Getting Started with SLURM and Python Environments on an HPC Cluster

Md Shahriar Forhad
shahriar.forhad.eee@gmail.com
https://shahriar88.github.io/

January 2, 2026

Contents

1 Introduction

This short tutorial explains how to:

- Write a basic SLURM batch script for running a TensorFlow job.
- Understand key `#SBATCH` directives, including nodes, partitions, GPUs, and time limits.
- Use and extend a shared TensorFlow environment on the cluster.
- Create your own Python virtual environments for CPU and GPU workloads.

All examples are designed to be copied directly into a terminal or SLURM job script on a typical HPC system (e.g., a university cluster).

1.1 Logging In to CRADLE

Once you have received your user account information, you can log in to the CRADLE cluster from a terminal or command-line prompt using your University credentials.

Step 1: Connect to the University Network (VPN)

If you are off campus, you must first connect to the University network using the approved VPN software. Install and start the VPN client, then log in with your University credentials. Once the VPN connection is active, you can reach the CRADLE login node.

Step 2: SSH into the Cluster

To log in, use `ssh` with your assigned username. In the examples below, replace `username` with your actual cluster username.

```
ssh username@login.cradle.university.edu
```

You will be prompted for your password:

```
username@login.cradle.university.edu's password:
```

After you enter your password correctly, you should see a prompt indicating that you are now on the cluster login node, similar to:

```
[username@login001 ~]$
```

At this point you are logged into CRADLE and ready to work with SLURM, Python environments, and the other tools described in this tutorial.

2 A Minimal SLURM Job Script Example

Listing 1 shows a basic SLURM job script that runs a TensorFlow test script in a shared environment.

Listing 1: Basic SLURM job script with TensorFlow environment

```bash
#!/bin/bash

# Job identification
#SBATCH --job-name=myFirstJob

# Standard output and error files (%j = job ID)
#SBATCH --output=myFirstJob.out.%j
#SBATCH --error=myFirstJob.err.%j

# Resource requests
#SBATCH -N 1
#SBATCH -p kimq
#SBATCH --gres=gpu:1
#SBATCH --time=1:00:00

# Display GPU assigned by SLURM
echo "CUDA_VISIBLE_DEVICES: $CUDA_VISIBLE_DEVICES"

# Activate TensorFlow virtual environment
echo "Activating TensorFlow-2.6.2 environment"
source /shared/tensorflow-2.6.2/tf_env/bin/activate

# Run TensorFlow test script
echo "Running testTF.py"
python3 ~/testTFForSlurm/testTF.py

# Deactivate environment
echo "Deactivating TensorFlow-2.6.2 environment"
deactivate
```

To submit this job to SLURM, save the script as `myFirstJob.slurm` (for example) and run:

```
sbatch myFirstJob.slurm
```

3 Understanding Key `#SBATCH` Directives

The `#SBATCH` lines at the top of the script tell SLURM what resources you want. They are not shell commands; they are scheduler directives.

3.1 Number of Nodes: `-N`

```
#SBATCH -N 1
```

This requests **1 compute node**. A node is a physical machine that may contain many CPU cores, memory, and possibly GPUs.

More examples:

```
#SBATCH -N 1 # Single-node TensorFlow training
#SBATCH -N 2 # Distributed job across 2 nodes
#SBATCH -N 1-4 # SLURM may allocate anywhere from 1 to 4 nodes
```

3.2 Partition (Queue): `-p`

```
#SBATCH -p kimq
```

The partition selects the *group of nodes* (queue) your job will run on.
Typical partitions might be:

- `compute` → Standard CPU nodes
- `bigmem` → High-memory nodes
- `gpu` or `kimq` → GPU-enabled nodes

Examples:

```
#SBATCH -p kimq # Run on GPU nodes in the kimq partition
#SBATCH -p compute # Run on CPU-only nodes
```

3.3 GPU Request: `-gres`

```
#SBATCH --gres=gpu:1
```

This requests **1 GPU** using the generic resource (GRES) mechanism.
More examples:

```
#SBATCH --gres=gpu:1 # One GPU
#SBATCH --gres=gpu:2 # Two GPUs
#SBATCH --gres=gpu:a100:1 # One NVIDIA A100 GPU (if supported)
```

Important: Partition and GPU must match. If you request GPUs but choose a CPU-only partition (e.g., `-p compute` with `-gres=gpu:1`), SLURM cannot satisfy the request and the job will stay pending or fail.

