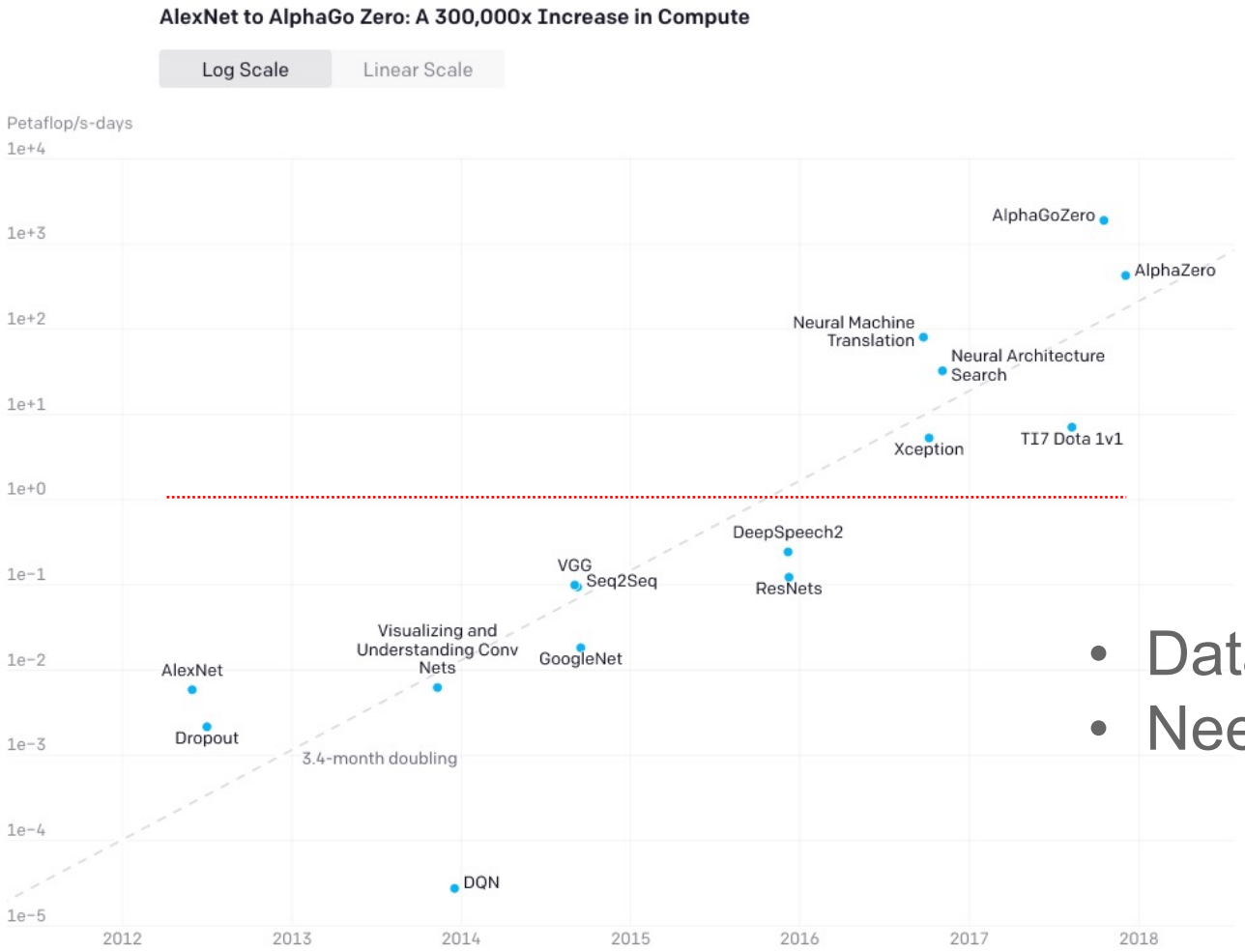
# Distributed Deep Learning on HAL

2021.10.20

Dawei Mu


NCSA | National Center for
Supercomputing Applications

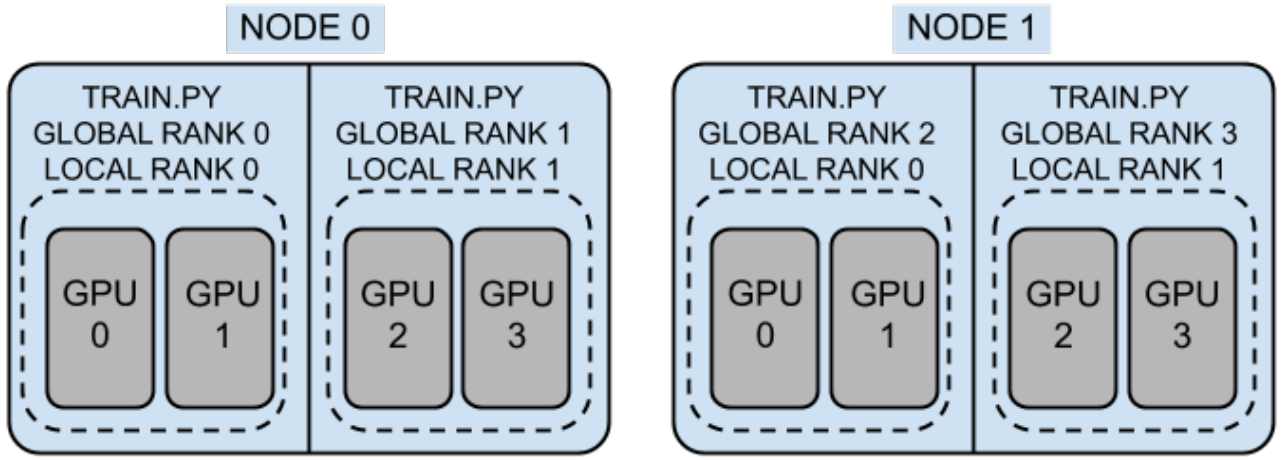
# Why use Distributed Training?




- Data are too large to fit on a machine
- Need complete the task with less time

# How Distributed Training Shortens Training Time

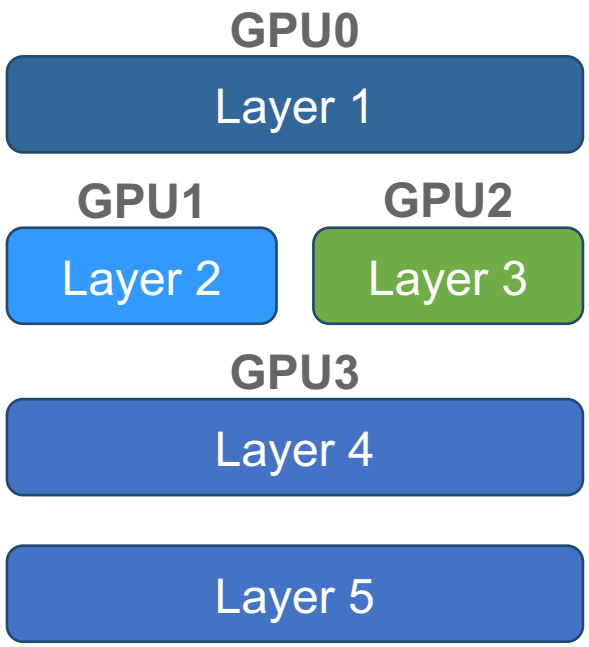
- Storage and compute power are magnified with each added GPU, which reducing training time.
- Every neural network has an optimal batch size, which affects training time. When the batch size is too small, each individual sample has a lot of influence, creating extra noise and delaying the convergence of the algorithm.



