Distributed Data Processing in the RCC HPC Platform

Trung Nguyen, Ph.D.
ndtrung@uchicago.edu
Research Computing Center
August 7, 2025

You will know

- Understand the key concepts of data parallel processing
- Become familiar with commonly used Python tools for distributed data parallel processing
- Apply the PyTorch distributed data parallel module in illustrative examples

https://github.com/rcc-uchicago/distributed-data-processing.git

RCC Midway Clusters

time for you to log in to **Midway3** …

1. Log in to the login node via SSH, or via ThinLinc

   ssh [your-cnetid]@midway3.rcc.uchicago.edu

2. Clone the github repo for the examples

   git clone https://github.com/rcc-uchicago/distributed-data-processing.git

3. Request an interactive job

   sinteractive -N 1 --ntasks-per-node=8 --account=**rcc-guest**

4. Load the modules and activate the environment

   module load python/miniforge-25.3.0 mpich/4.1.2+gcc-10.2.0

   ulimit –l unlimited

   source activate ddp

No access to Midway3?

1. Clone the github repo for the examples

   git clone https://github.com/rcc-uchicago/distributed-data-processing.git

   cd distributed-data-processing

2. Create your own environment and install the necessary packages

   python3 –m venv my_ddp

   source activate my_ddp

   pip install –r requirements.txt

Distributed Data Processing

General idea: Using multiple processing units to process

- Many files (100 –10,000 or more)
  - Each file contains columns of output data, corresponding to an input parameter set
  - Examples: time series, transaction logs
- Several big files (1 GB – 1 TB)
  - Each file contains structured data
  - Examples: genomics, climate

Why parallelize your data processing?

Need to perform repeated tasks on different datasets or input parameters

Need to scale up your problem size with bigger datasets

Need to meet some deadline

Use the HPC resources more efficiently

Use Case 1: Processing many files concurrently

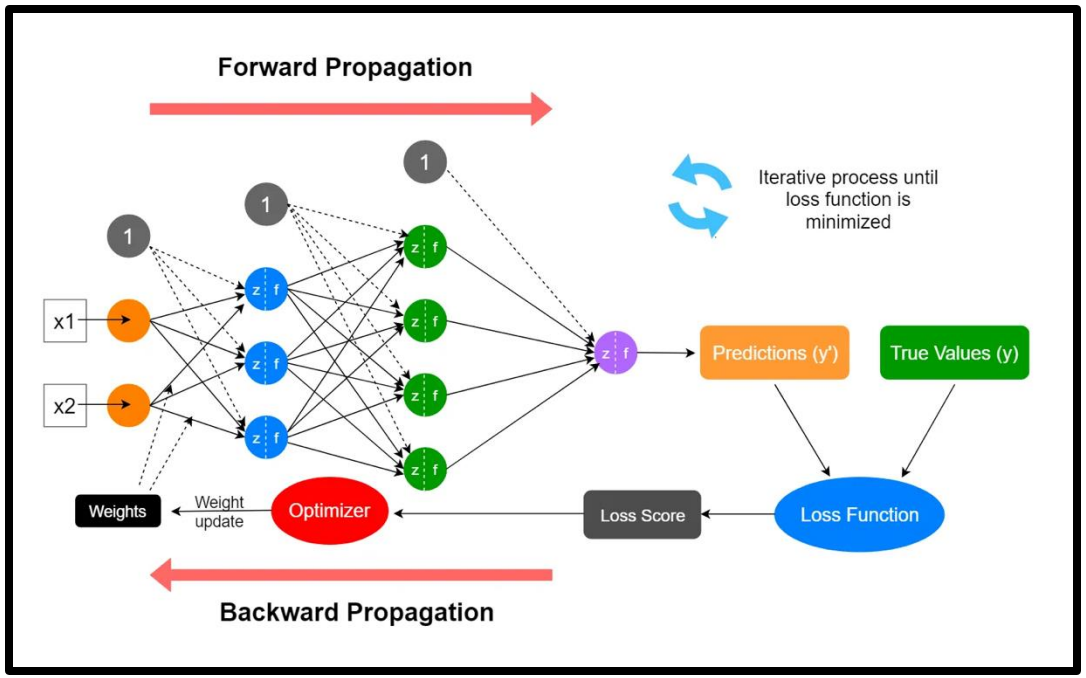
- Researcher needs to scan through thousands of input parameter sets (Help Desk Ticket #57390)
  - For each input parameter set, process input data from a set of files, and write the results to separate output files
- Proposed solution:
  - Divide the list of input parameter sets into equal-sized chunks
    - for each chunk of parameter sets, bind the Python process to a set of CPU cores on a compute node
    - combine the output files for further analysis

