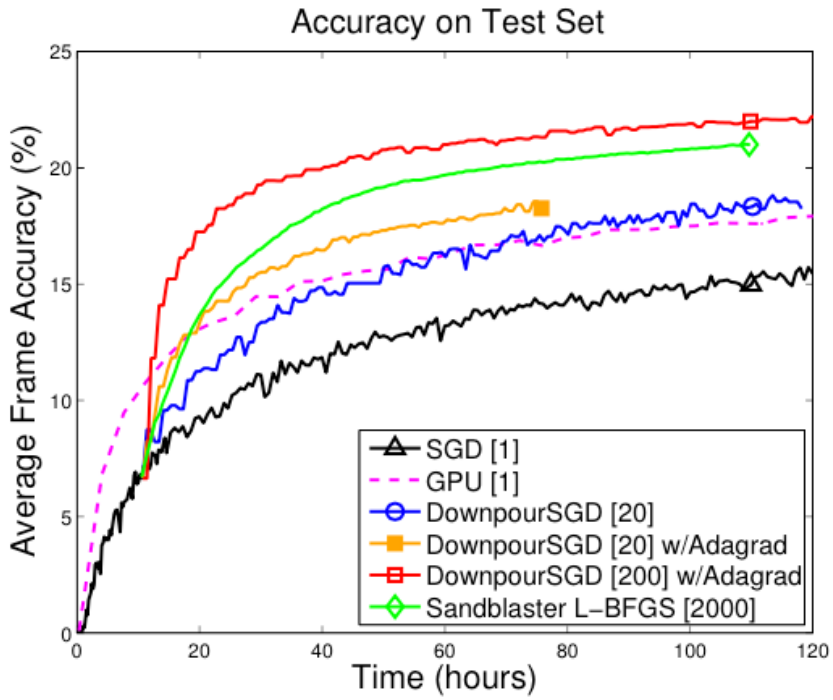
- Fully connected speech model with 42M parameters: 2.2x with 8 machines
- Locally connected image model with 1.7B parameters: 12x with 81 machines

Less work per machine, network overhead dominates

Same as number of partitions used in a single machine instance

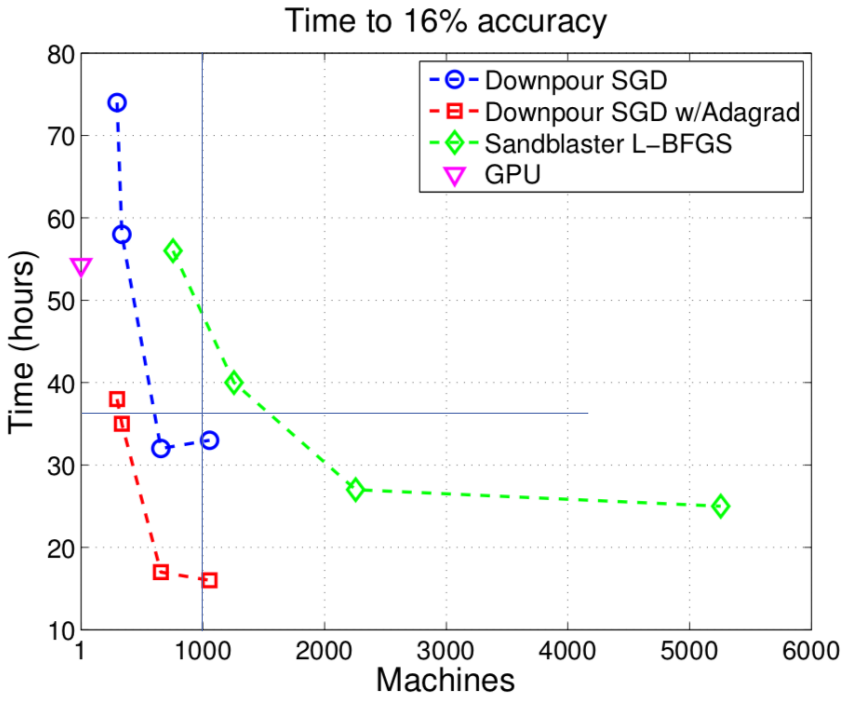
Scaling with Downpour SGD





Each model replica is partitioned on 8 machines

Runtime-Resource Tradeoff in Downpour SGD



Points closer to the origin are preferable in that they take less time while using fewer resources.
Which gives the best tradeoff ?

staleness = 2. (model is updated twice until this time)

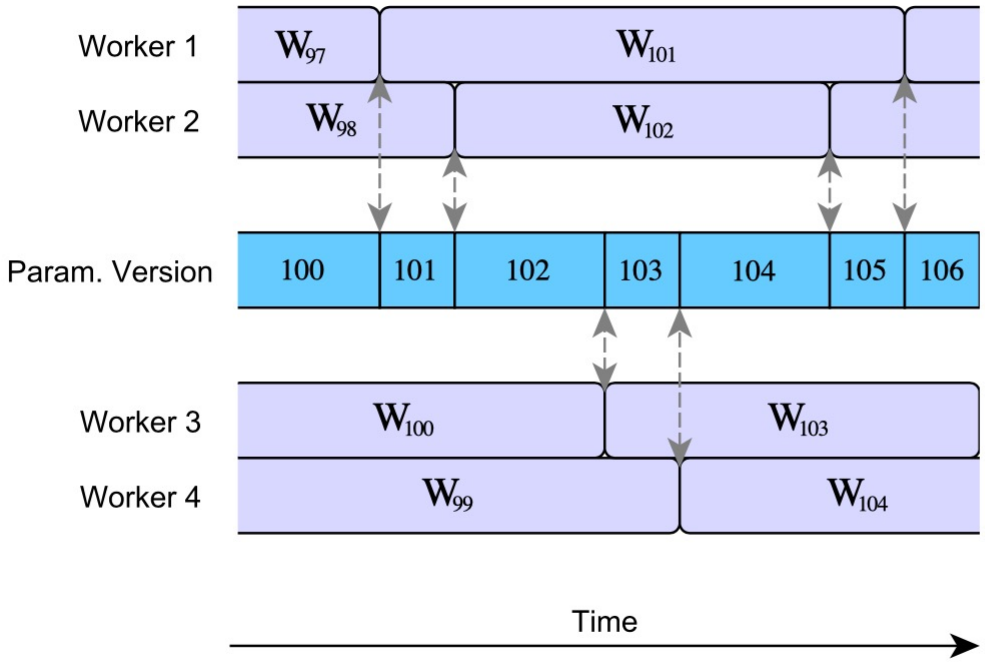
Asynchronous SGD and Variants

- K-async SGD: PS waits for gradients from K *learners* before updating parameters but the remaining learners are not canceled; each learner may be giving a gradient calculated at stale version of the parameters
- When K = 1 , K-async SGD is same as Async-SGD
- K-batch async: PS waits for gradients from K *mini-batches* before updating parameters; **the remaining learners are not canceled**
  - Whenever any learner finishes, it pushes its gradient to the PS, fetches current parameter at PS and starts computing gradient on the next mini-batch based on the current value of the PS
- Runtime per iteration reduces with K-batch async; error convergence is same as K-async

Asynchronous SGD and Stale Gradients

- PS updates happen without waiting for all learners
- Weights that a learner uses to evaluate gradients may be old values of the parameters at PS
  - Parameter server asynchronously updates weights
  - By the time learner gets back to the PS to submit gradient, the weights may have already been updated at the PS (by other learners)
  - Gradients returned by this learner are stale (i.e., were evaluated at an older version of the model)
- Stale gradients can make SGD unstable, slowdown convergence, cause sub-optimal convergence (compared to Sync-SGD)

Stale Gradient Problem in Async-SGD



$$\vdots$$

$$W_{101} = W_{100} - \lambda \Delta W_{97}$$ staleness = 3

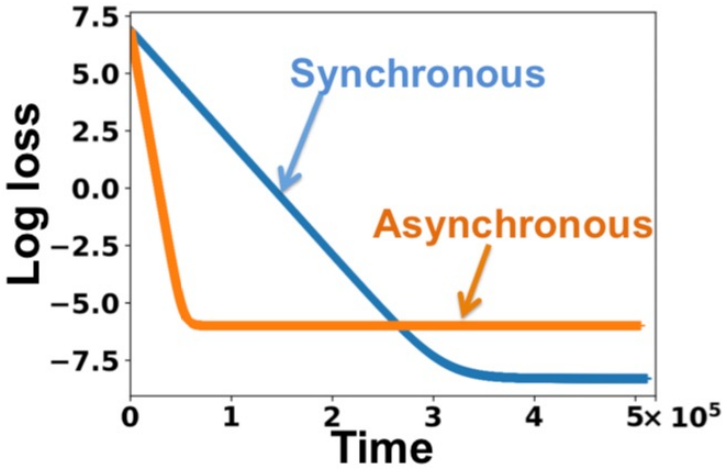
$$W_{102} = W_{101} - \lambda \Delta W_{98}$$