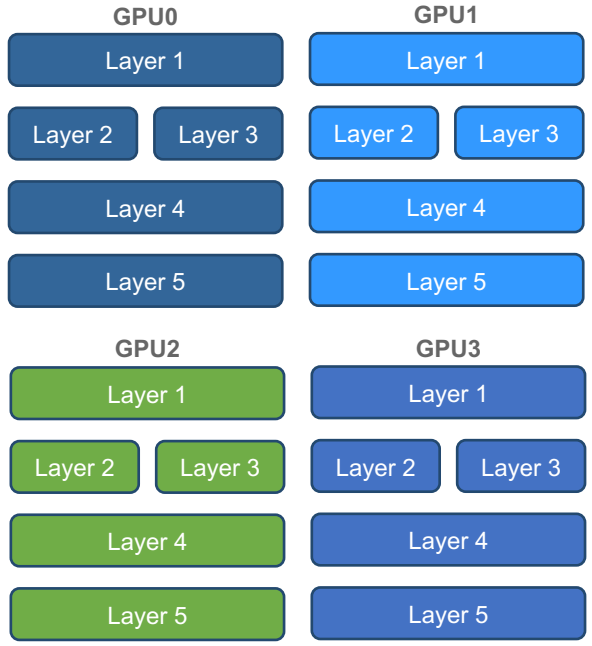
# The Basic Training Workflow

1. A training step begins with preprocessing our data
2. Feed data into the network
3. Predicts the output save intermediates for gradient computation
4. Compare the prediction with the label by computing the loss
5. Apply the chain rule to compute the gradients of the loss function
6. Update the weights based on the gradients
7. Repeat 2-6

# Data and Model Parallelism

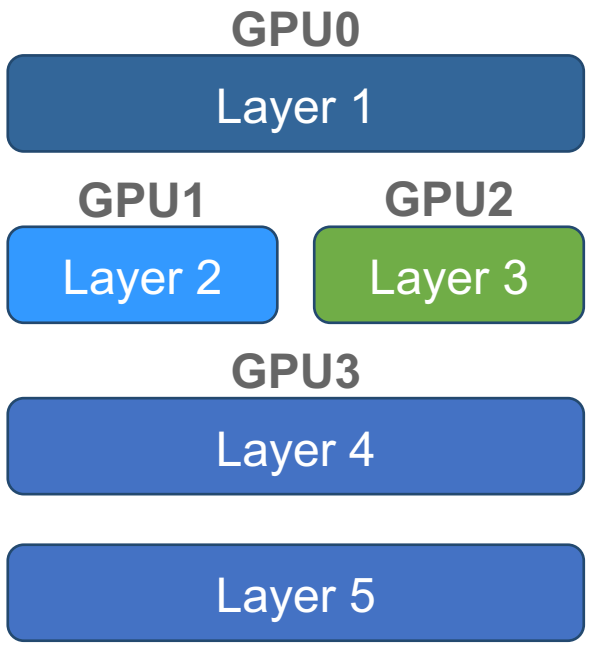



Model Parallelism




Data Parallelism

# Model Parallelism




Model Parallelism

- The model itself is divided into parts
- Trained across different worker nodes
- Very difficult to implement
- Only works well in models with naturally parallel architectures

# Model Parallelism

**TensorFlow**

```
import tensorflow as tf
from tensorflow.keras import layers

with tf.device('GPU:0'):
    layer1 = layers.Dense(16, input_dim=8)

with tf.device('GPU:1'):
    layer2 = layers.Dense(4, input_dim=16)
```

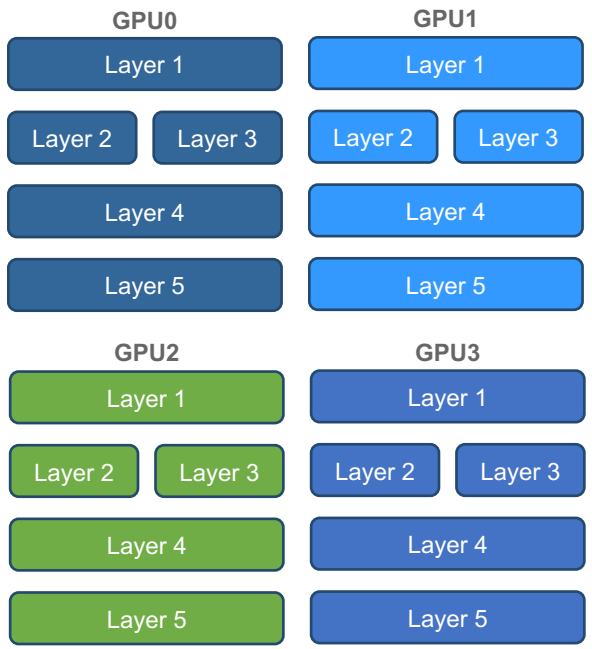
**Pytorch**

```
import torch.nn as nn

layer1 = nn.Linear(8,16).to('cuda:0')

layer2 = nn.Lienar(16,4).to('cuda:1')
```

# Data Parallelism




Data Parallelism

- Scatter our data throughout a set of GPUs or machines
- Perform the training loops in all of them
- Allreduce the gradients synchronously or asynchronously