Use Case 2: Iterating through a long list

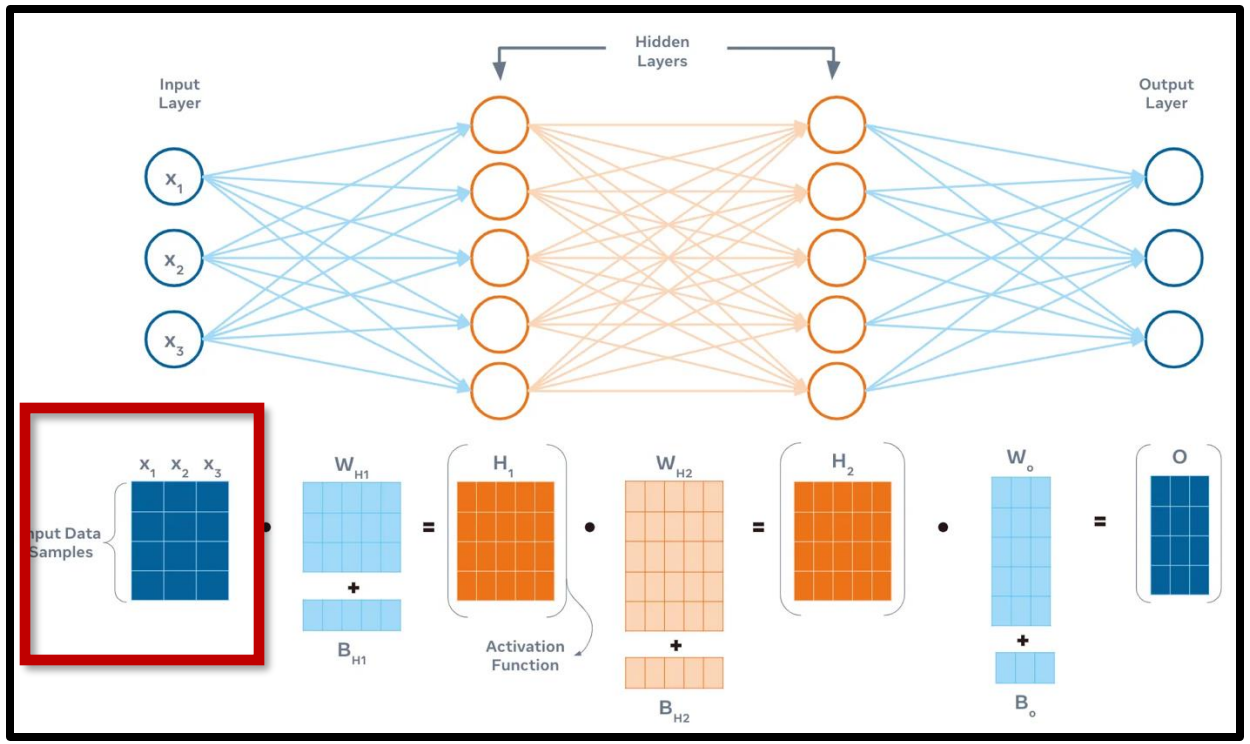
- NSF Collaborator Affiliations Form of a principal investigator (PI)
  - Given the PI's full name, list all the coauthors within a period and their affiliations
- https://github.com/rcc-uchicago/collaborators
  - Search over Google Scholar for the list of PI's coauthors
  - Divide the list of coauthors into equal-sized chunks
    - for each chunk of coauthors, launch a process to search for the affiliation of a coauthor over Google Scholar and/or ORCID
    - combine the results into a single list

