HIGH PERFORMANCE RESEARCH COMPUTING

ACES: Introduction to HPC and AI for Faculty and Researchers

Distributed AI Model Training on NVIDIA GPUs

10/07/2025

Zhenhua He and Dinesh S. Devarajan

High Performance
Research Computing
DIVISION OF RESEARCH

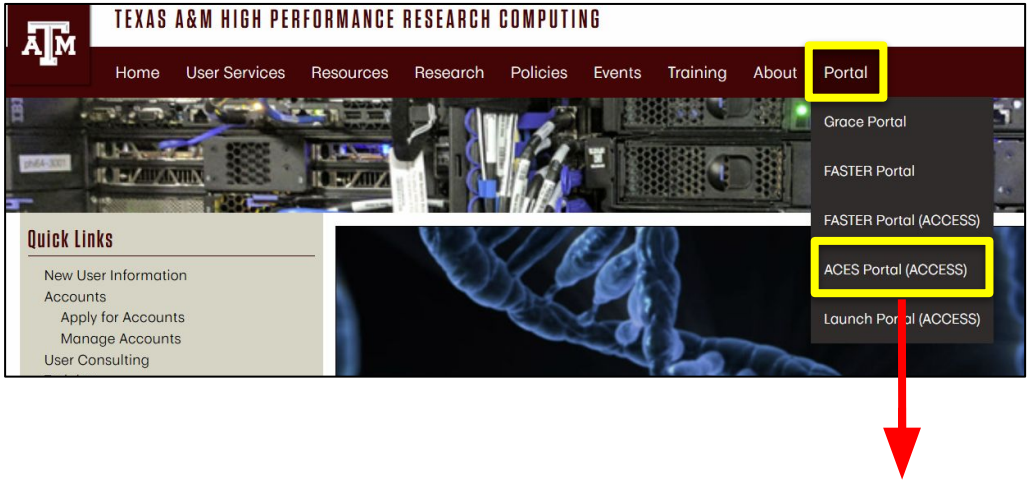
ACES
ACCELERATING COMPUTING FOR EMERGING SCIENCES

Outline

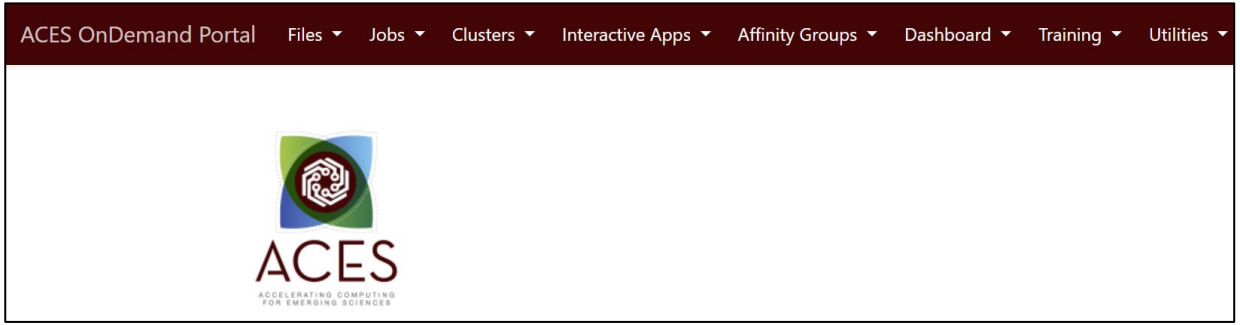
- Lab 0: Virtual Environment Set up on HPC cluster
- Lab 0 (cont.): Introduction to DL in JupyterLab
- Lab 1: Transition from CPU to a single GPU
- Lab 2: Scaling from one GPU to multiple GPUs on a single node
- Lab 3: Extending to multi-node distributed training

Lab 0. Virtual Environment Setup

ACES Portal

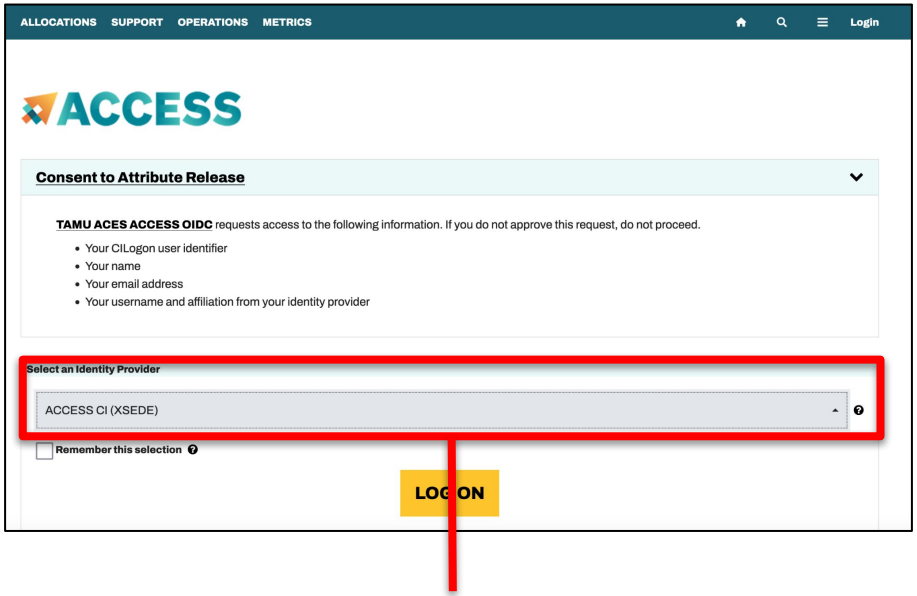


ACES Portal https://portal-aces.hprc.tamu.edu/ is the web-based user interface for the Launch cluster