3.4 Time Limit: `-time`

```
#SBATCH --time=1:00:00
```

This sets a **wall-clock time limit** of 1 hour for your job. When this time is reached, SLURM terminates the job.
More examples:

```
#SBATCH --time=00:30:00 # 30 minutes
#SBATCH --time=04:00:00 # 4 hours
#SBATCH --time=2-00:00:00 # 2 days
```

3.5 Summary Table of Core Directives

4 Why Use Python Environments on an HPC Cluster?

On HPC clusters, shared software environments (for example, a shared TensorFlow installation) are usually:

- Optimized for the cluster hardware (correct CUDA, cuDNN, drivers).

Directive	Controls	Example
`-N`	Number of nodes	`-N 1`
`-p`	Hardware pool	`-p kimq`
`-gres`	GPU count	`gpu:1`
`-time`	Runtime limit	`1:00:00`

Table 1: Core SLURM directives and examples.

- Read-only for users.
- Not modifiable by individual users.

You *cannot* safely run:

```
pip install pandas
```

inside a shared environment such as

```
 /shared/tensorflow-2.6.2/tf_env/
```

because:

- It is shared by everyone.
- You do not have write permission.
- Changes could break other users' jobs.

Solution: Clone the shared environment into your home directory, then customize the copy.

5 Transferring Files to and from the CRADLE Cluster

When working on the CRADLE HPC cluster, you will often need to move files between your local computer and the cluster. Common examples include:

- Uploading scripts, data files, or configuration files to the cluster.
- Downloading results, logs, or trained models back to your local machine.

This section presents two recommended and commonly used methods:

1. Secure Copy (`scp`)
2. Cloning repositories from GitHub

5.1 Using `scp` (Secure Copy)

The `scp` command allows you to securely transfer files between your local machine and the cluster over SSH. It works on Linux, macOS, and Windows (with PowerShell, Git Bash, or WSL).

5.1.1 Transferring a Local File to the Cluster

To copy a file from your local directory to your home directory on the cluster:

```
scp localFileInMyDirectory.txt \
    yourUserName@login.cradle.university.edu:~/destinationForYourFile/
```

Explanation:

- `localFileInMyDirectory.txt` is the file on your local computer.
- `yourUserName@login.cradle.university.edu` is your cluster login.
- `~/destinationForYourFile/` is the target directory on the cluster.

If the destination directory does not exist, create it first after logging in:

```
mkdir -p ~/destinationForYourFil
```

5.1.2 Transferring a File from the Cluster to Your Local Machine

To copy a file from the cluster back to your local directory:

```
scp yourUserName@login.cradle.university.edu:~/destinationForYourFile/
    localFileInMyDirectory.txt \
    .
```

Here, the dot (`.`) represents your current local directory.

5.2 Working with Directories in the Terminal

In both `scp` examples above, the file name refers to a file located in your *current working directory*. You can usually identify this directory by looking at your terminal prompt.

For example:

```
[username@mycomputer:~/Desktop/test]$
```

This indicates that your current directory is:

```
~/Desktop/test
```

You can explicitly check your current directory using:

On Linux or macOS:

```
pwd
```

This prints something like:

```
/home/username/Desktop/test
```

On Windows (PowerShell):

```
cd
```

Understanding your current directory is essential to avoid copying the wrong files or encountering `file not found` errors.

5.3 Cloning a GitHub Repository

Instead of manually transferring files, it is often easier to store your project code in a GitHub repository and clone it directly on the cluster.

5.3.1 Example: Cloning a Repository

After logging into the cluster, run:

```
git clone https://github.com/emenriquez/testTFForSlurm
```

This creates a new directory called `testTFForSlurm` containing the full repository.
Advantages of using GitHub:

- Version control and change tracking.
- Easy synchronization across multiple machines.
- Clean separation between code and large datasets.