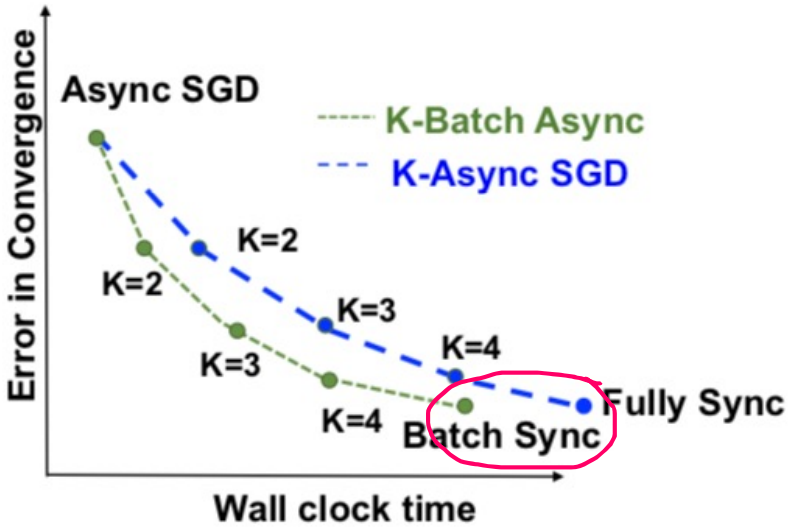
$$W_{103} = W_{102} - \lambda \Delta W_{100}$$ staleness = 2

$$W_{104} = W_{103} - \lambda \Delta W_{99}$$

$$\vdots$$

Error-Runtime Tradeoff in SGD Variants





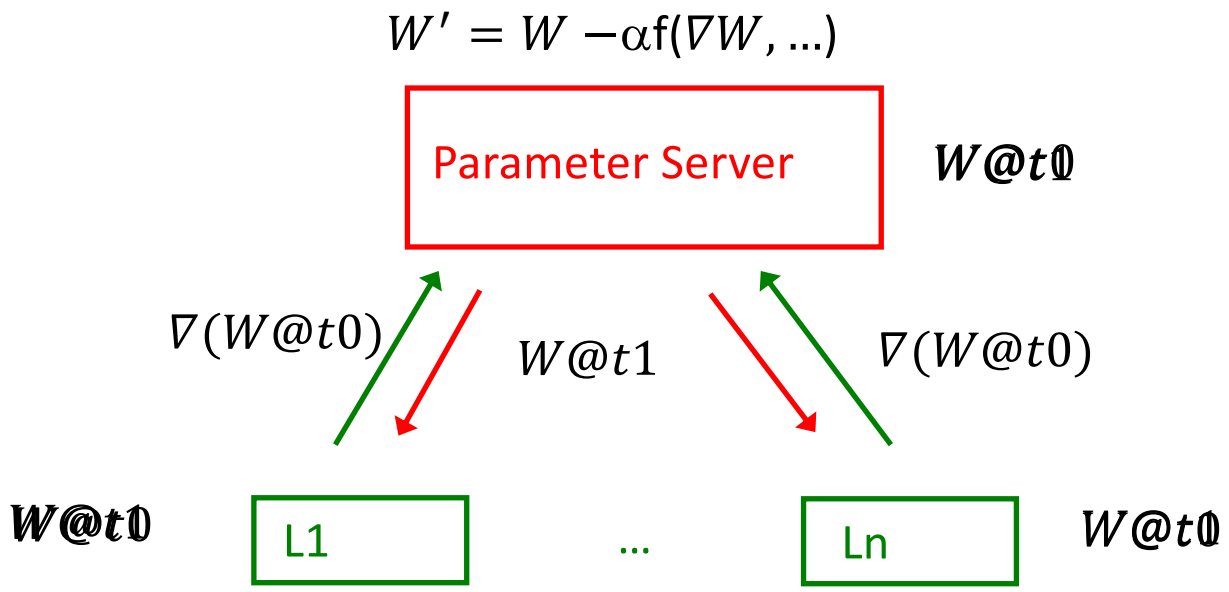
- Error-runtime trade-off for Sync and Async-SGD with same learning rate.
- Async-SGD has faster decay with time but a higher error floor.

Terminologies

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

Model Accuracy and Runtime Tradeoff in Distributed Deep Learning: A Systematic Study. ICDM 2016

Hard-sync


$W' = W - \alpha f(\nabla W, ...)$
Parameter Server
$W@t1$
$\nabla(W@t0)$
$W@t1$
$\nabla(W@t0)$
$W@t1$
L1
...
Ln
$W@t1$

Hard-sync

lambda - number of learners

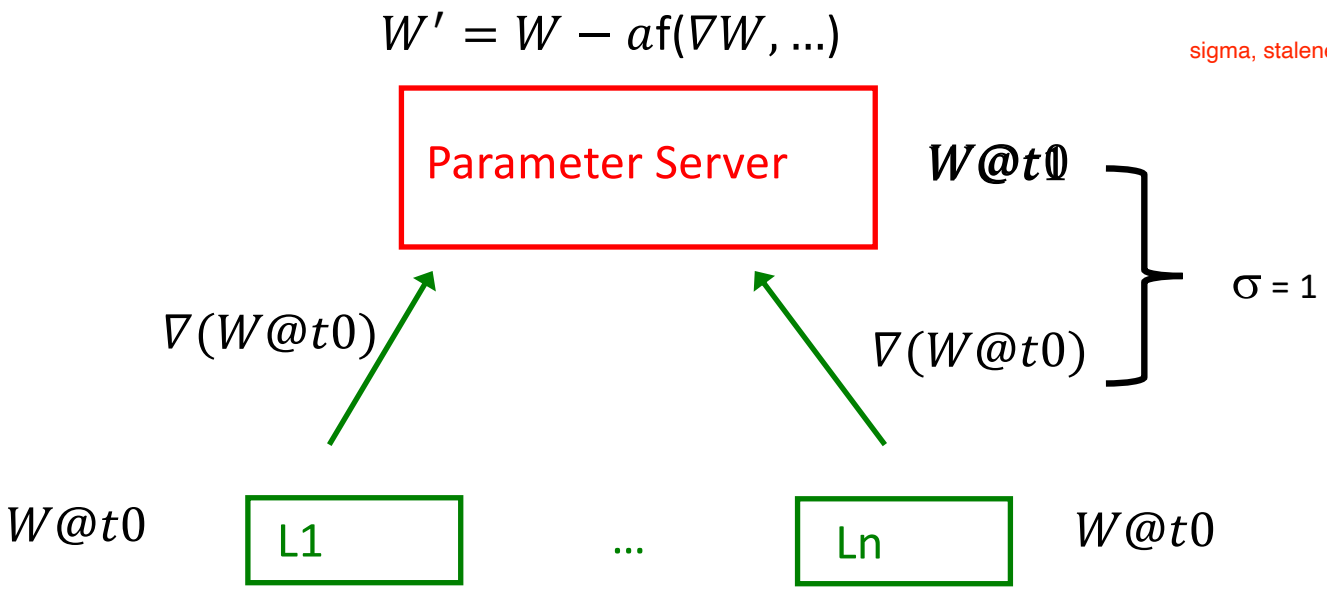
$$\nabla W(i) = \frac{1}{\lambda}\sum_{l=1}^{l=\lambda} \nabla W_l(i)$$

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

μ, minibatch size.

Equivalent to single-learner SGD w/ batchsize of λ*μ

Soft-sync


$W' = W - a\mathrm{f}(\nabla W, \ldots)$
sigma, staleness of the gradients.
Parameter Server
$W@t1$
$\nabla(W@t0)$
$\nabla(W@t0)$
σ = 1
$W@t0$
L1
...
Ln
$W@t0$

n-softsync

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

c

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

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

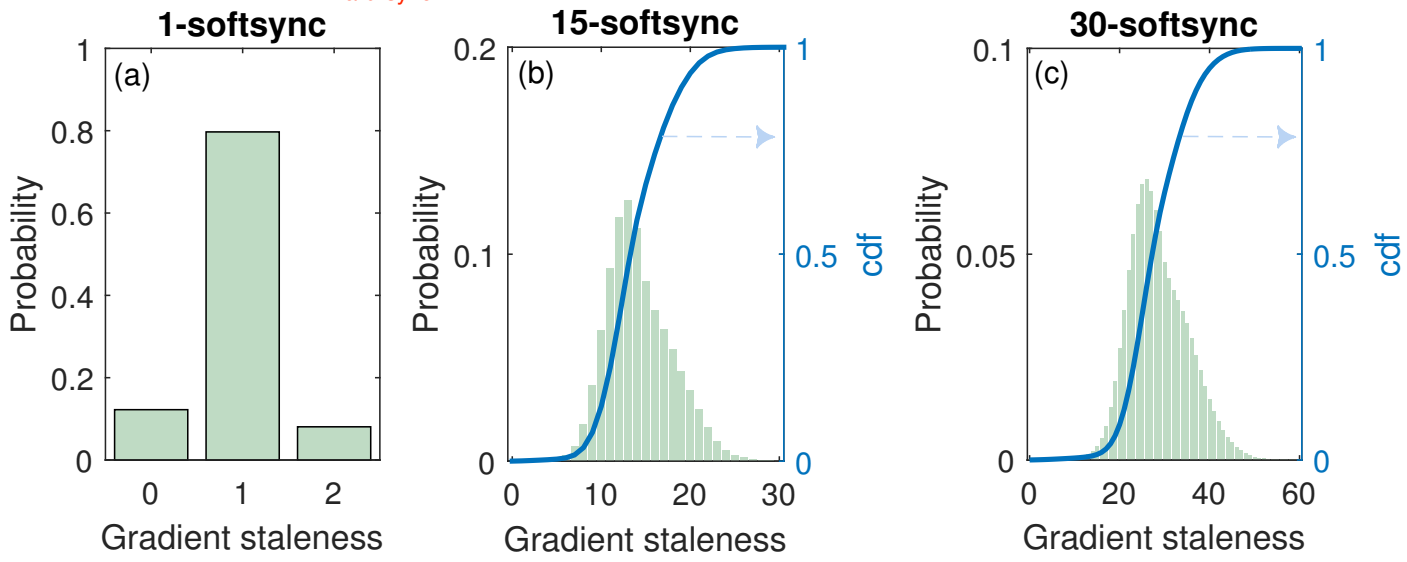
Staleness distribution

$\lambda = 30$ number of learners

Homogeneous system

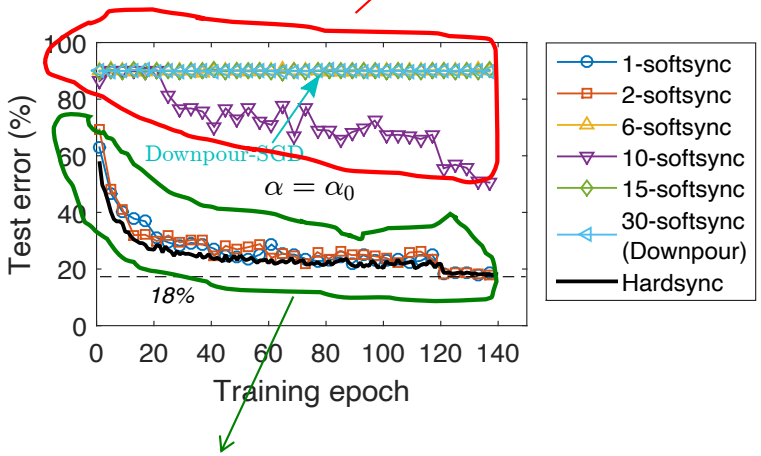
Statleness distribution is centered around n n-softsync