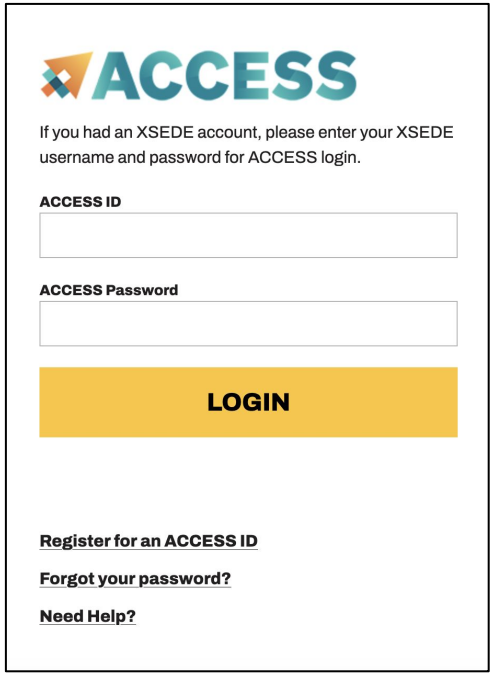


Open OnDemand (OOD) is an advanced web-based graphical interface framework for HPC users

Accessing ACES via the Portal (ACCESS)



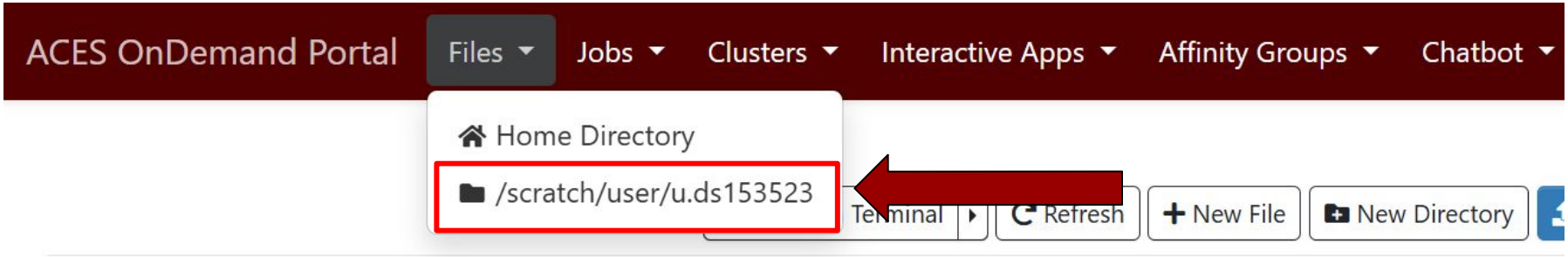
Select the Identity Provider appropriate for your account.



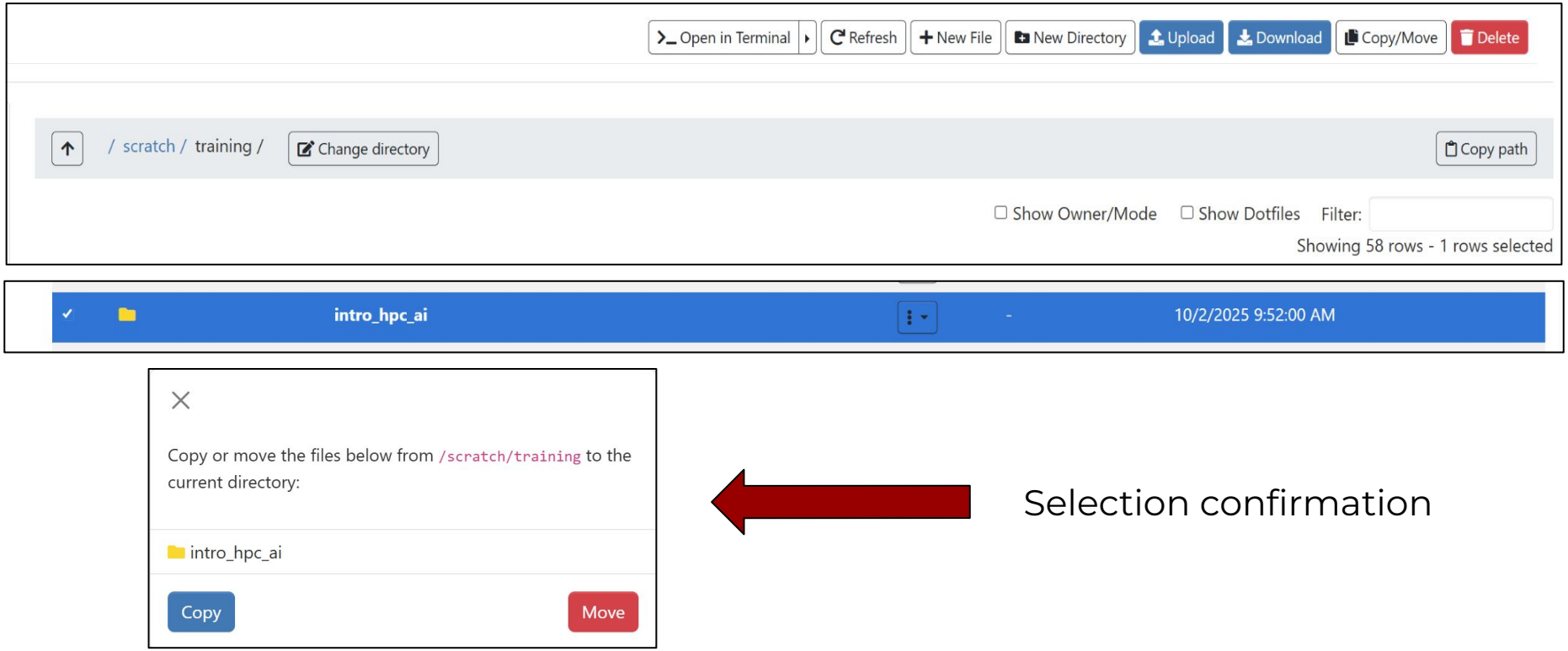
Log in using your ACCESS or institutional credentials.