Use Case 3: Training a ML model with a big dataset

Deep learning model training



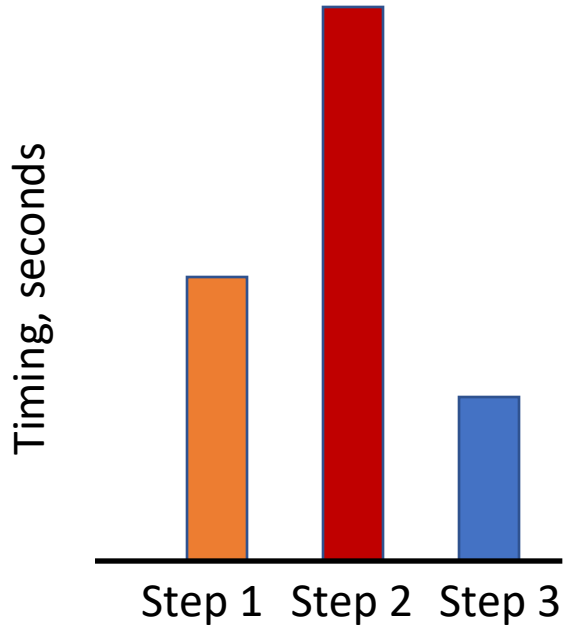
Forward pass



https://medium.com/data-science-365/overview-of-a-neural-networks-learning-process-61690a502fa
https://shivambharuka.medium.com/deep-learning-a-primer-on-distributed-training-part-1-d0ae0054bb1c

Understand your code: Is data processing the bottleneck?

- Identify the bottlenecks in the flow chart of your program – using timers and profilers
- Can the data processing step be parallelized?
  - that is, can the workload be distributed among processing units, aka “workers”?



Amdahl's Law

- Theoretical speedup is limited by the contribution of the non-parallelized parts

$$S(N) = \frac{1}{(1-p) + \frac{p}{N}} \qquad \lim_{N \to \infty} S(N) = \frac{1}{1-p}$$

S = theoretical speedup with N processing units

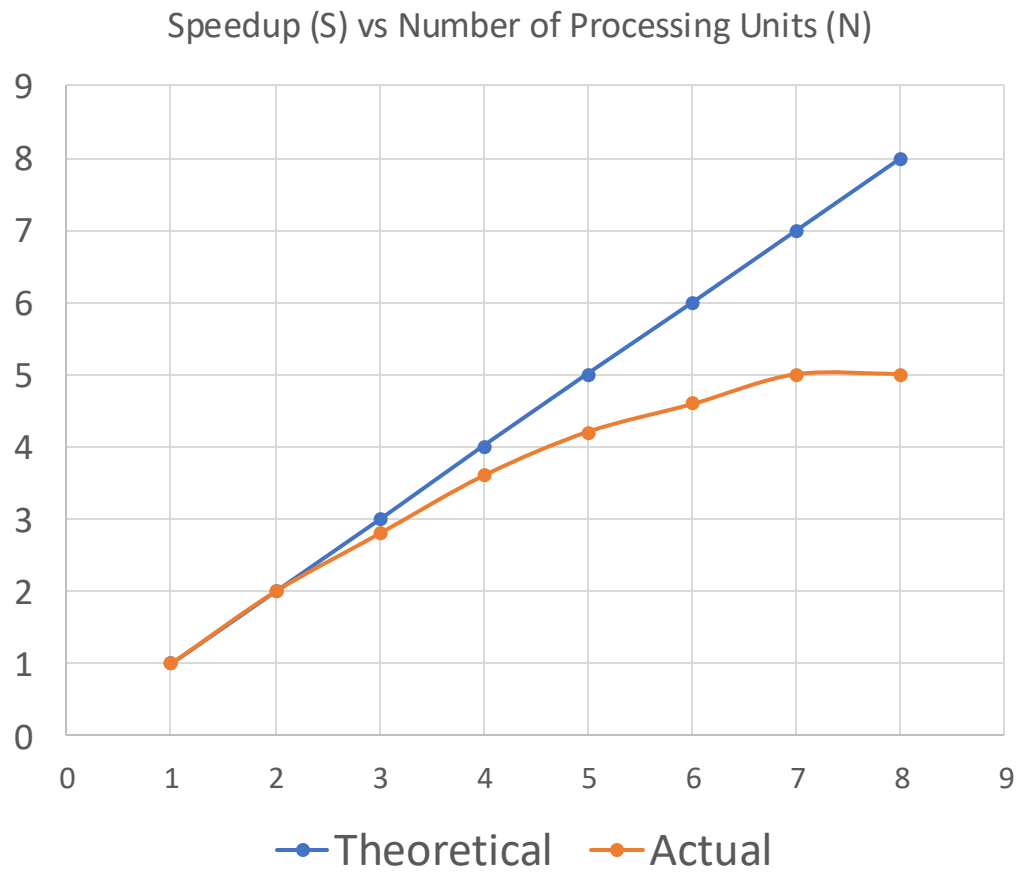
p = time percentage of the parallelized parts