Hard-sync



Model Accuracy (baseline)

staleness = 10,15,30



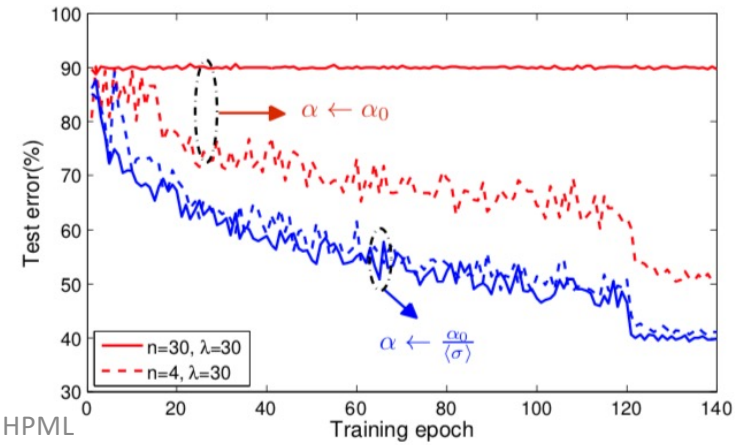
staleness = 0,1, 2, 6

CIFAR10

Staleness Dependent Learning Rate -1

- With staleness-dependent learning rate setting Async-SGD can achieve accuracy comparable to Sync-SGD while achieving linear speedup of ASGD
- Decrease the learning rate by mean staleness

$$learning\ rate\ (\alpha) = \frac{base\ learning\ rate\ (\alpha_0)}{average\ staleness\ (\langle\sigma\rangle)}$$



Staleness Dependent Learning Rate -2

$$\eta_j = \min\left\{\frac{C}{||\mathbf{w}_j - \mathbf{w}_{\tau(j)}||_2^2}, \eta_{max}\right\}$$

η_j: learning rate at jth iteration of parameters at PS

w_j: parameter value at jth iteration at PS ⟵ to calculate staleness

$w_{\tau(j)}$: parameter value used by the learner (to calculate gradient) whose gradient pushing triggered jth iteration at PS

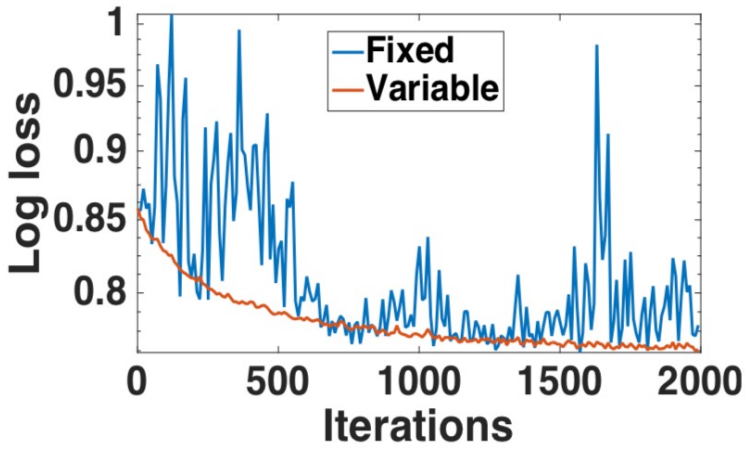
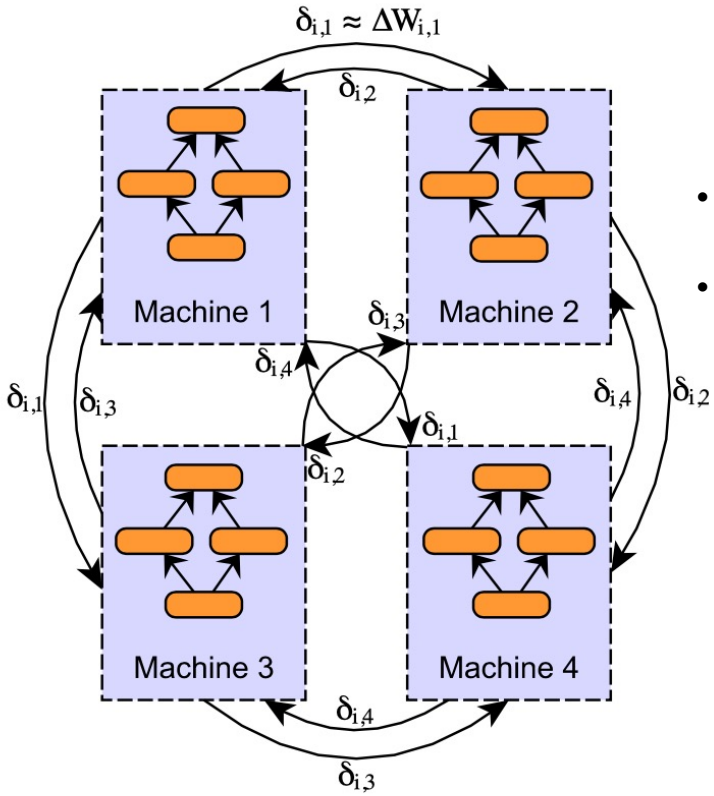


Figure 9: Async-SGD on CIFAR10 dataset, with $X \sim \exp 20$, mini-batch size $m = 250$ and $P = 40$ learners. We compare fixed $\eta = 0.01$, and the variable schedule given in (13) for $\eta_{max} = 0.01$ and $C = 0.005\eta_{max}$. Observe that the proposed schedule can give fast convergence, and also maintain stability, while the fixed η algorithm becomes unstable.

Is C a hyperparameter ?

Is the learning rate same for different model parameters ?

Decentralized Aggregation: Peer-to-Peer



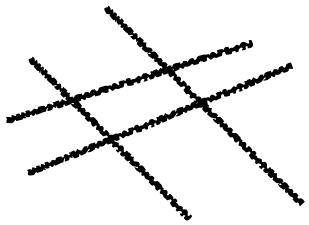
- Peer to peer communication is used to transmit model updates between workers).
- Updates are heavily compressed, resulting in the size of network communications being reduced by some 3 orders of magnitude.

Distributed Scaling Speed up -- Facebook

- Demonstrated training of a ResNet-50 network in one hour on 256 GPUs
- Major techniques:
  - Data parallelism
  - Very large batch sizes
  - Learning rate adjustment technique to deal with large batches
  - Gradual warm up of learning rate from low to high value in 5 epochs ?
  - MPI_AllReduce for communication between machines
  - NCCL communication library between GPUs on a machine connected using NVLink
  - Gradient aggregation performed *in parallel* with backprop
    - No data dependency between gradients across layers ?
    - As soon as the gradient for a layer is computed, it is aggregated across workers, while gradient computation for the next layer continues

Imagenet Distributed Training

- Each server contains 8 NVIDIA Tesla P100 GPUs interconnected with NVIDIA NVLink
- 32 servers => 256 P100 GPUs
- 50 Gbit Ethernet
- Dataset: 1000-way ImageNet classification task; ~1.28 million training images; 50,000 validation images; top- 1 error
- ResNet-50
- Learning rate: $\eta = 0.1 \cdot \frac{kn}{256}$ **How was this chosen ? What is k and n ?**
  Is it batch size per GPU (k) and total number of GPUs (n) ?
- Mini-batch size per GPU: 32 (fixed, weak scaling across servers)
- Caffe2



Learning Rates for Large Minibatches

Linear Scaling Rule: *When the minibatch size is multiplied by k, multiply the learning rate by k.*

$$w_{t+1} = w_t - \eta \frac{1}{n} \sum_{x \in \mathcal{B}} \nabla l(x, w_t)$$

$$w_{t+k} = w_t - \eta \frac{1}{n} \sum_{j<k} \sum_{x \in \mathcal{B}_j} \nabla l(x, w_{t+j})$$