Accessing File Manager

Click on “Files” menu → /scratch/user/USERNAME

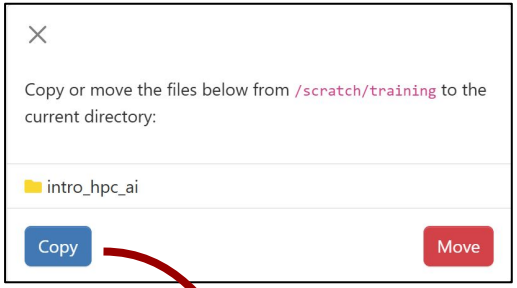


Copying the training materials

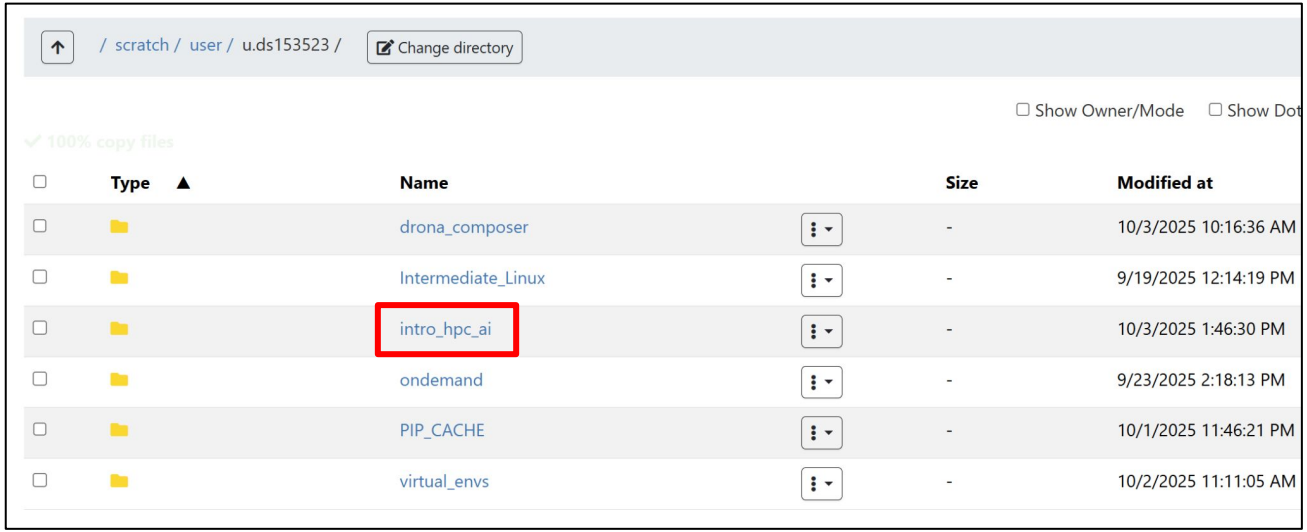


Selection confirmation

Copying the training materials (cont.)

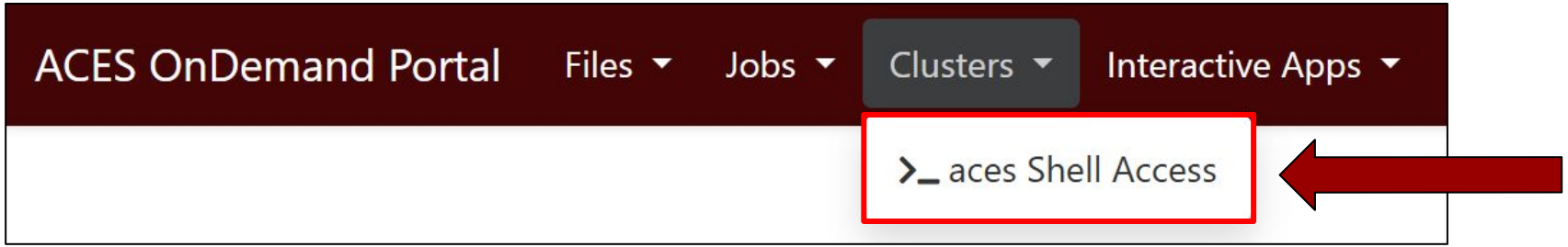


Copying from scratch/training to scratch/user/USERNAME



Get a Shell on ACES

Click on “Clusters” menu → _ aces Shell Access



Success!

Welcome to the ACES login node.
Check which login node you are on and type it in the chat.

```
Your current disk quotas are:
Disk                          Disk Usage      Limit    File Usage      Limit
/home/u.ds153523                     84K      10.0G            17      10000
/scratch/user/u.ds153523            793M       1.0T            47     250000
/scratch/group/p.tra220029.000     12.5G       1.0T         10286     500000
Type 'showquota' to view these quotas again.
[u.ds153523@aces-login2 ~]$
```

ModuLair Framework

- The ModuLair Framework is designed to streamline the management of Python environments on HPC clusters.

  Core Functionalities:

  - ➢ Creating Environments (`create_venv`)
  - ➢ Activating Environments (`activate_venv`)
  - ➢ Listing Environments (`list_envs`)
  - ➢ Deleting Environments (`delete_venv`)

Short course: Setting up coding environments for HPC (Nov. 21, 2025)

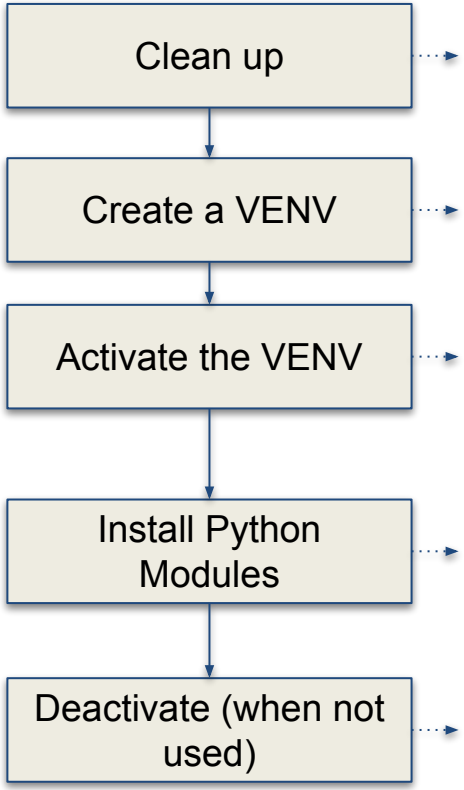
hprc.tamu.edu/kb/Software/ModuLair

Advantages of ModuLair

- ModuLair builds virtual environments in $SCRATCH (at $SCRATCH/virtual_envs) instead of using $HOME to avoid filling your home directory space quickly.

- ModuLair automatically records the toolchain and Python modules used to create each environment, so you don’t need to keep track of them yourself.