# Data Parallelism

**TensorFlow**

```
import tensorflow as tf
strategy = tf.distribute.MirroredStrategy()
with strategy.scope():
    model = …
    model.compile(...)
```

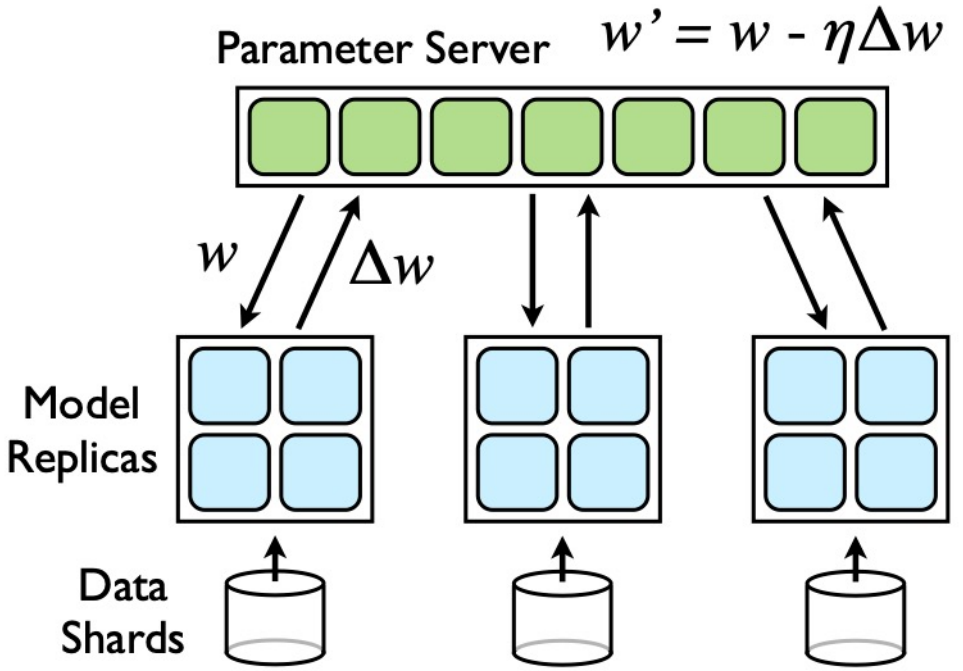
**Pytorch**

```
import torch
model = ...
model = torch.nn.DataParallel(model)
model = model.to(torch.device('cuda'))
```

- *tf.distribute.MirroredStrategy()* and *torch.nn.DataParallel()*
  - create a replica for each available device
  - replicas runs in lock-step & synchronize at each step
- *tf.distribute.MultiWorkerMirroredStrategy()* and *torch.nn.parallel.DistributedDataParallel()*
  - multiple GPUs on multiple machines
- *tf.distribute.CentralStorageStrategy()*
  - only CPU holds the whole model
- *tf.distribute.ParameterServerStrategy()*

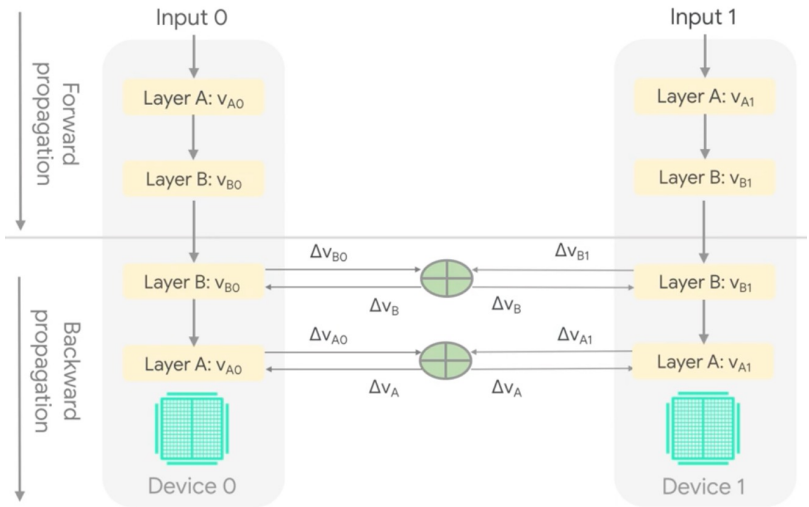
# Parameter Server Strategy

- Parameter servers: hold the parameters of model and are responsible for updating them (global state of model).
- Training workers: run the actual training loop and produce the gradients and the loss from the data.
  - Replicate the model in all of the workers
  - Worker fetches the parameters from PS
  - Worker Performs a training loop
  - Worker sends the gradients back to the PS
  - PS update the model weights