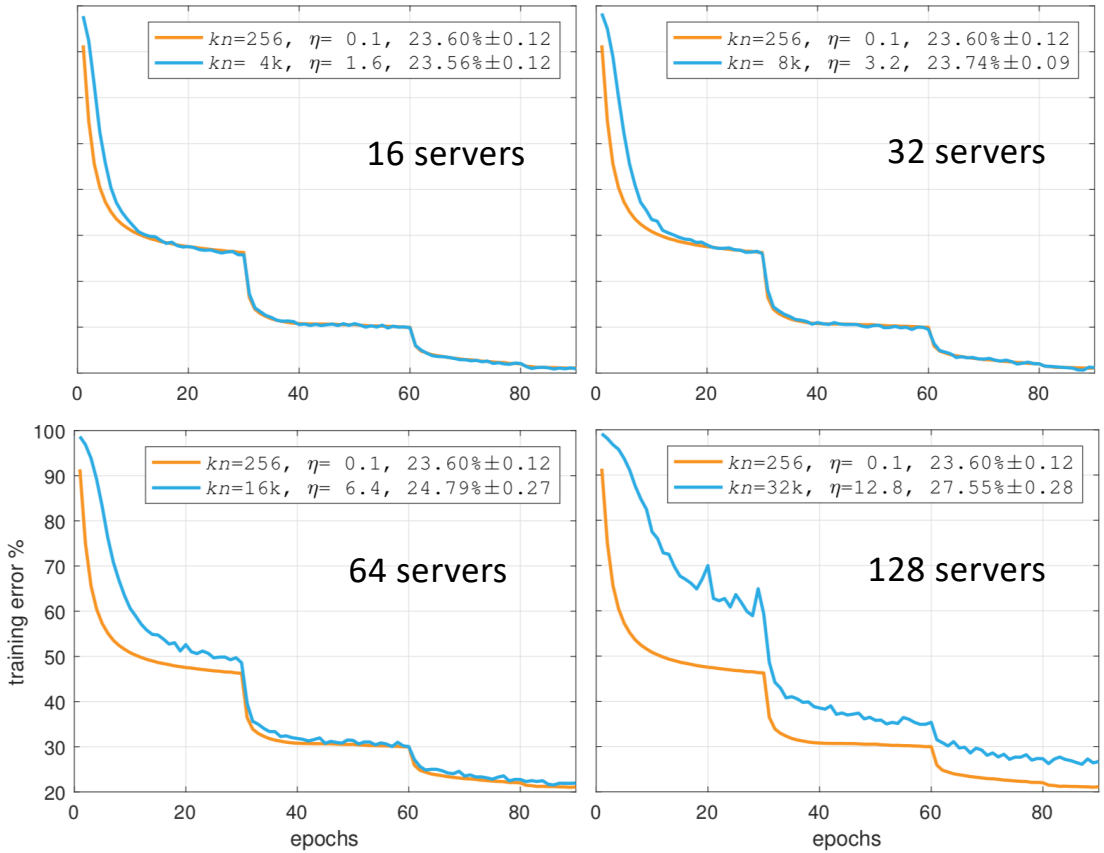
$$\hat{w}_{t+1} = w_t - \hat{\eta} \frac{1}{kn} \sum_{j<k} \sum_{x \in \mathcal{B}_j} \nabla l(x, w_t)$$

Suppose, $\nabla l(x, w_t) \approx \nabla l(x, w_{t+j})$ for $j < k$,
when will

$$\hat{w}_{t+1} \approx w_{t+k},$$

eta_hat = k * eta

Training Error Vs Batch Size: Distributed Training



Beyond 8k batch size, the training error deteriorates

Runtime Scaling

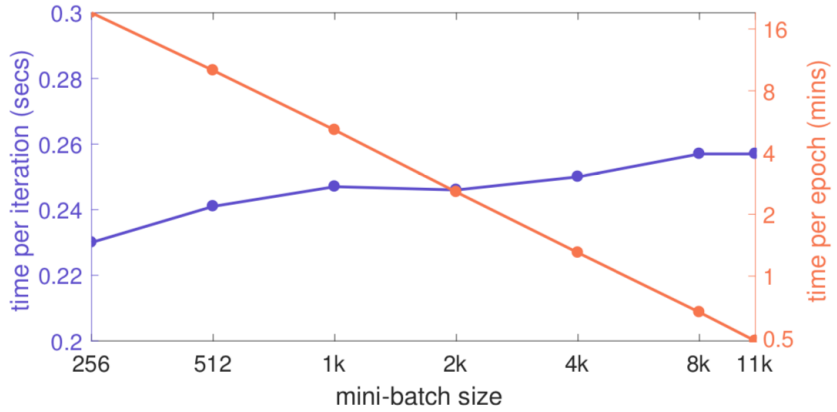


Figure 7. **Distributed synchronous SGD timing.** Time per iteration (seconds) and time per ImageNet epoch (minutes) for training with different minibatch sizes. The baseline ($kn = 256$) uses 8 GPUs in a single server , while all other training runs distribute training over ($kn/256$) server. With 352 GPUs (44 servers) our implementation completes one pass over all ∼1.28 million ImageNet training images in about 30 seconds.

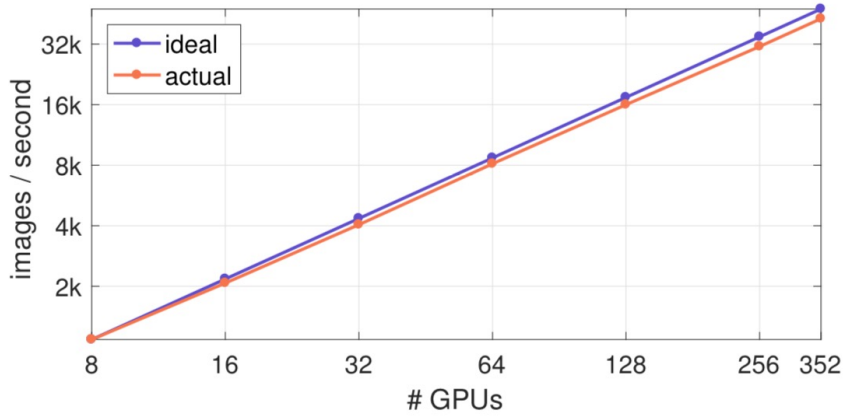


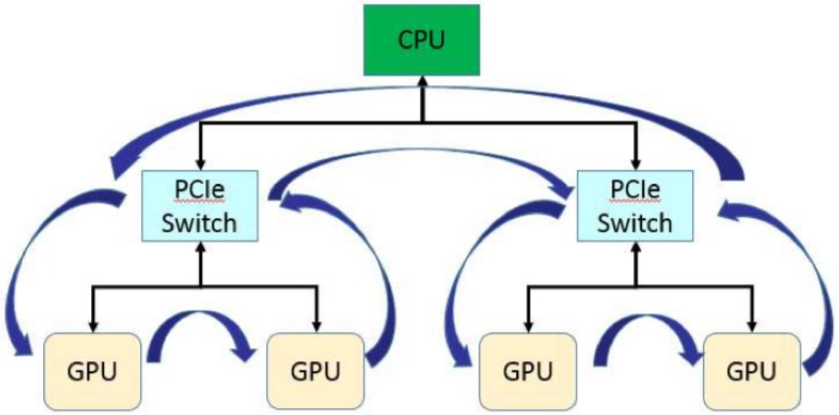
Figure 8. **Distributed synchronous SGD throughput.** The small overhead when moving from a single server with 8 GPUs to multi-server distributed training (Figure 7, blue curve) results in linear throughput scaling that is marginally below ideal scaling (∼90% efficiency). Most of the allreduce communication time is hidden by pipelining allreduce operations with gradient computation. Moreover, this is achieved with commodity Ethernet hardware.

Questions

- Why time per iteration increases little with increasing minibatch size ?
- Why time per epoch decreases with increasing batch size ?
- With 44 servers how much time it takes to finish 1 epoch of training ?
- Can you get throughput (images/sec) from time per epoch ? Do you need to know batch size ?
- Can you get throughput (images/sec) from time to process a mini-batch (iteration) ? Do you need to know batch size ?
- If K-batch sync or K-sync (K < number of servers) was applied would the convergence been faster ? What about the final training error ?

PowerAI DDL --IBM

Communication ring over network tree topology



All nodes communicate concurrently with one other node

- 64 IBM Power8 servers
- Each server has 4 NVidia Tesla P100 GPUs connected through NVLink
- Batch size: 32 per GPU
- Effective batch size: 8192
- Infiniband
- IBM-Caffe and optimized Torch

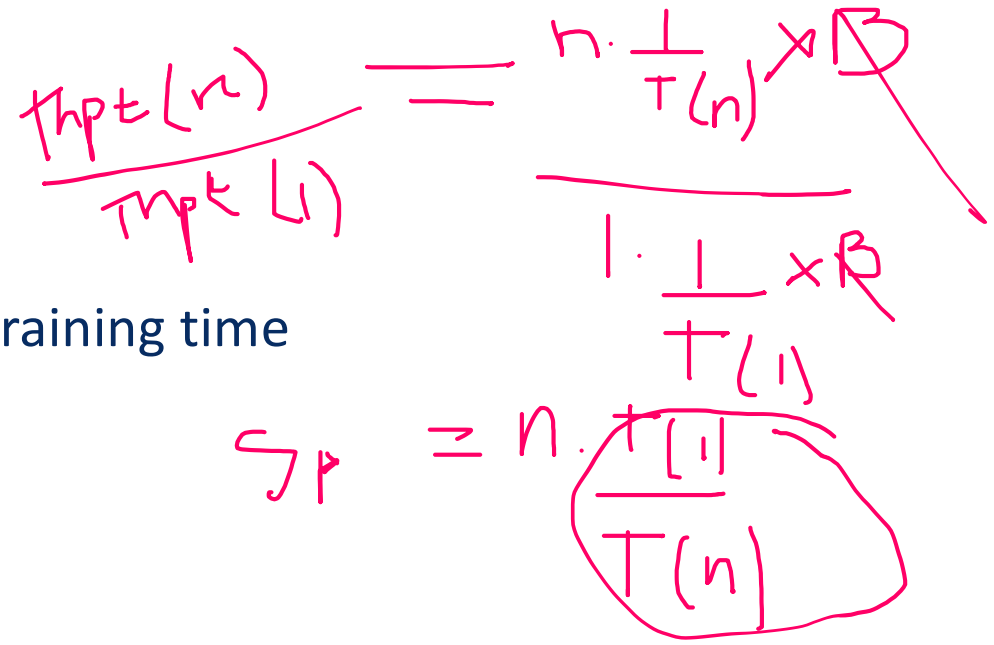
PowerAI DDL Library

- Based on MPI
- Object is a tensor of gradients along with metadata (host, device etc.)