N = theoretical speedup gained for the task with N processing units

What is max of S if p = 80%?

Parallelization performance: Strong scaling

- Parallelization introduces overhead:
  - communication/sync between workers
- Strong scaling analysis show time to solution, or speedup, as a function of number of processing units (threads, or processes)
  - linear scaling: $S_{linear} = t_1/t_N = N$
  - computation vs. communication break-even point
    - P* where speedup stops increasing with P



Know the hardware where your code is running

to decide which parallelization strategies would be optimal

- Single-node configuration
  - Multi-core CPUs: how many physical CPU cores? hardware threading off/on? (lscpu, /etc/cpuinfo) supporting vectorization (avx512)? L1 cache size?
  - Memory
  - GPUs attached? Types? Memory size and bandwidth?
  - Storage: local scratch (on the compute node) or somewhere else (e.g. under /project)
- Multiple-node configuration
  - Interconnect bandwidth: Infiniband?

Midway3 Compute Nodes

to choose the resources for your batch jobs

- Show the partition information:

  scontrol show partition caslake

  sinfo –p caslake

- Show the node information: scontrol show node midway3-xxxx
  - How many physical CPU cores?
  - Memory
  - GPUs attached?

Processes and Threads: Basic Concepts

A process is a program managed by the operating system (OS)

1.operating on separate memory spaces

2.execute a series of instructions (also called **a command queue**), or

3.consisting of an infinite loop waiting for OS events (GUI programs)

4.able to spawn/fork child processes each having separate memory spaces

A thread is a sub-process created and managed by a process

1. sharing the memory space with peer threads in the same process

2. able to spawn/fork child threads

Python is a program that is by default run with a single process with a single thread

python your_script.py

Multithreading and Multiprocessing Programming Models

- **Multithreading**:
  - the main thread spawns/forks multiple threads
  - each thread access to the data in the shared memory pool
  - each thread executes the instructions in order, may sync with other threads
  - threads are terminated (joined) when done
- **Multiprocessing**:
  - create child processes, or launch peer processes
  - each process allocates data in its memory space
  - each process executes the instructions in order (**queue**), may send/receive data among the processes
  - processes are closed (finalized) when done

A Python process can create multiple threads or multiple processes.

Multithreading is not suitable for data processing

- Performance gain with multithreading is generally prohibited by the Global Interpreter Lock (GIL) used by Python
  - to avoid write conflicts
  - to prevent memory leaks (object mem allocation and release)

- While it is possible to use the multithreading module to read/write files concurrently, the data processing part is limited by the GIL.