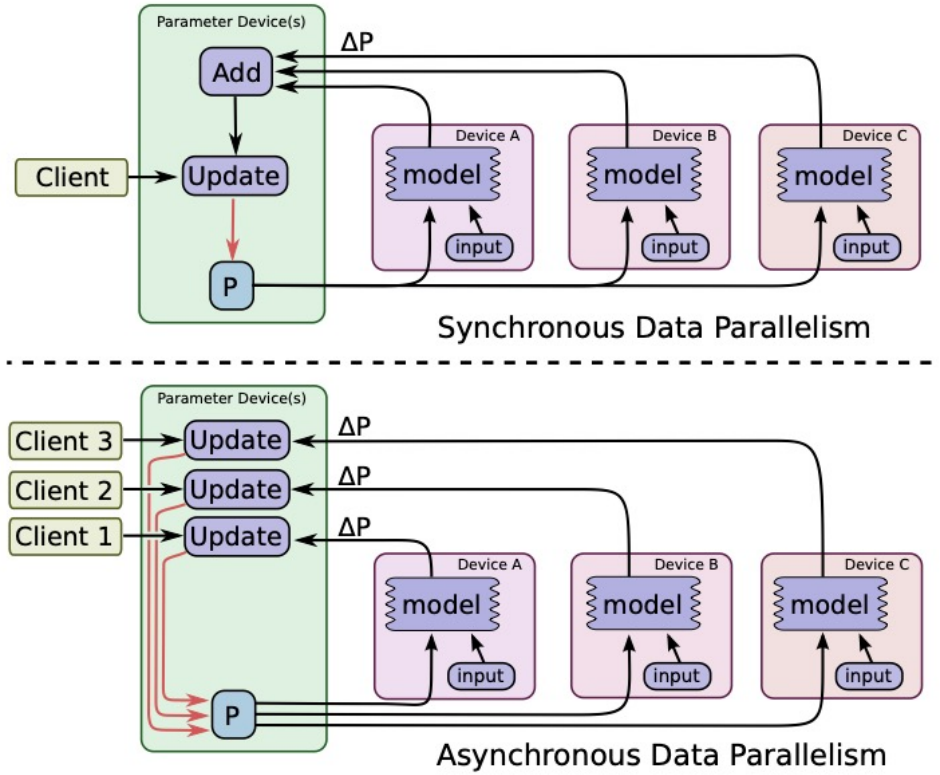
# Synchronous Training

- send different slices of data into each worker/accelerator
- **each device has a full replica of the model**
- workers trained only on a part of the data
- workers all compute a different output and gradients.
- all the devices are communicating with each other
- aggregating the gradients using the **all-reduce algorithm**
- after the gradients are combined, they are sent back to all of the devices
- each device continues updating the local copy of the weights
- the next forward past doesn't begin until all the variables are updated



# Asynchronous Training

- The difference from sync training is that the workers are executing the training of the model at different rates and each one of them doesn't need to wait for the others.
- Synchronous training has a lot of advantages but it can be kind of hard to scale.
  - If we have many small, unreliable and with limited capabilities devices, it's better to use an **async** approach
  - On the other hand, if we have strong devices with powerful communication links, a synchronous approach might be a better choice