matrices -

Distributed Deep Learning Benchmarking Methodology

- Speedup
- Scaling efficiency
- Accuracy and end-to-end training time
- Neural network
- Deep learning framework
- GPU type
- Communication overhead



speedup = n * scaling_efficinecy

Scaling efficiency

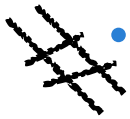
it is meaningless without convergence

measure of communication time

one iteration (one batch), not one epoch

- Scaling efficiency: ratio between the run time of one iteration on a single GPU and the run time of one iteration when distributed over N GPUs with the same batch size per GPU. **Why is this ratio a measure of scaling efficiency ?**

T(n) = t_n_compute + t_communication



- Too big a (effective) batch size will result in converging to an unacceptable accuracy or no convergence at all
- A high scaling efficiency without being backed up by convergence to a good accuracy and end to end training time is meaningless

a good matric would be that combine convergence time and accuracy

if compute to comm. ratio is higher it will show lower scaling efficiency

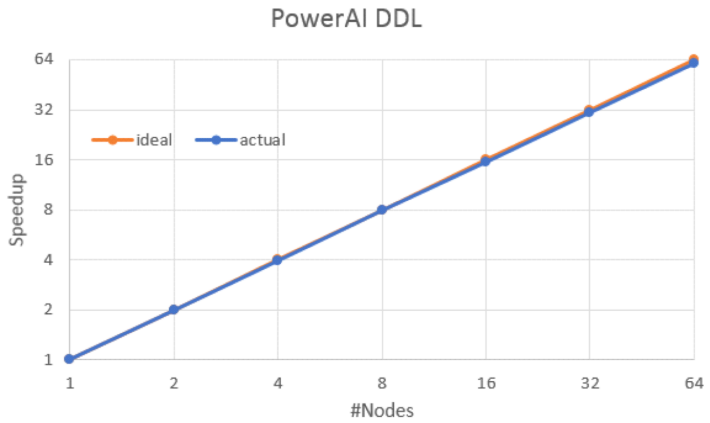
Other considerations

- Systems under comparison should train to same accuracy
- Accuracy should be reported on sufficiently large test set to avoid overfit
- Compute to communication ratio can vary widely for different neural networks. Using a neural network with high compute to communication ratio can hide the ills of an inferior distributed Deep Learning system.
  - a sub-optimal communication algorithm or low bandwidth interconnect will not matter that much a bad NN, (with high compute time) - will result in high compute to comm. ratio. But we won't be able to comment on distributed system's performace, as NN's performance is bad as well. it will hide our distributed network
- Computation time for one Deep Learning iteration can vary by up to 50% when different Deep Learning frameworks are being used. This increases the compute to communication ratio and gives the inferior distributed Deep Learning system an unfair uplift to the scaling efficiency.

Other considerations

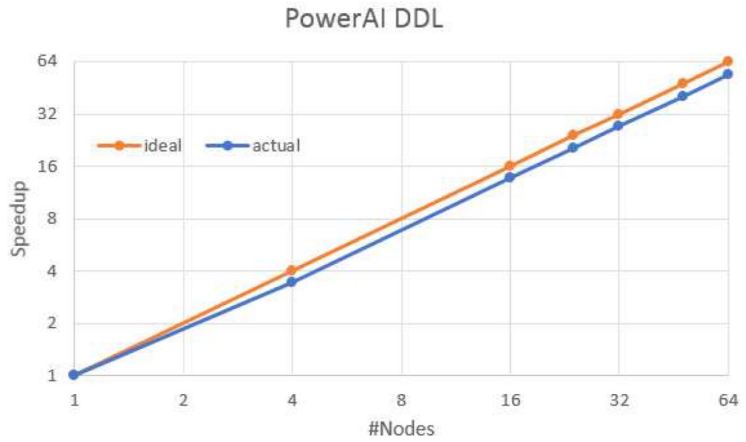
- A slower GPU increases the compute to communication ratio and again gives the inferior distributed Deep Learning system an unfair uplift to the scaling efficiency.
  - Nvidia P100 GPUs are approximately 3X faster than Nvidia K40 GPUs.
  - When evaluating the communication algorithm and the interconnect capability of a Deep Learning system, it is important to use a high performance GPU.
- Communication overhead is the run time of one iteration when distributed over N GPUs minus the run time of one iteration on a single GPU.
  - Includes the communication latency and the time it takes to send the message (gradients) among the GPUs.
  - Communication overhead gives an indication of the quality of the communication algorithm and the interconnect bandwidth.

PowerAI DDL --IBM



#GPUs	4	8	16	32	64	128	256
#Nodes	1	2	4	8	16	32	64
Speedup	1.0	2.0	3.9	7.9	15.5	30.5	60.6
Scaling efficiency	1.00	1.00	0.98	0.99	0.97	0.95	0.95

Figure 2: Resnet-50 for 1K classes using up to 256 GPUs with Caffe.



#GPUs	4	16	64	96	128	192	256
#Nodes	1	4	16	24	32	48	64
Speedup	1.0	3.5	13.7	20.5	27.2	40.3	53.7
Scaling efficiency	1.00	0.86	0.86	0.85	0.85	0.84	0.84

Figure 3: Resnet-50 for 1K classes using up to 256 GPUs with Torch.

- In Torch implementation, the reduction of gradients is not overlapped with compute; scaling efficiency is 84% vs 95% with Caffe

speedup = n * scaling_efficiency (derive this)

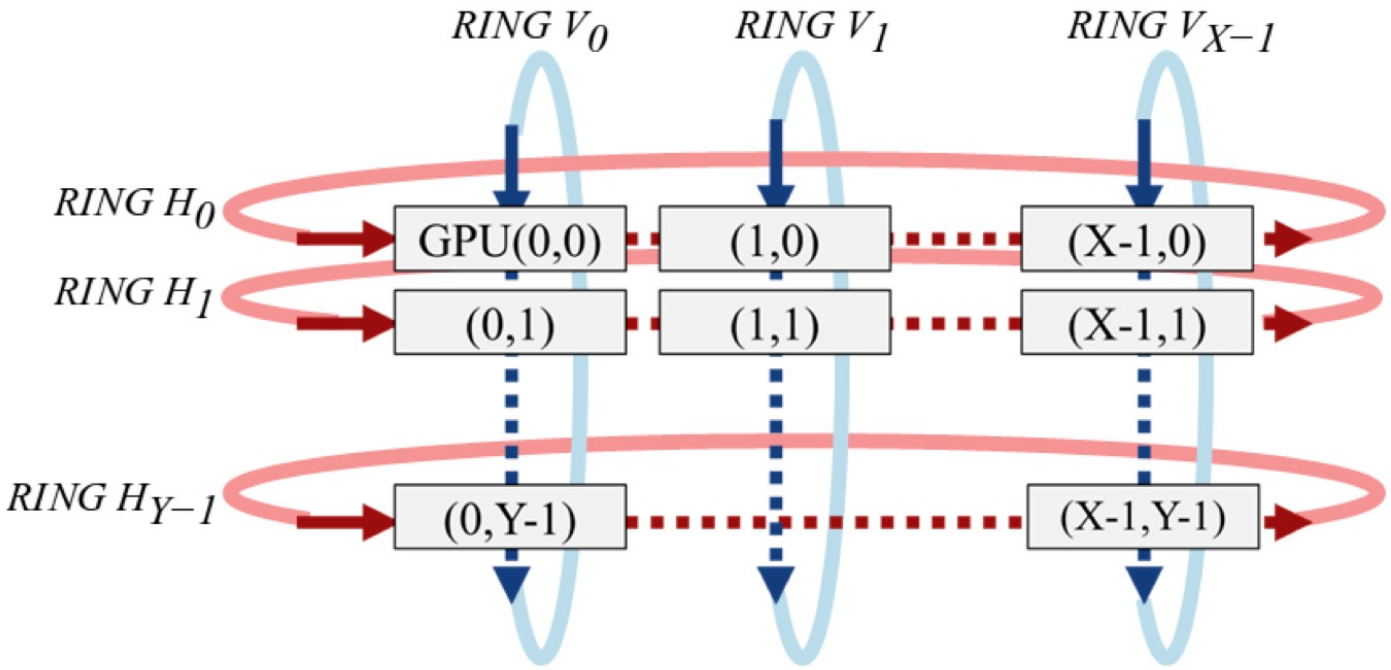
Imagenet1K/ResNet50 Training at Scale

Work	Batch size	Processor	DL Library	Interconnect	Training Time	Top-1 Accuracy	Scaling Efficiency
He et al	256	Tesla P100 x8	Caffe		29 hrs	75.3%	
Goyal et al (Facebook)	8K	Tesla P100 x256	Caffe2	50 Gbit Ethernet	60 mins	76.3%	~90%
Cho et al (IBM)	8K	Tesla P100 x256	Caffe	Infiniband	50 mins	75.01%	95%
Smith et al	8K → 16K	Full TPU Pod	Tensorflow		30 mins	76.1%	
Akiba et al	32K	Tesla P100 x1024	Chainer	Infiniband FDR	15 mins	74.9%	80%
Jia et al	64K	Tesla P40 x2048	Tensorflow	100 Gbit Ethernet	6.6 mins	75.8%	87.9%
Ying et al	32K	TPU v3 x1024	Tensorflow		2.2 mins	76.3%	
Ying et al	64K	TPU v3 x1024	Tensorflow		1.8 mins	75.2%	
Mikami et al	54K	Tesla V100 x3456	NNL	Infiniband EDR x2	2.0 mins	75.29%	84.75%

Cho et al achieved highest scaling efficiency; Goyal et al and Ying et al achieved highest accuracy

2-D Torus Topology for inter-gpu communication

multiple rings in horizontal and vertical orientations.