5.4 Recommended Best Practices

- Use `scp` for small files, configuration files, and quick transfers.
- Use GitHub for project code and scripts.
- Avoid transferring very large datasets via `scp`; instead, place them directly on shared cluster storage if available.
- Keep your home directory organized (e.g., `projects/`, `data/`, `results/`).

These practices help ensure efficient workflows and reduce errors when working between your local system and the CRADLE cluster.

6 Cloning a Shared TensorFlow Environment

6.1 Step 1: Install the cloning tool (one time)

```
pip3 install --user virtualenv-clone
```

This:

- Installs `virtualenv-clone` into your user space.
- Does not require admin/root access.
- Only needs to be run once.

6.2 Step 2: Clone the shared environment

```
virtualenv-clone /shared/tensorflow-2.6.2/tf_env ~/myEnvs/tf_env
```

Here:

- `/shared/tensorflow-2.6.2/tf_env/` is the original, optimized, read-only TensorFlow environment.
- `~/myEnvs/tf_env` is your private, writable copy.

What gets copied:

- Python version.
- TensorFlow build (CUDA, cuDNN matched to cluster GPUs).
- All installed dependencies.
- Environment configuration tuned for the cluster.

Advantages:

- Runs as fast as the shared environment.
- You can safely add or upgrade packages.
- No conflicts with other users.

6.3 Step 3: Activate the cloned environment

```
source ~/myEnvs/tf_env/bin/activate
```

After activation:

- `python` and `pip` point to your cloned environment.
- Any `pip install` affects only your copy.

The shell prompt typically changes to something like:

```
(tf_env) username@node \$
```

6.4 Step 4: Add new packages (example: `pandas`)

With the environment active:

```
pip3 install pandas
```

What happens:

- `pandas` is installed only into `~/myEnvs/tf_env`.
- TensorFlow and CUDA support remain intact.
- The customization is isolated and safe.

You can also add:

```
pip install scikit-learn matplotlib seaborn tqdm
```

6.5 Step 5: Use the cloned environment in your SLURM job

Update the activation lines in your job script.

Old (shared environment):

```
echo "Activating TensorFlow-2.6.2 environment"
source /shared/tensorflow-2.6.2/tf_env/bin/activate
```

New (your cloned environment):

```
echo "Activating my custom environment"
source ~/myEnvs/tf_env/bin/activate
```

This ensures:

- Your job sees TensorFlow and any extra packages (e.g., `pandas`).
- You avoid dependency errors at runtime.
- The behavior of your job is reproducible.

6.6 Mental Model Summary

Piece	Purpose
Shared environment	Fast, optimized, read-only
Cloned environment	Same speed, user-modifiable
`virtualenv-clone`	Copies entire environment
`pip install`	Safe only in cloned environment
Job script	Must activate your chosen environment

Table 2: Conceptual roles of shared and cloned environments.

7 Creating Your Own Python Virtual Environment

Sometimes you want a completely independent Python environment. There are two main options; here we focus on standard `venv` and the cloned approach.

7.1 Option A (Recommended for CPU-Only): Create from Scratch

7.1.1 Step 1: Choose a location

Create a directory to hold your environments:

```
mkdir -p ~/myEnvs
```

7.1.2 Step 2: Create the virtual environment

```
python3 -m venv ~/myEnvs/myenv
```

This creates something like:

```
~/myEnvs/myenv/
|- bin/
|- lib/
‘- pyvenv.cfg
```

7.1.3 Step 3: Activate the environment

```
source ~/myEnvs/myenv/bin/activate
```

Your shell prompt changes to:

```
(myenv) username@login $
```

7.1.4 Step 4: Upgrade `pip` (recommended)

```
pip install --upgrade pip
```

7.1.5 Step 5: Install packages

Example:

```
pip install tensorflow pandas numpy matplotlib
```

Note: On clusters, installing TensorFlow this way may not include GPU support unless CUDA and drivers are correctly matched. For GPU workloads, the cloned shared environment (Option B) is often safer.