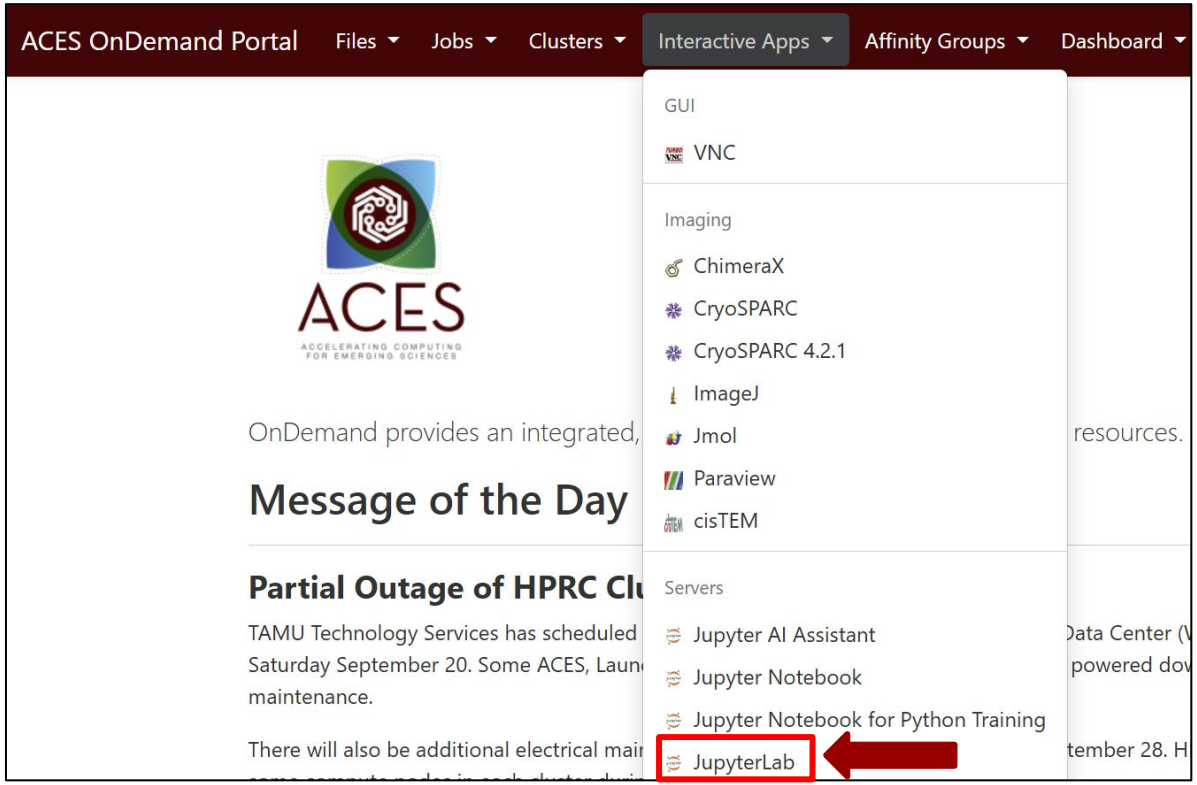
ModuLair



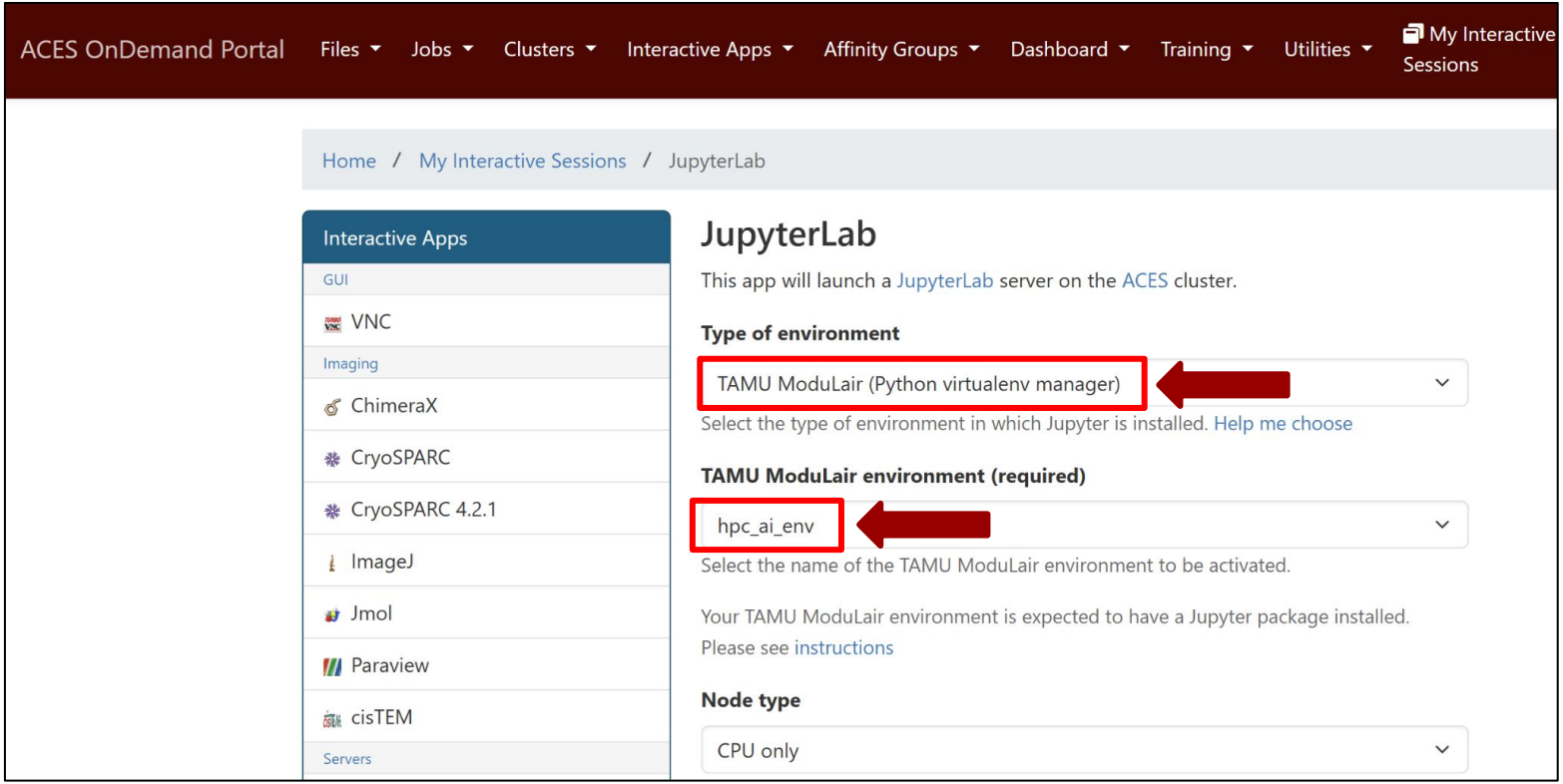

```
# clean up
cd $SCRATCH/intro_hpc_ai/notebooks
module purge

# create a Python virtual environment
create_venv hpc_ai_env -d "Environment for Intro to HPC
and AI" -t "GCCcore/12.2.0 Python/3.10.8"

# activate the virtual environment
source activate_venv hpc_ai_env

# install required packages
pip3 install jupyter
pip3 install pandas matplotlib
pip3 install scikit-learn
pip3 install torch torchvision \
  --index-url https://download.pytorch.org/whl/cu126

# deactivate the virtual environment
deactivate
```