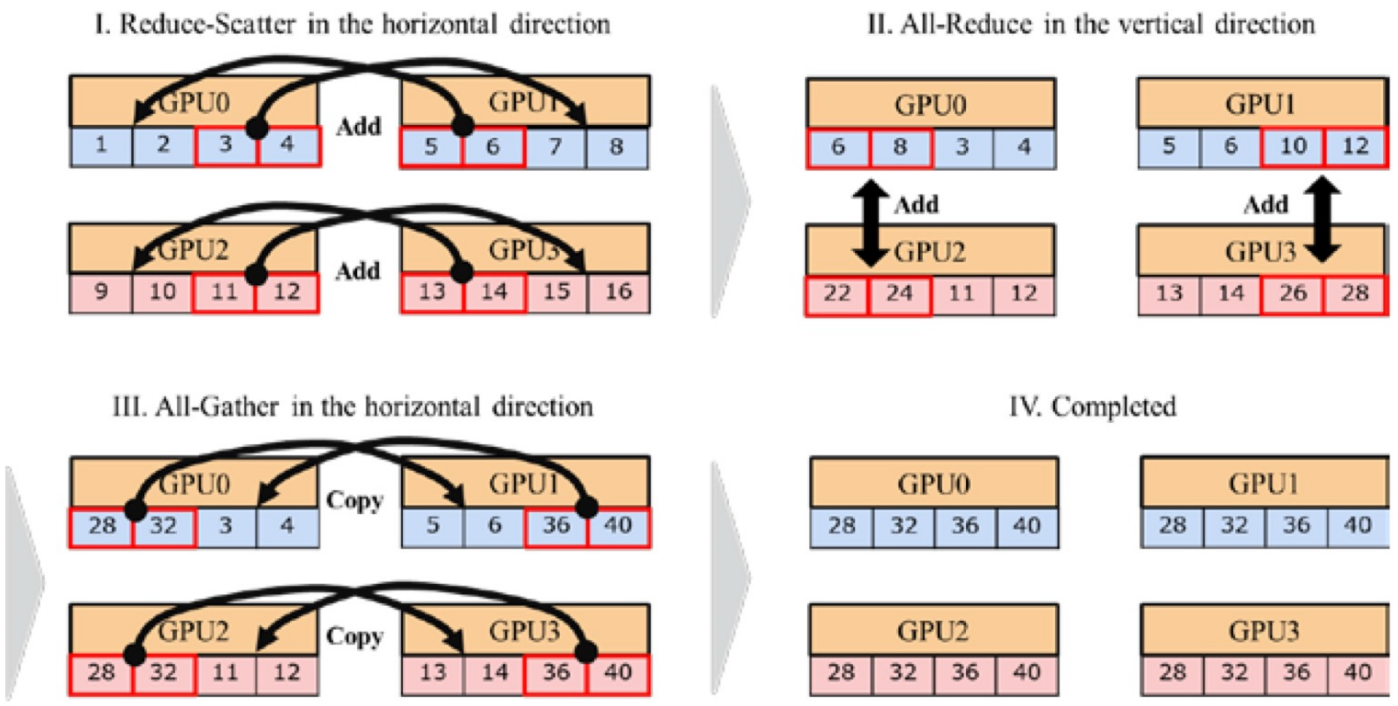
2-D Torus all-reduce

try it out

a cool way to share data efficiently

network topology

2D-Torus all-reduce steps of a 4-GPU cluster, arranged in 2x2 grid



https://arxiv.org/pdf/1811.05233.pdf

Meaningful Distributed DL System Comparison

- A popular neural network that has been widely trained and tuned
- Use a Deep Learning framework that is computationally-efficient
- Train to best accuracy on high performance GPUs
- Report:
  - Accuracy achieved
  - Training time to attain the accuracy
  - Scaling efficiency
  - Communication overhead
- Metric may also include power and cost.

Pytorch

- torch.distributed package
- https://pytorch.org/tutorials/beginner/dist_overview.html
- Data Parallel Training:
  - torch.nn.DataParallel: multi-thread parallelism with GIL
  - torch.nn.parallel.DistributedData Parallel: multi-process parallelism

PyTorch Distributed DL

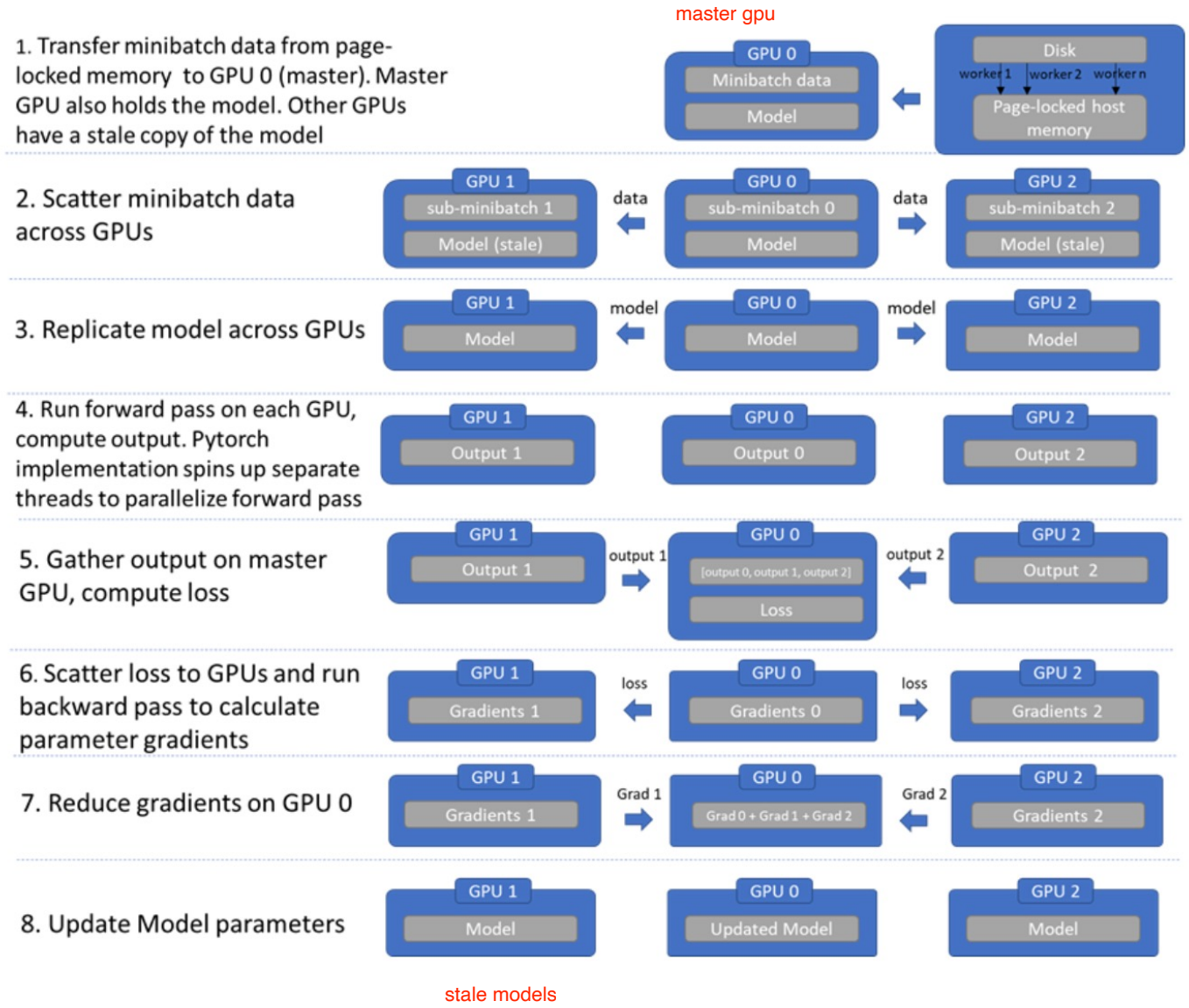
Data Parallelism: Multiple GPU configurations

- PyTorch provides several options for data-parallel training. For applications that gradually grow from simple to complex and from prototype to production, the common development trajectory would be:
  1. Use single-device training if the data and model can fit in one GPU, and training speed is not a concern.
  2. Use single-machine multi-GPU DataParallel to make use of multiple GPUs on a single machine to speed up training with minimal code changes.
  3. Use single-machine multi-GPU DistributedDataParallel, if you would like to further speed up training and are willing to write a little more code to set it up.
  4. Use multi-machine DistributedDataParallel and the launching script, if the application needs to scale across machine boundaries.

Data Parallel

One GPU (0) acts as the master GPU and coordinates data transfer.