7.1.6 Step 6: Use this environment in a SLURM job

In your job script:

```
source ~/myEnvs/myenv/bin/activate
python train.py
```

7.2 Option B (Best for GPU): Clone an Optimized Shared Environment

This is essentially the workflow described in Section 6.

Step 1: Install the clone tool:

```
pip install --user virtualenv-clone
```

Step 2: Clone the shared environment:

```
virtualenv-clone /shared/tensorflow-2.6.2/tf_env ~/myEnvs/tf_env
```

Step 3: Activate it:

```
source ~/myEnvs/tf_env/bin/activate
```

Step 4: Add packages:

```
pip install pandas scikit-learn seaborn
```

Step 5: Update your SLURM job to activate:

```
source ~/myEnvs/tf_env/bin/activate
```

This preserves CUDA/cuDNN compatibility and performance while allowing you to customize the environment.

8 Verifying Your Environment and Common Pitfalls

8.1 Verifying Python and GPU Visibility

After activating an environment, verify what `python` and `pip` you are using:

```
which python
which pip
```

For TensorFlow GPU detection:

```
python -c "import tensorflow as tf; print(tf.config.list_physical_devices('GPU'))"
```

You should see at least one GPU listed when running on a GPU node with correct drivers and environment.

8.2 Common Mistakes to Avoid

- Installing packages without activating the environment first.
- Using `pip install -user` inside a virtual environment.
- Mixing Conda environments and `venv` in the same workflow.
- Using the system Python (no virtual environment) for SLURM jobs.
- Requesting GPUs in a CPU-only partition.

8.3 Quick Decision Guide

Situation	Best choice
CPU-only work	Option A (new `venv`)
GPU + TensorFlow	Option B (clone shared env)
Cluster-optimized builds	Option B
Full custom control	Option A

Table 3: Choosing between creating and cloning environments.

9 Using Micromamba and Portable Conda Environments on the HPC Cluster

On many HPC clusters, including CRADLE, traditional Conda/Anaconda installations are discouraged or restricted because they are large, modify shell startup files aggressively, and may conflict with system libraries or CUDA drivers. A lightweight and HPC-friendly alternative is *Micromamba*, a single static binary that implements the Conda ecosystem without requiring a global base environment.

Micromamba allows you to:

- Recreate a Conda environment from a portable YAML file exported on your laptop.
- Install packages into your home directory without administrator rights.

- Activate environments explicitly inside SLURM job scripts.

The overall workflow is:

1. Export a portable Conda environment YAML on your local machine.
2. Copy the YAML file to CRADLE.
3. Install Micromamba in your home directory on CRADLE.
4. Create a Micromamba environment from the YAML file.
5. Activate the environment in your SLURM job scripts.

9.1 Exporting a Portable Conda Environment on Your PC

On your local machine where you already use Conda, start by creating a portable YAML description of your environment. Suppose your local environment is called `IR`.

First, activate it:

```
conda activate IR
```

Then export a *portable* environment file using `fromhistory`:

```
conda env export --from-history > IR_portable.yml
```

This creates a file `IR_portable.yml` containing the channels and high-level dependencies (without build hashes). It is much more likely to recreate successfully on a different system such as CRADLE.

9.2 Copying the YAML File to CRADLE

Use `scp` or an equivalent file transfer method to copy `IR_portable.yml` to your home directory on CRADLE. For example, from your local terminal:

```
scp IR_portable.yml username@login.cradle.university.edu:~
```

After logging into CRADLE, you should see the file in your home directory:

```
ls ~/IR_portable.yml
```

9.3 Installing Micromamba in Your Home Directory

Micromamba is distributed as a single binary, which makes it ideal for installation in a user home directory (no root access required).