The training materials are also available at https://github.com/happidence1/Intro_HPC_AI.git

Go to JupyterLab Page



JupyterLab Page



Other fields:

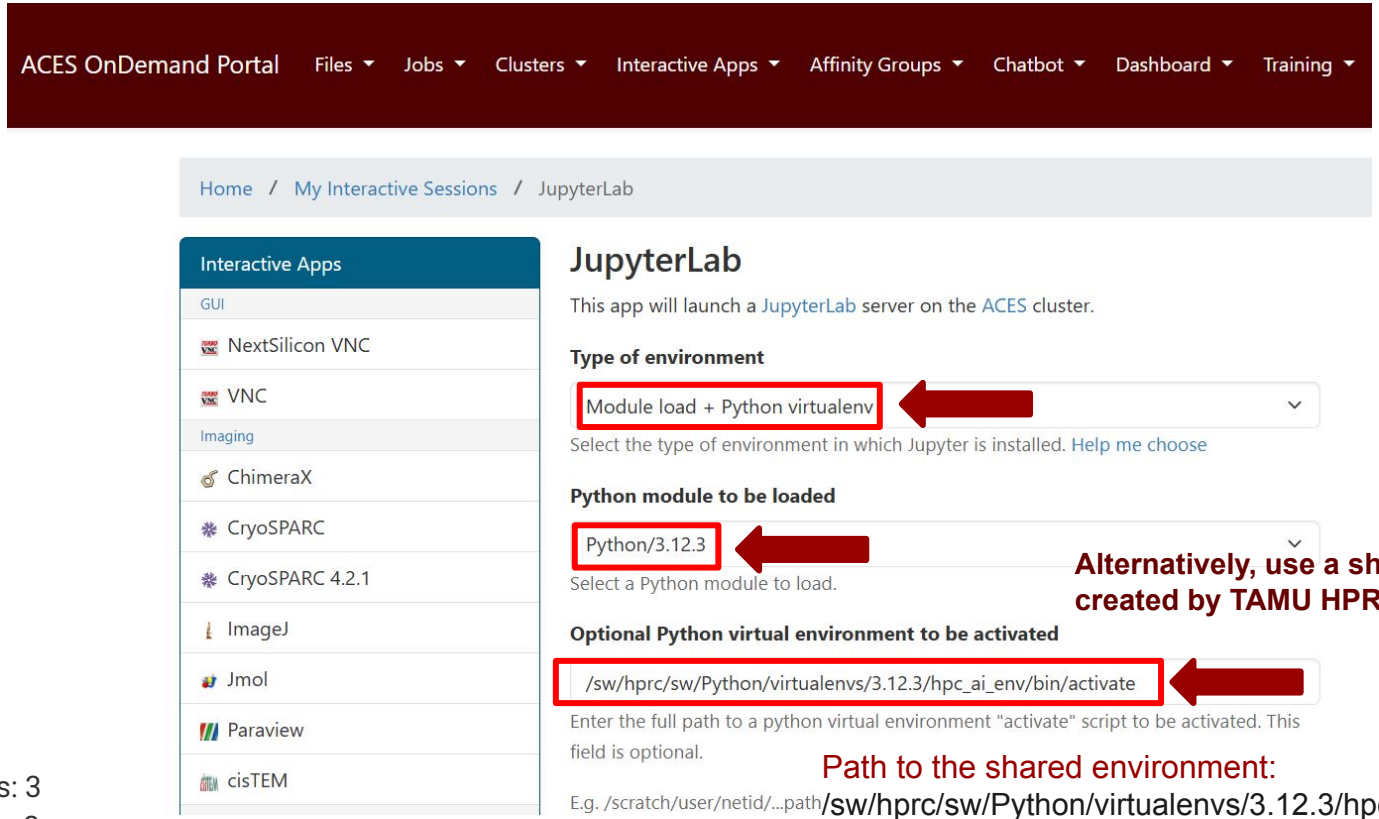
Node Type: **CPU**

Number of hours: 3

Number of cores: 3

Total memory (GB): 10