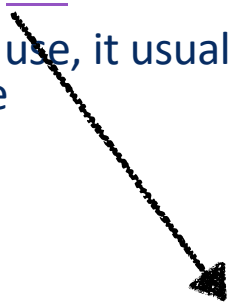
Implemented in PyTorch data_parallel module



DataParallel Example

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

- Implements data parallelism at the module level
- Forward pass:
  - Replicates models on each GPU
  - **Divides each batch into #devices and does it in parallel**
- Backward pass: **computes gradients separately and sums them into the original module**
- Single-process multi-thread parallelism naturally suffers from GIL contention.
- Very easy to use, it usually does not offer the best performance



global interpreter lock

```
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
```

Inefficiencies in DataParallel

- Redundant data copies
  - Data is copied from host to master GPU and then sub-minibatches are scattered across other GPUs
- Model replication across GPUs before forward pass
  - Since model parameters are updated on the master GPU, model must be re-synced at the beginning of every forward pass
- Thread creation/destruction overhead for each batch
  - Parallel forward is implemented in multiple threads (this could just be a Pytorch issue)
- Gradient reduction pipelining opportunity left unexploited
  - In Pytorch 1.01 data-parallel implementation, gradient reduction happens at the end of backward pass. I'll discuss this in more detail in the distributed data parallel section.
- Unnecessary gather of model outputs on master GPU
- Uneven GPU utilization
  - Loss calculation performed on master GPU
  - Gradient reduction, parameter updates on master GPU

torch.nn.DistributedDataParallel

- torch.nn.parallel.DistributedDataParallel implements data parallelism at the module 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**
- https://pytorch.org/docs/stable/generated/torch.nn.parallel.DistributedDataParallel.html
- DistributedDataParallel is proven to be significantly faster than torch.nn.DataParallel for single-node multi-GPU data parallel training.

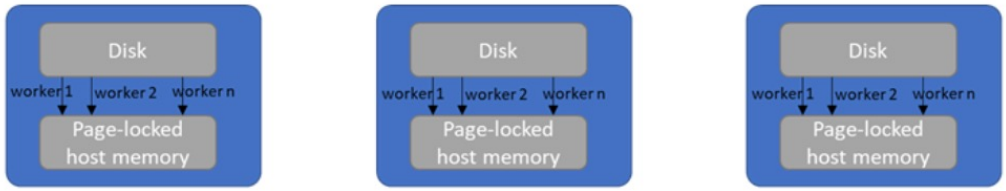
no concept of master gpu

Distributed Data Parallel

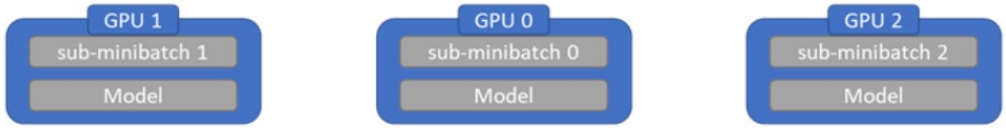
No master GPUs

Implemented in PyTorch DistributedDataParallel module

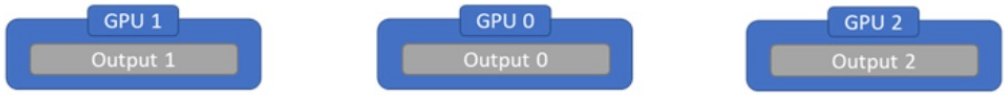
1. Load data from disk into page-locked memory on the host. Use multiple worker processes to parallelize data load. Distributed minibatch sampler ensures that each process loads non-overlapping data



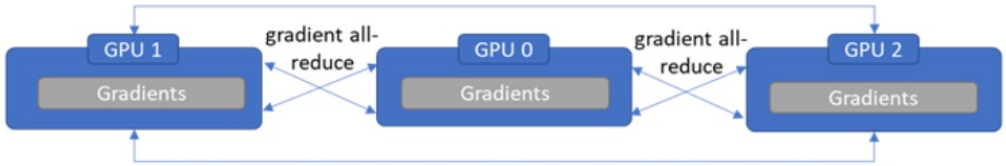
2. Transfer minibatch data from page-locked memory to each GPU concurrently. No data broadcast is needed. Each GPU has an identical copy of the model and no model broadcast is needed either



3. Run forward pass on each GPU, compute output



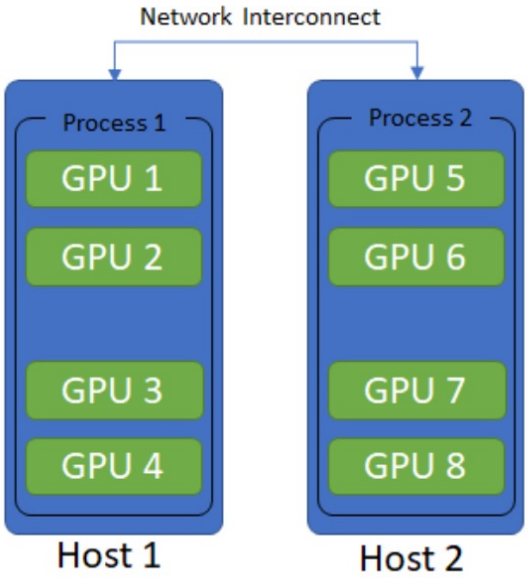
4. Compute loss, run backward pass to compute gradients. Perform gradient all-reduce in parallel with gradient computation



5. Update Model parameters. Because each GPU started with an identical copy of the model and gradients were all-reduced, weights updates on all GPUs are identical. Thus no model sync is required

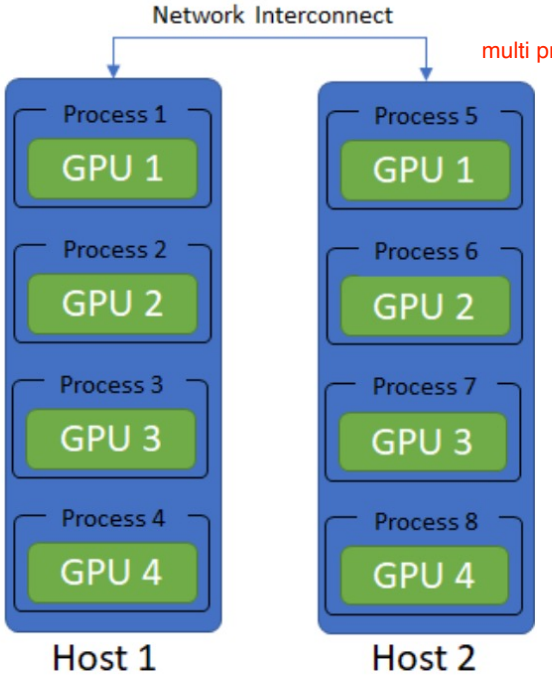
Distributed Data Parallel in Multi-host Setting

multi threaded parallelism



Configuration 1: Each process operates on all four GPUs in data parallel mode

multi process parallelism



Configuration 2: Each process operates on a single GPU

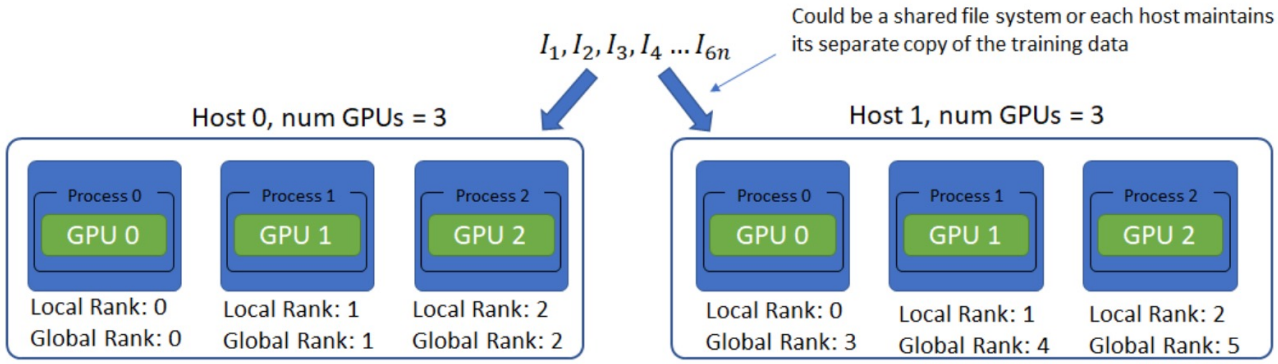
Steps in multi-host setting

- Distributed-data-parallel is launched on each host independently
- Mechanism needed to have synchronization between multiple processes running on different hosts
  - **init_process_group** function that uses a shared file system or a TCP IP address/port to sync the processes.
- Each process should load non-overlapping copies of the data
  - DistributedDataSampler

DataParallel (DP) vs. Distributed DataParallel (DDP)

- DP uses multi-threaded parallelism, while DDP uses multi-process parallelism.
- DP is slower than DDP due to Python GIL contention among threads
- DP repeatedly copies model parameters between the threads which introduces yet another overhead. communication overhead

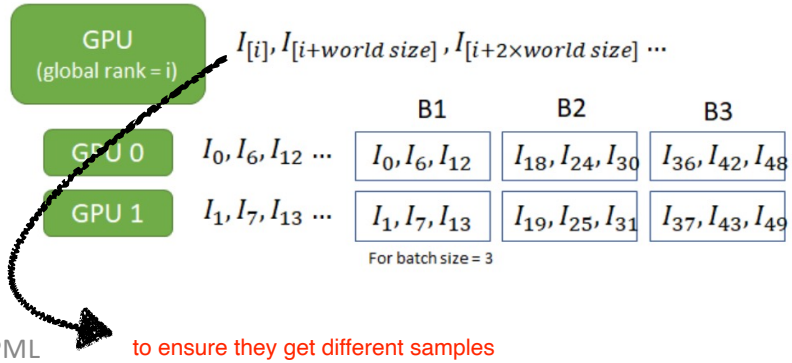
DistributedDataSampler



Worldsize: *num Hosts* × *num GPUs per host* = 2×3 = 6

Each process knows its *local rank* and *host number*. Can calculate *global rank* from this info.