Serial processing (Exercise 1)

```
python3 generate_data_files.py –-num-files=100
python3 processing_serial.py
```

- Generate the data files
- Loop through all the files
  - invoke the custom processing function
- Note the added functions for measuring the elapsed time
- Serve as the baseline for correctness and performance

Parallel processing with multiprocessing Queue (Exercise 2)

python3 processing_multi.py

- Each worker
  - creates a command queue object
  - applies a custom processing function to assigned data and puts the result to the queue (line 39)
  - creates a child process and executes the commands in the queue (41)
  - stores all the queues with the returned results
- Terminate all the child processes
  - join() into the parent process (lines 45-46)
- Combine the results across the queues (lines 49-55)

Finish the TODO part of the Python code

Parallel processing with multiprocessing Pool (Exercise 3)

python3 processing_multi.py

- Each worker creates a process with the custom processing function
- Combine the results across the processes

Switch to using Pool in the Python code

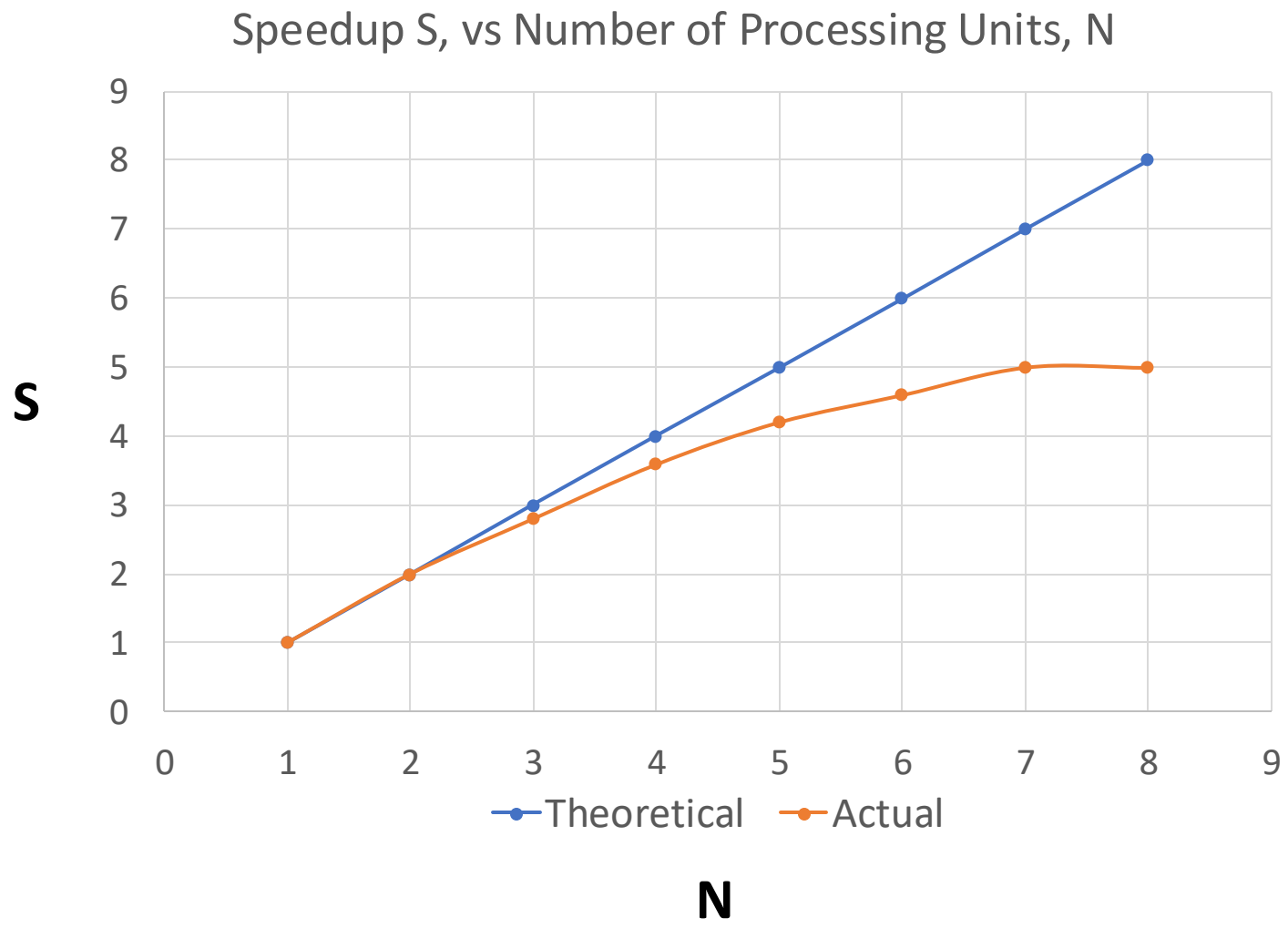
Measure the elapsed time

```
python3 generate_data_files.py –-num-files=1000 –-num-rows=32000
python3 processing_multi.py –n 8
```