If your venv fails, please use a shared one

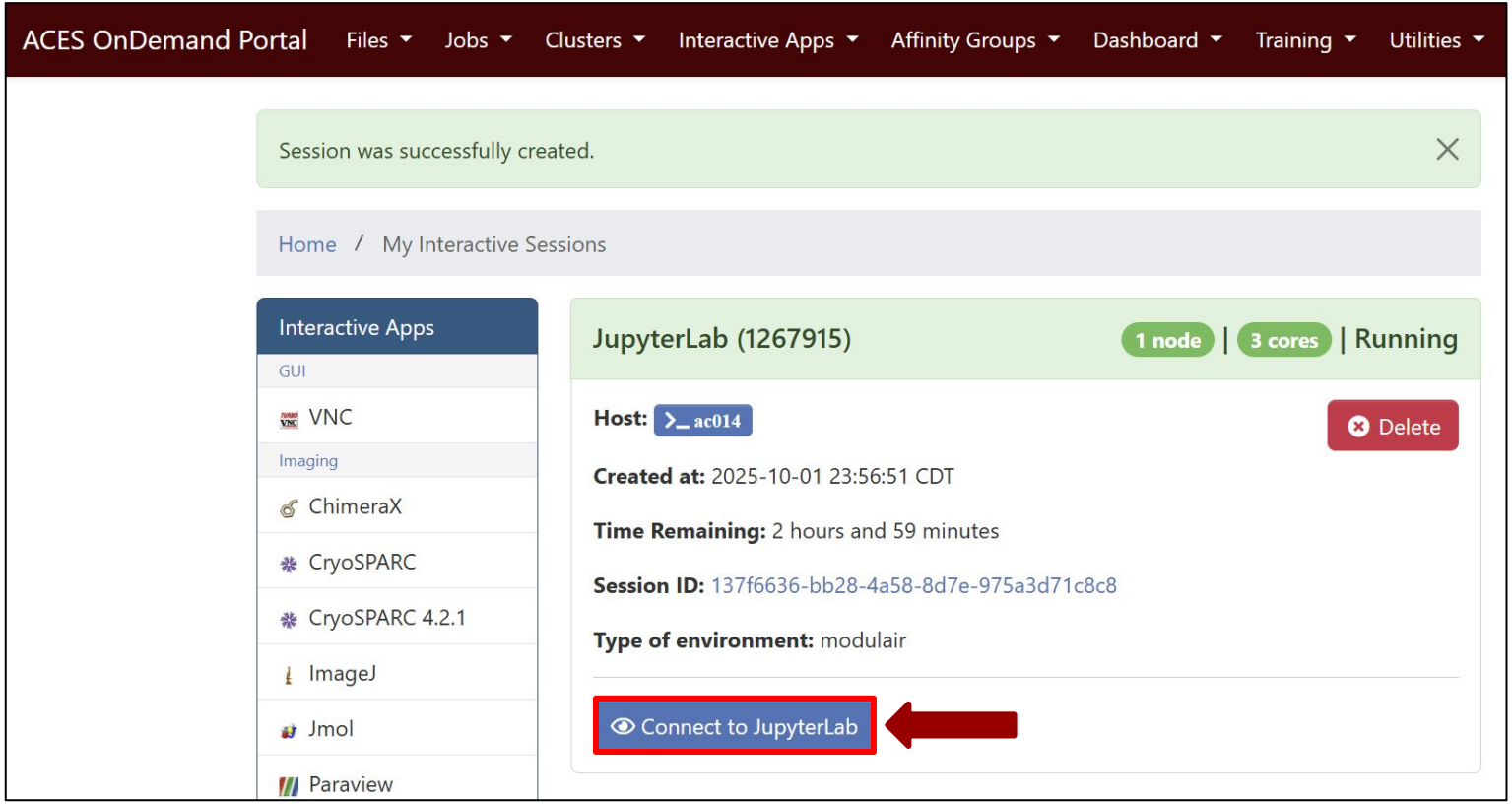


Alternatively, use a shared environment created by TAMU HPRC for this workshop

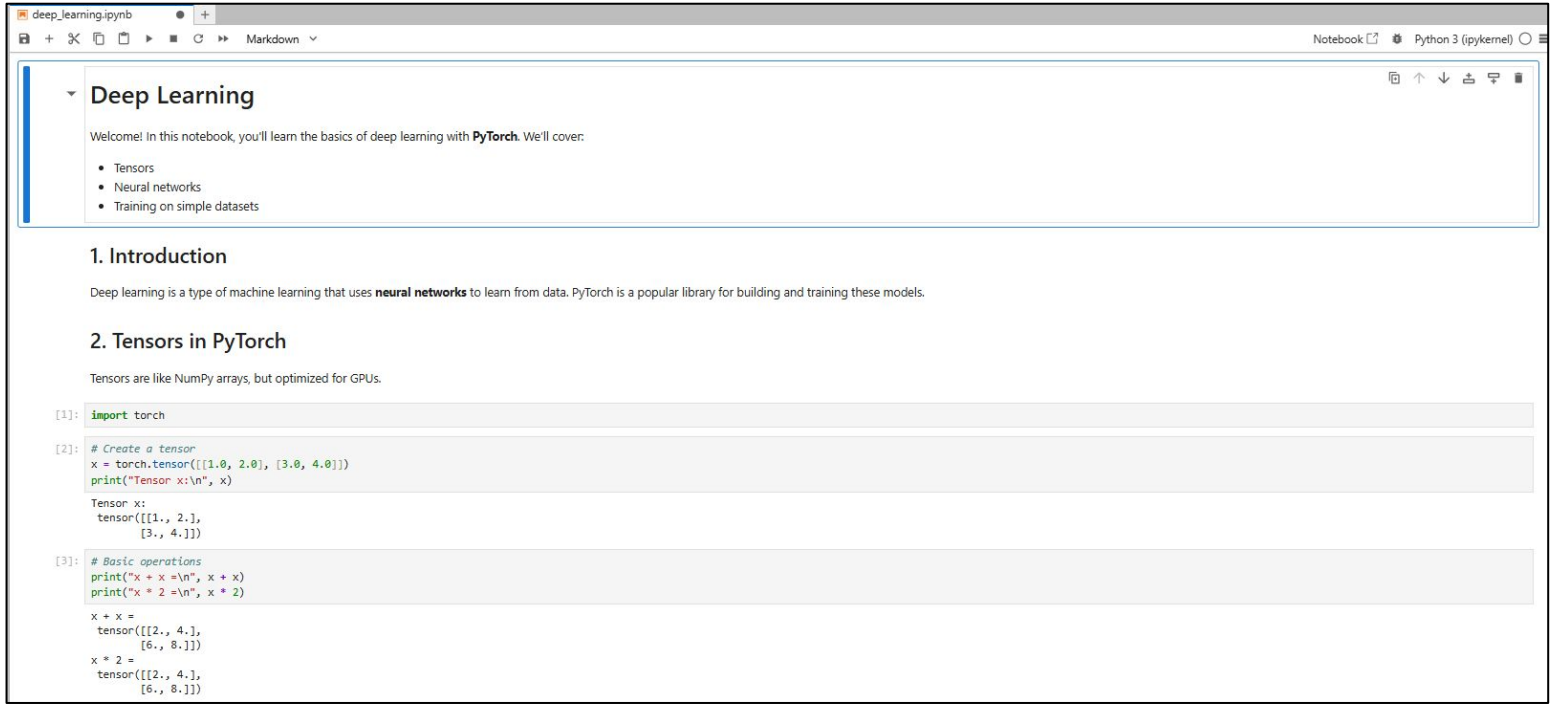
Path to the shared environment:
/sw/hprc/sw/Python/virtualenvs/3.12.3/hpc_ai_env/bin/activate