Create a directory for the binary:

```
mkdir -p ~/bin
cd ~/bin
```

Download and unpack the latest Micromamba binary (Linux 64-bit):

```
curl -L https://micro.mamba.pm/api/micromamba/linux-64/latest \
  | tar -xvj bin/micromamba
```

Add `$HOME/bin` to your `PATH` in ~/`.bashrc`:

```
echo 'export PATH=$HOME/bin:$PATH' >> ~/.bashrc
source ~/.bashrc
```

Verify that `micromamba` is available:

```
micromamba --version
```

9.4 Creating a Micromamba Environment from the YAML File

Choose a directory where Micromamba will store its environments, for example:

```
mkdir -p ~/micromamba/envs
```

Initialize Micromamba for Bash (one time):

```
micromamba shell init -s bash -p ~/micromamba
source ~/.bashrc
```

Now create a new environment (for example, `IR_NEW`) from your portable YAML file:

```
micromamba create \
  -f ~/IR_portable.yml \
  -n IR_NEW \
  -p ~/micromamba/envs/IR_NEW
```

Here:

- `~/IR_portable.yml` is the YAML exported on your PC.
- `IR_NEW` is the environment name.
- `~/micromamba/envs/IR_NEW` is the full path Micromamba uses.

9.5 Activating the Environment on CRADLE

Once created, you can activate the environment on the CRADLE login node:

```
micromamba activate ~/micromamba/envs/IR_NEW
```

After activation:

- `python` and `pip` refer to the Micromamba environment.
- Any installs via `pip` or `micromamba install` affect only that environment.

You can verify:

```
which python
python --version
pip list
```

9.6 Using a Micromamba Environment in a SLURM Job

In a SLURM job script, you should explicitly set the Micromamba root and activate the environment before running Python code. For example:

```bash
#!/bin/bash
#SBATCH --job-name=ir_gpu_job
#SBATCH --output=ir_gpu_job.out.%j
#SBATCH --error=ir_gpu_job.err.%j
#SBATCH -N 1
#SBATCH -p kimq # GPU partition
#SBATCH --gres=gpu:1 # 1 GPU
#SBATCH --time=02:00:00 # 2 hours

# Ensure Micromamba root is defined
export MAMBA_ROOT_PREFIX=$HOME/micromamba

# Load shell initialization (if micromamba shell init was used)
source $HOME/.bashrc

echo "Host: $(hostname)"
echo "CUDA_VISIBLE_DEVICES: $CUDA_VISIBLE_DEVICES"

echo "Activating Micromamba environment IR_NEW"
micromamba activate $HOME/micromamba/envs/IR_NEW

echo "Checking TensorFlow GPU visibility"
python -c "import tensorflow as tf; print(tf.config.list_physical_devices('GPU'))"

echo "Running training script"
python ~/projects/train_ir_model.py
```

This pattern ensures that:

- The correct Micromamba environment is active inside the job.
- GPU visibility can be checked via TensorFlow (or PyTorch).
- The job is reproducible and does not depend on the login-node Conda setup.

9.7 Verifying GPU Support Inside the Environment

To confirm that your Micromamba environment has working GPU support for TensorFlow (or another framework), you can run:

```bash
python -c "import tensorflow as tf; print(tf.config.list_physical_devices('GPU'))"
```

On a GPU node with correct drivers and CUDA libraries, this should list one or more GPU devices. You can also check the low-level GPU visibility with:

```bash
nvidia-smi
```

9.8 Best Practices and Common Pitfalls for Micromamba on HPC

Best practices:

- Export environments from Conda using `-from-history` or `-no-builds` to avoid hardware-specific build strings.
- Use Micromamba to recreate the environment on the cluster, not full Anaconda.
- Keep all Micromamba files under `~/micromamba` or a dedicated directory in your home space.
- Activate environments explicitly inside SLURM job scripts.

Common pitfalls:

- Installing full Anaconda on an HPC login node (disk usage and conflicts).
- Relying on Conda's `base` environment in batch jobs.
- Mixing Conda and Micromamba environments in the same workflow.
- Installing GPU frameworks with incompatible `cudatoolkit` versions; always check which CUDA versions the cluster supports.

Using Micromamba with a portable Conda YAML exported from your local machine gives you a reproducible, HPC-safe way to manage Python environments on CRADLE while respecting cluster policies and maximizing compatibility with the installed NVIDIA drivers and CUDA libraries.

10 Worked Examples: Environments and SLURM Job Scripts

This section provides concrete, copy-pasteable examples covering:

1. Creating a new environment.
2. Cloning a shared TensorFlow environment.
3. Selecting (activating) an environment.
4. Installing extra libraries.
5. Checking and verifying resources.
6. Writing SLURM scripts for different resource requirements.