- Vary the number of files, or number of rows in each file
- Vary the number of processes (N)

Record the elapsed times as a function of N

Strong scaling performance for a fixed size problem



Quiz 1

Strong scaling shows how the performance of a parallel code varies

A) as the number of processing units (P) increases for a fixed problem size per processing unit (N/P).

B) as the number of processing unit (P) increases for a fixed problem size (N).

C) when the input and output data are strongly correlated.

Quiz 2

What could be the reason for the strong scaling curve to become plateau as the number of processes increases?

A) The communication overhead between the processes increases.

B) The computation workload per process decreases.

C) We do not use GPUs for the computation yet.

D) We do not have fast enough CPUs.

E) A and B.

F) B and C.

Parallel processing with PySpark (Exercise 4)

python3 processing_pyspark.py

- Create a PySpark session
- Create a Resilient Distributed Dataset (RDD) object from the session's context with all the files and a number of slices (each for a worker)
- The RDD object maps the custom processing function to the files
- The RDD object collects the results across the workers

Vary the number of workers and measure the elapsed time

Parallel processing with Dask (Homework)

python3 processing_dask.py

- Create a LocalCluster object with a given number of workers
- Create a Client object with the cluster object
- The client submits the custom processing function to the files
- The client gathers the results across the workers

Vary the number of workers and measure the elapsed time

Parallel processing across multiple nodes with Message Passing Interface (MPI)

- Message Passing Interface (MPI) is a specification (programming model) for exchange data between processes
  - communications: point-to-point, collective (one-to-all, all-to-all), one-sided
  - C/C++/Fortran bindings
- Different implementations (vendors): OpenMPI, Intel MPI and MPICH
- Allow you to compile C/C++/Fortran codes with wrappers like mpicc, mpicxx, and mpifort

mpi4py module

- work as a wrapper for an underlying Message Passing Interface (MPI) library (OpenMPI, MPICH or Intel MPI)
- mpirun launches multiple Python instances

```
from mpi4py import MPI


comm = MPI.COMM_WORLD
rank = comm.Get_rank()
size = comm.Get_size()
...
# assign a chunk of the data to this rank
```

Parallel processing with MPI (Exercise 5)

python3 processing_multi_nodes.py

- Each MPI process (rank) is assigned with a subset of the data files
- Combine the results across the MPI ranks into the first rank

Finish the TODO part of the Python code

Midway3: ulimit –l unlimited (if getting errors with UCX workers init)

mpi4py provides MPIPoolExecutor similar to multiprocessing.Pool

```
from mpi4py.futures import MPIPoolExecutor


with MPIPoolExecutor() as executor:
  futures = executor[submit([processing_file, f, channel_id) for f in files]
  results = [future.result for future in futures]
```

Homework: Using MPIPoolExecutor

PySpark and Dask across multiple nodes

- Need to launch a scheduler on one of the compute nodes
- then launch multiple workers on the all the nodes, specifying the host name where the scheduler is running

Processing a big file with multiple processes (Exercise 6)