Other fields:
Node Type: **CPU**
Number of hours: 3
Number of cores: 3
Total memory (GB): 10

Connect to JupyterLab



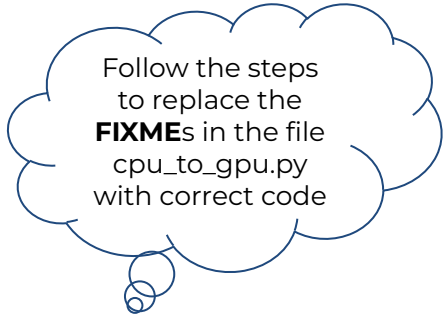
Lab 0 (cont.). Intro to DL in JupyterLab



Lab 1. Transition from CPU to a single GPU

1. Define the device

In cpu_to_gpu.py, add this near the top:




```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

2. Move the model to GPU

Change:

```
model = Net()
```

to:

```
model = Net().to(device)
```

3. Move data and labels to GPU during training

Inside the training loop, change:

```
outputs = model(data)
loss = criterion(outputs, target)
```

to:

```
data, target = data.to(device), target.to(device)

outputs = model(data)
loss = criterion(outputs, target)
```

4. Move data and labels to GPU during evaluation

Inside the evaluation loop, change:

```
outputs = model(data)
loss = criterion(outputs, target)
```

to:

```
data, target = data.to(device), target.to(device)
outputs = model(data)
loss = criterion(outputs, target)
```

5. Handle Lazy Initialization Layer

In Net.forward, the fully connected layer is created lazily. Modify:

```
self._fc = nn.Linear(self.flatten_dim, 10)
```

to:

```
self._fc = nn.Linear(self.flatten_dim, 10).to(x.device)
```

6. Modify SLURM Script for GPU

In cpu_to_gpu.slurm,

```
#SBATCH --job-name=gpu_job  # Change the job name to be gpu related
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=1
#SBATCH --partition=gpu     # Change the partition to be gpu
#SBATCH --gres=gpu:h100:1    # Request 1 h100 GPU
#SBATCH --cpus-per-task=8   # Set number of CPU cores per task (e.g., 8)
#SBATCH --mem=32G
#SBATCH --time=01:00:00
##SBATCH --reservation=training

module load WebProxy
source activate_venv hpc_ai_env

# ---- Run Training ----
# use modified GPU training script
python cpu_to_gpu.py
```

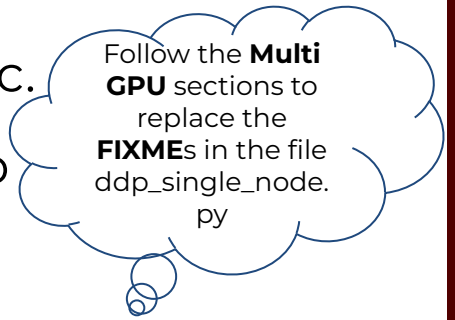
After completing all the steps above, go back to your terminal and run:

```
cd $SCRATCH/intro_hpc_ai/exercises/1.CPU_to_GPU
sbatch cpu_to_gpu.slurm
```

Lab 2. Scaling from one GPU to multiple GPUs on a single node

1. Imports

- **Single GPU** only torch, torch.nn, torch.optim, etc.
- **Multi GPU** adds distributed training modules to ddp_single_node.py




```
import torch.distributed as dist
from torch.nn.parallel import DistributedDataParallel as DDP
from torch.utils.data import distributed
```

2&3. Process group initialization

- **Single GPU** No process group (runs as one process).
- **Multi GPU** Adds setup/cleanup for distributed training.

```
def setup(rank, world_size):
    os.environ['MASTER_ADDR'] = '127.0.0.1'
    os.environ['MASTER_PORT'] = '29500'
    dist.init_process_group("nccl", rank=rank, world_size=world_size)
    torch.cuda.set_device(rank)

def cleanup():
    dist.destroy_process_group()
```

4. Device setup

- **Single GPU**

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

- **Multi GPU** Each process explicitly binds to a GPU based on rank

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

5. Dataset handling

- **Single GPU** Normal DataLoader with shuffle=True

```
train_loader = DataLoader(dataset, batch_size=64, shuffle=True)
```

- **Multi GPU** Use DistributedSampler so each process gets a unique shard of data.

```
sampler = distributed.DistributedSampler(dataset,
   num_replicas=world_size, rank=rank)
loader = DataLoader(dataset, batch_size=64, sampler=sampler)
```

6. Model wrapping

- **Single GPU** Just moves the model to device.

```
model = Net().to(device)
```

- **Multi GPU** Wraps the model in DDP.

```
model = Net().to(device)
model = DDP(model, device_ids=[rank])
```

7. Training loop

- **Single GPU** Standard training loop — one process prints progress.
- **Multi GPU**
  Calls sampler.set_epoch(epoch) at the start of each epoch to reshuffle across processes and avoid bias.

```
sampler.set_epoch(epoch)
```

Only rank 0 (the main process) prints logs

```
if batch_idx % 100 == 0 and rank == 0:
    print(...)
```

End-of-epoch stats also restricted to rank == 0.

8. Launch training

- **Single GPU**

```
python single_gpu.py
```

- **Multi GPU** Use torch.multiprocessing.spawn to launch one process per GPU

```
if __name__ == "__main__":
    world_size = torch.cuda.device_count()
    torch.multiprocessing.spawn(train, args=(world_size,),
nprocs=world_size, join=True)
```

9. Modify SLURM Script for Multi GPU

Follow the instructions in ddp_single_node.slurm and modify it. After completing all the steps above, go back to your terminal and run:

```
cd $SCRATCH/intro_hpc_ai/exercises/2.SingleGPU_to_MultiGPU
sbatch ddp_single_node.slurm
```

Lab 3. Extending to multi-node distributed training

1&2. DDP Setup Changes

- **Single node**