Global rank = local rank + num GPUs per host × *host number*



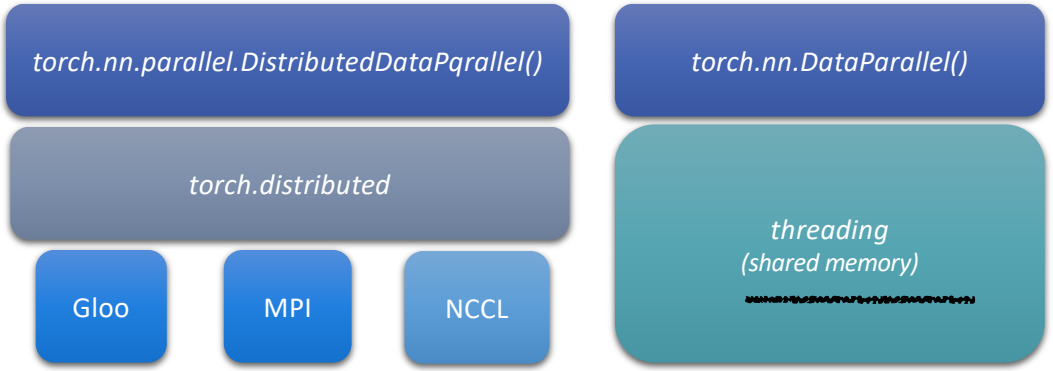

```
def __iter__(self):
    # deterministically shuffle based on epoch
    g = torch.Generator()
    g.manual_seed(self.epoch)
    indices = torch.randperm(len(self.dataset), generator=g).tolist()

    # add extra samples to make it evenly divisible
    indices += indices[:(self.total_size - len(indices))]
    assert len(indices) == self.total_size

    # subsample
    indices = indices[self.rank:self.total_size:self.num_replicas]
    print("len indices {0}".format(len(indices)))
    return iter(indices)
```

DistributedSampler
(distributed.py)

to ensure they get different samples

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***: distributed communication package; provides an MPI-like interface for exchanging tensor data across multi-machine networks. It supports a few different backends and initialization methods
  - **Backends:** gloo, MPI, NCCL
- ***torch.nn.DataParallel()*** *is not distributed: valid in a single node only*
  - *Based on **threading** module*



PyTorch – *torch.distributed* package support

- PyTorch offer support for several backends: gloo, mpi, nccl
- NCCL and GLOO backends are available by default
- MPI is an optional backend that can only be included if you build PyTorch from source. (e.g. building PyTorch on a host that has MPI installed.)
- **MPI** has the more flexible support for collectives with gpu

Backend	`gloo`		`mpi`		`nccl`	
Device	CPU	GPU	CPU	GPU	CPU	GPU
send	✓	✘	✓	?	✘	✓
recv	✓	✘	✓	?	✘	✓
broadcast	✓	✓	✓	?	✘	✓
all_reduce	✓	✓	✓	?	✘	✓
reduce	✓	✘	✓	?	✘	✓
all_gather	✓	✘	✓	?	✘	✓
gather	✓	✘	✓	?	✘	✓
scatter	✓	✘	✓	?	✘	✘
reduce_scatter	✘	✘	✘	✘	✘	✓
all_to_all	✘	✘	✓	?	✘	✓
barrier	✓	✘	✓	?	✘	✓

PyTorch - Distributed Deep Learning

- **Simple Approach:** use ***torch.nn.parallel.DistributedDataParallel()*** class
  - Only available with Gloo and NCCL backends
  - Cannot use implemented algorithms
- **Advanced Approach:** use ***torch.distributed*** package
  - Write your own distributed algorithm with **MPI** using collectives and point-to-point communication
- Common guideline
  - Use the NCCL backend for distributed **GPU** training
  - Use the Gloo backend for distributed **CPU** training.

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

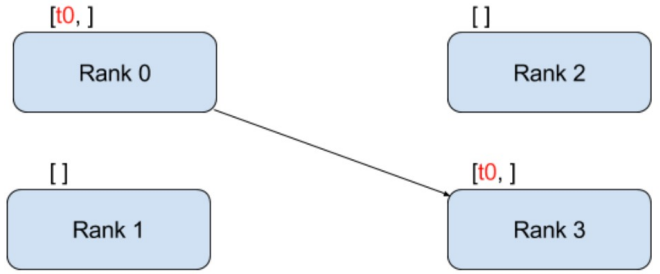
```
import os
import torch
import torch.distributed as dist

# 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 supported communications

- Point-to-point communication
  - transfer of data from one process
  - achieved through the send and recv functions or their *immediate* counter-parts, isend and irecv
- Collective communication
  - allow for communication patterns across all processes in a **group**
  - group is a subset of all processes
  - default group is 'world' (set of all the processes)
- Writing distributed application with Pytorch tutorial.
  https://github.com/pytorch/tutorials/blob/master/intermediate_source/dist_tuto.rst

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



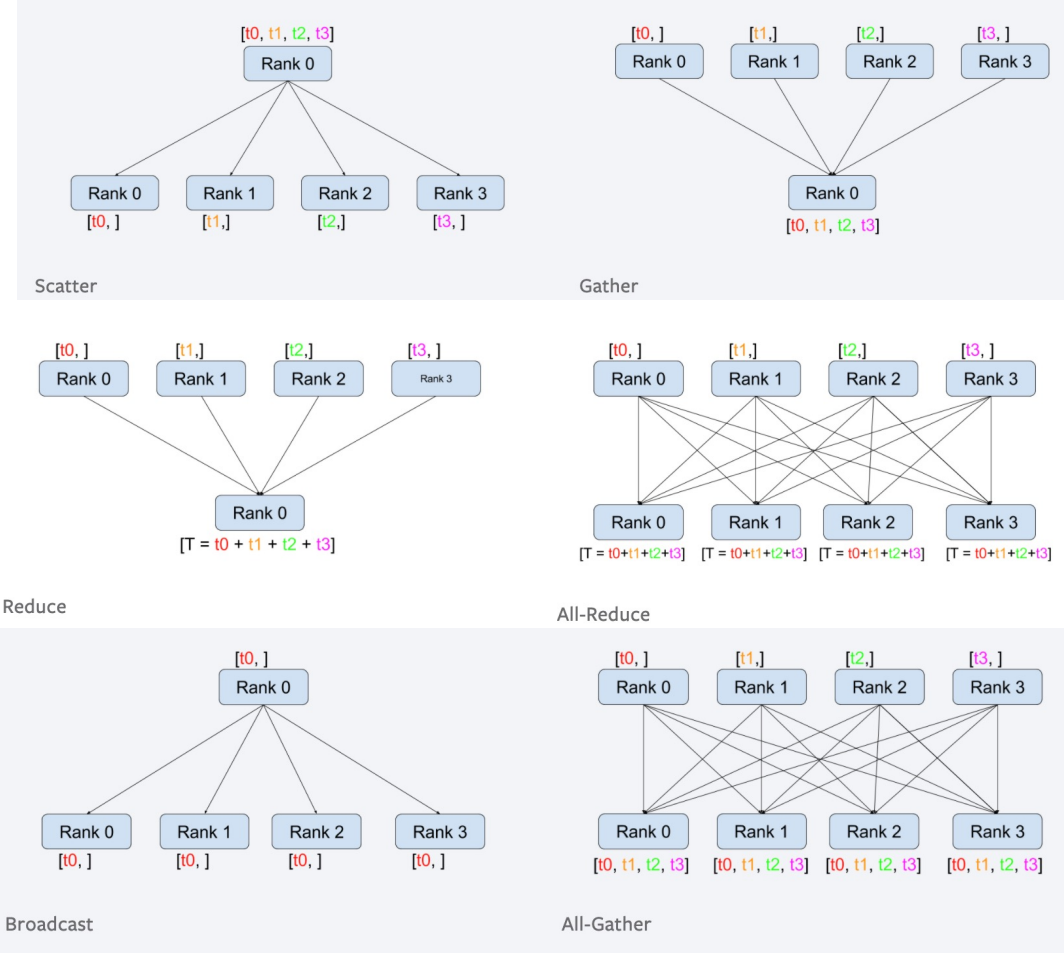

```
"""Blocking point-to-point communication."""

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 (immediates):
  - *dist.isend()*
  - *dist.irecv()*
- Support for *tensor* type:
  - serialization done automatically
- Should not modify the sent tensor nor access the received tensor before req.wait() has completed
- Writing to tensor after dist.isend() will result in undefined behavior.
- Reading from tensor after dist.irecv() will result in undefined behavior

```
"""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()  as it is non-blocking
    print('Rank ', rank, ' has data ', tensor[0])
```

Collectives



PyTorch – *torch.distributed* All-Reduce

- All-Reduce Operation
- Using the *dist.reduce_op.SUM* reduce operation
- Available reduce operations (commutative math):
  - *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])
```

https://pytorch.org/tutorials/intermediate/dist_tuto.html#our-own-ring-allreduce

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 its data to the next rank
  - The last one sends it back to the first

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

    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_buff[:]
        else:
            # Send recv_buff
            send_req = dist.isend(recv_buff, right)
            dist.recv(send_buff, left)
            accum[:] += send_buff[:]
        send_req.wait()
    recv[:] = accum[:]
```

n - 1 steps are required in ring all reduce? - Nice

Lesson Key Points

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