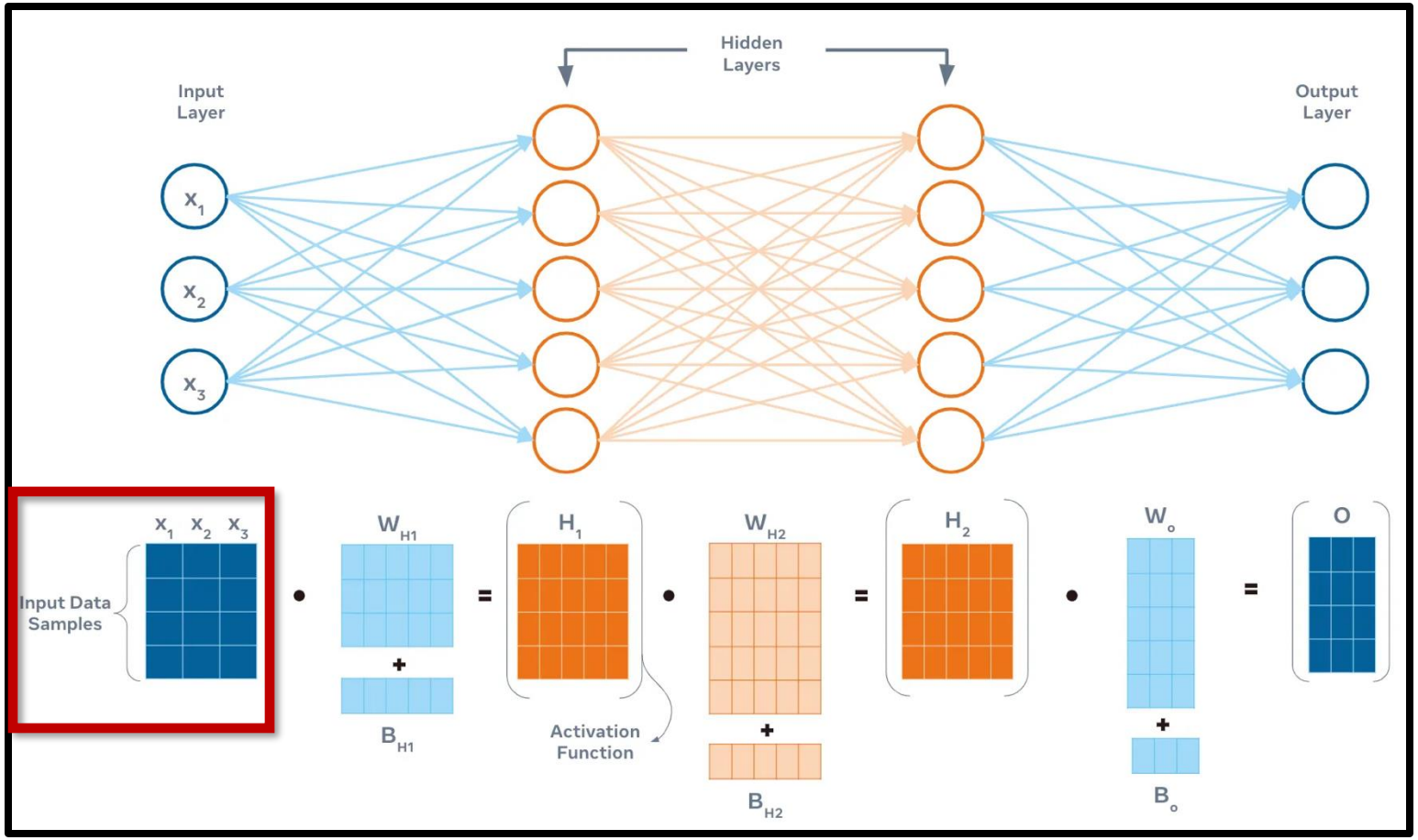
python3 processing_hdf5.py

- Multiple processes can open the same file to read
  - each process works on a separate segment of the file
- Launch multiple processes via either multiprocessing, or mpi4py