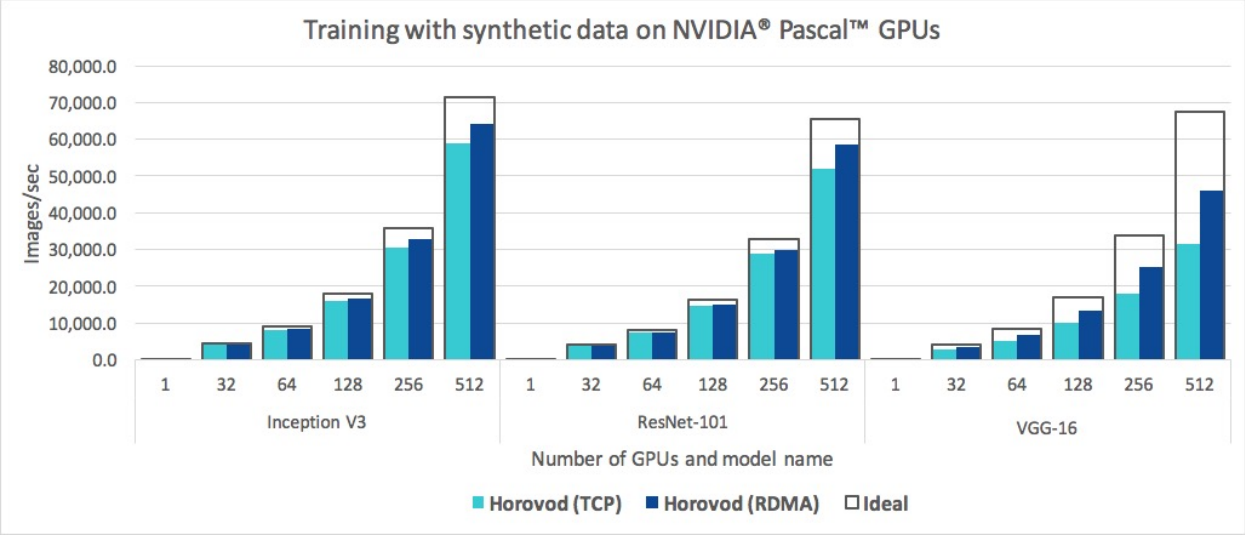


# Horovod

- Horovod is a distributed deep learning training framework for
  - TensorFlow,
  - Keras,
  - PyTorch,
  - Apache MXNet.
- The goal of Horovod is to make distributed deep learning fast and easy to use.

# Horovod

- Internally developed by Uber, engineers found the MPI model to be much more straightforward and require far less code changes than previous solutions such as Distributed TensorFlow with parameter servers.
- Once a training script has been written for scale with Horovod, it can run on a single-GPU, multiple-GPUs, or even multiple hosts without any further code changes.



# Horovod Usage

```
import tensorflow as tf

# Choose the right backend
import horovod.tensorflow as hvd

# Initialize Horovod
hvd.init()

# Pin GPU to be used to process local rank (one GPU per process)
config = tf.ConfigProto()
config.gpu_options.visible_device_list = str(hvd.local_rank())

# Build model...
loss = ...
opt = ...

# Add Horovod Distributed Optimizer
opt = hvd.DistributedOptimizer(opt)

# Add hook to broadcast variables from rank 0 to all other processes during initialization.
hooks = [hvd.BroadcastGlobalVariablesHook(0)]

# Make training operation
train_op = opt.minimize(loss)
```

# Running Horovod

- To run on a machine with 4 GPUs:
  - horovodrun -np 4 -H localhost:4 python train.py
- To run on 3 machines with 4 GPUs each:
  - horovodrun -np 12 -H server1:4,server2:4,server3:4 python train.py

# Now is

Real Case Demo Time!

On HAL System:
/home/shared/distributed-deep-learning-on-hal

Later on GitHub:
https://github.com/ncsa/distributed-deep-learning-on-hal

# Case 01: PyTorch DataParallel Mode

- 01_pytorch_dataparallel.py
- 01_run_pytorch_dataparallel.sb

# Case 02: TensorFlow MirroredStrategy Mode

- 02_tensorflow_mirroredstrategy.py
- 02_tensorflow_mirroredstrategy_run.sb

# Case 03: PyTorch DistributedDataParallel

- 03_pytorch_distributeddataparallel.py
- 03_pytorch_distributeddataparallel_run.sb

# Case 04: TF MultiWorkerMirroredStrategy Mode

- 04_tensorflow_multiworkermirroredstrategy_n1.py
- 04_tensorflow_multiworkermirroredstrategy_n2.py

# Case 05: Horovod MNIST TensorFlow

- 05_horovod_tensorflow_mnist.py
- 05_horovod_tensorflow_mnist_run.sb

# Case 06: Horovod MNIST Pytorch

- 06_horovod_pytorch_mnist.py
- 06_horovod_pytorch_mnist_run.sb


ILLINOIS
NCSA | National Center for
Supercomputing Applications