```
def setup(rank, world_size):
    os.environ['MASTER_ADDR'] = '127.0.0.1'
    os.environ['MASTER_PORT'] = '29500'
    dist.init_process_group("nccl", rank=rank, world_size=world_size)
    torch.cuda.set_device(rank)
```

- **Multi node** In ddp_multi_node.py

```
def setup():
    rank = int(os.environ.get("RANK", os.environ.get("SLURM_PROCID", 0)))
    local_rank = int(os.environ.get("LOCAL_RANK", os.environ.get("SLURM_LOCALID", 0)))
    world_size = int(os.environ.get("WORLD_SIZE", os.environ.get("SLURM_NTASKS", 1)))

    if "MASTER_ADDR" not in os.environ:
        os.environ["MASTER_ADDR"] = os.environ.get("SLURM_NODELIST", "127.0.0.1")
    if "MASTER_PORT" not in os.environ:
        os.environ["MASTER_PORT"] = str(12000 + os.getpid() % 10000)

    dist.init_process_group("nccl", rank=rank, world_size=world_size)
    torch.cuda.set_device(local_rank)
```

Follow the **Multi node** sections to replace the **FIXME**s in the file ddp_multi_node.py

Key differences

- Single-node: rank and world size explicitly passed.
- Multi-node: rank, local_rank, and world_size inferred from SLURM environment variables.
- MASTER_ADDR and MASTER_PORT can be dynamically set from SLURM for multi-node runs.

3. Multiprocessing / Launching

- **Single node**

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

- **Multi node**

```
# Direct call to train(), as SLURM + srun handles spawning across nodes
train()
```

Key differences

- Single-node uses torch.multiprocessing.spawn.
- Multi-node relies on SLURM srun to launch one process per GPU per node, so no explicit spawn is needed.

4. SLURM Job Script Changes

- **Single node**

```
#SBATCH --nodes=1
#SBATCH --gres=gpu:h100:2
#SBATCH --ntasks-per-node=2

...
python ddp_single_node.py
```

- **Multi node** In ddp_multi_node.slurm,

```
#SBATCH --nodes=2
#SBATCH --gres=gpu:h100:2
#SBATCH --ntasks-per-node=2
...
export MASTER_ADDR=$(scontrol show hostnames $SLURM_NODELIST | head -n 1)
export MASTER_PORT=$((12000 + RANDOM % 10000))
export NCCL_DEBUG=INFO
export NCCL_SOCKET_IFNAME=^lo,docker

srun --export=ALL python ddp_multi_node.py
```

Key differences

- Nodes increased from 1 → 2.
- MASTER_ADDR is set to the first hostname in the node list.
- MASTER_PORT is randomized to avoid conflicts.
- srun handles launching multiple processes across nodes.
- Optional: NCCL debug flags for multi-node troubleshooting.

Submit the job

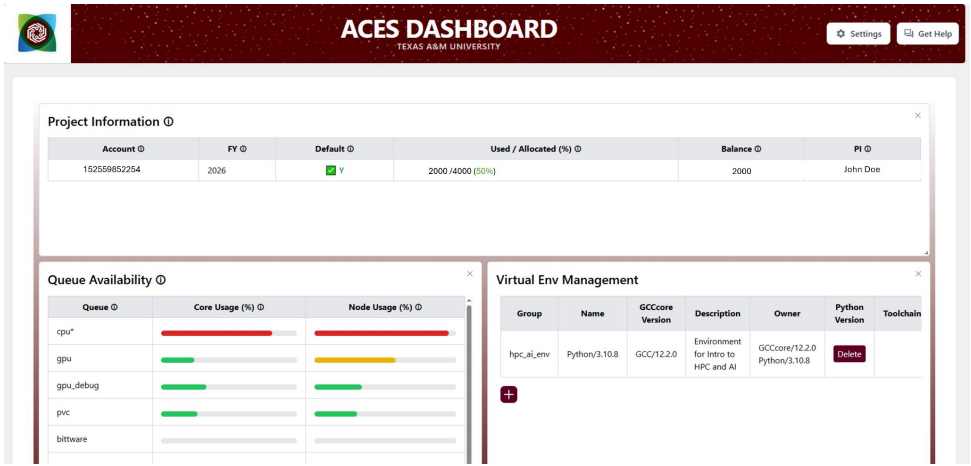
After completing all the steps above, go back to your terminal and run:

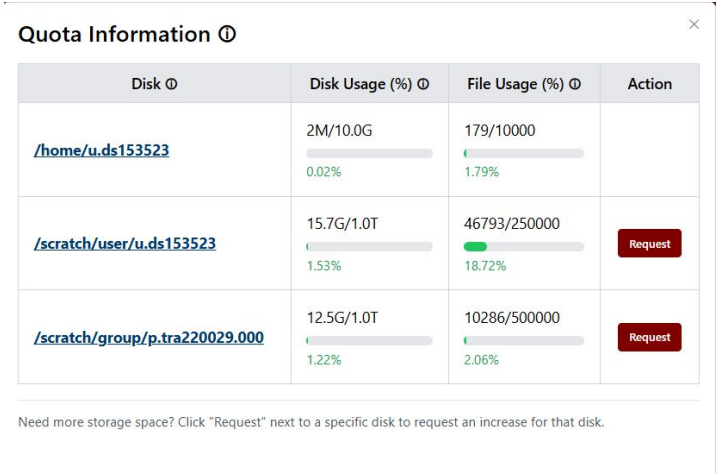
```
cd $SCRATCH/intro_hpc_ai/exercises/3.SingleNode_to_MultiNode
sbatch ddp_multi_node.slurm
```

Need Help?

First check the FAQ: https://hprc.tamu.edu/kb/FAQ/Accounts

- ACES User guide: https://hprc.tamu.edu/kb/User-Guides/ACES
- Email your questions to help@hprc.tamu.edu
- See our training sessions: hprc.tamu.edu/training





Remember the Dashboard!

High Performance Research Computing

DIVISION OF RESEARCH

https://hprc.tamu.edu

HPRC Helpdesk:

help@hprc.tamu.edu
Phone: 979-845-0219

Take our short course survey!

https://u.tamu.edu/hprc_shortcourse_survey

HPRC Survey

https://u.tamu.edu/hprc_shortcourse_survey

Thank you! Any Questions?