Vary the size of the HDF5 file and number of workers, and measure the elapsed time

Distributed Data Training with A Large Data Set

Forward pass



https://shivambharuka.medium.com/deep-learning-a-primer-on-distributed-training-part-1-d0ae0054bb1c

Distributed Data Parallel

- Model training requires iterating through the entire dataset multiple times (epochs)
- Data parallel training splits the entire dataset across multiple workers:
  - each worker holds a copy of the model (i.e. its weights and hyperparameters)
  - for each batch, each worker works on its own partition of the dataset, calculates the loss function and the gradients w.r.t to the model weights
    - the gradients are then synchronized across the workers (via allreduce())
    - the model parameters are then synced across the workers

Distributed Data Parallel with PyTorch (Example)

python3 ml_training_ddp.py

```
import torch.distributed as dist
import torch.multiprocessing as mp
```

```
...
world_size = torch.cuda.device_count()
mp.spawn(train, args=(world_size, batchsize, num_epochs), nprocs=world_size, join=True)
```

```
# inside train(rank, world_size, batchsize, num_epochs)

  # Setup the training group
   setup(rank, world_size)

   # Set device
   device = torch.device(f'cuda:{rank}')
```

torch.multiprocessing wraps multiprocessing, creates world_size processes and invokes the train function in each process.

Distributed Data Parallel with PyTorch

python3 ml_training_ddp.py

```
from torch.utils.data import DataLoader, DistributedSampler

...
# Load dataset with DistributedSampler
train_dataset = torchvision.datasets.CIFAR100(root='./data', train=True, download=False, transform=transform)
test_dataset = torchvision.datasets.CIFAR100(root='./data', train=False, download=False, transform=transform)

train_sampler = DistributedSampler(train_dataset, num_replicas=world_size, rank=rank)
test_sampler = DistributedSampler(test_dataset, num_replicas=world_size, rank=rank, shuffle=False)

train_loader = DataLoader(train_dataset, batch_size=batchsize, shuffle=False, num_workers=4, sampler=train_sampler, pin_memory=True)
test_loader = DataLoader(test_dataset, batch_size=batchsize, shuffle=False, num_workers=4, sampler=test_sampler, pin_memory=True)
...
```

DistributedSampler partitions the dataset into chunks (or shards).

DataLoader uses DistributedSampler to load the corresponding data chunks.

Each loader can also launch multiple child workers.

Distributed Data Parallel with PyTorch

```
# Load ResNet50 model and offload to the GPU
model = models.resnet50(weights=None)
model.fc = nn.Linear(model.fc.in_features, 100) # CIFAR-100 has 100 classes
model = model.to(device)

# Wrap model in DistributedDataParallel:
model = nn.parallel.DistributedDataParallel(model, device_ids=[rank])
```

DistributedDataParallel takes the model to individual ranks and handles the gradient synchronization between them.

Distributed Data Parallel is Scalable

- Distributed data parallel is advantageous for scaling up the size of the datasets
  - just need to increase the number of processing units and GPUs.
- The communication overhead between the processes for the model weights synchronization can be mitigated by
  - fast network bandwidth between the processes
  - fast communication between the GPUs in a node (via NVLink)

Quiz 3

Which Python packages can be used for distributed data parallel processing?

A) multithreading
B) multiprocessing
C) PySpark
D) mpi4py
E) B, C and D
F) All of the above

Quiz 4

What are the challenges with distributed data parallel processing?

A) Selecting the suitable modules
B) Configuring for multiple hosts
C) Having hardware constraints (e.g. few CPU cores and no GPUs)
D) Having poor performance than the serial version
E) Debugging the parallel code
F) None

Summary

- Distributed data processing is beneficial for
  - many small files
  - big files
- Popular Python packages
  - multiprocessing
  - mpi4py
  - PySpark and Dask
- Some exercises and examples

https://github.com/rcc-uchicago/distributed-data-processing.git

help@rcc.uchicago.edu
ndtrung@uchicago.edu