10.1 Creating a New CPU-Only Environment

```
# 1. Create a directory to store your environments
mkdir -p ~/myEnvs

# 2. Create a new CPU-only Python virtual environment
python3 -m venv ~/myEnvs/cpu_env

# 3. Activate the environment
source ~/myEnvs/cpu_env/bin/activate

# 4. Upgrade pip
pip install --upgrade pip

# 5. Install some common libraries
pip install numpy scipy pandas matplotlib scikit-learn

# 6. When finished, deactivate
deactivate
```

10.2 Cloning a Shared GPU-Optimized TensorFlow Environment

```
# 1. Install the clone tool (only once per user)
pip3 install --user virtualenv-clone

# 2. Create a directory to store your environments (if not already created)
mkdir -p ~/myEnvs

# 3. Clone the shared TensorFlow environment into your home directory
virtualenv-clone /shared/tensorflow-2.6.2/tf_env ~/myEnvs/tf_env

# 4. Activate the cloned environment
source ~/myEnvs/tf_env/bin/activate

# 5. Install extra libraries needed for your project
pip install pandas scikit-learn matplotlib seaborn tqdm

# 6. Deactivate when done
deactivate
```

10.3 Selecting an Environment in a Job Script

Activate the CPU-only environment:

```
# Inside your SLURM job script
echo "Activating CPU-only environment"
source ~/myEnvs/cpu_env/bin/activate
```

Activate the cloned GPU TensorFlow environment:

```
# Inside your SLURM job script
echo "Activating GPU TensorFlow environment"
source ~/myEnvs/tf_env/bin/activate
```

10.4 Installing Libraries After Environment Activation

Inside the CPU-only environment:

```
source ~/myEnvs/cpu_env/bin/activate
pip install jupyter notebook ipykernel
deactivate
```

Inside the cloned GPU environment:

```
source ~/myEnvs/tf_env/bin/activate
pip install opencv-python pillow tensorboard
deactivate
```

10.5 Checking and Verifying Resources

This subsection shows how to verify what resources are available on the cluster and what resources SLURM has actually allocated to your jobs.

10.5.1 Check Available Partitions and Nodes

To see which partitions and nodes exist and their current status:

```
# Show all partitions and their status
sinfo
```

For more detailed information about each node:

```
# Detailed node information
scontrol show nodes
```

10.5.2 Check Your Jobs

To see all of your jobs in the queue:

```
# Show jobs for your user
squeue -u $USER
```

To inspect a specific job by its job ID:

```
# Show a specific job
squeue -j <JOBID>
```

10.5.3 Inspect Resources Allocated by SLURM

To see precisely what resources were allocated to a job:

```
# Show full job allocation details
scontrol show job <JOBID>
```

Look for fields such as `NumNodes`, `NumCPUs`, `Gres`, and `NodeList`.

10.5.4 Check Resources Inside a Job Script

Inside a running job, you can query SLURM environment variables to see what was allocated:

```
echo "Job ID: $SLURM_JOB_ID"
echo "Nodes allocated: $SLURM_NNODES"
echo "Node list: $SLURM_NODELIST"
echo "CPUs per node: $SLURM_CPUS_ON_NODE"
echo "Tasks per node: $SLURM_TASKS_PER_NODE"
echo "Job partition: $SLURM_JOB_PARTITION"
```

10.5.5 Check GPU Allocation

For GPU jobs, it is important to verify which GPUs are visible:

```
# GPUs assigned by SLURM (logical IDs)
echo "CUDA_VISIBLE_DEVICES: $CUDA_VISIBLE_DEVICES"

# List visible GPUs and utilization
nvidia-smi
```

10.5.6 Check CPU and Memory Usage

You can inspect CPU and memory characteristics on the node:

```
# CPU info
lscpu

# Memory info
free -h

# Per-node memory summary (first lines of /proc/meminfo)
cat /proc/meminfo | head
```

10.5.7 Check Resources Used by Completed Jobs

After a job finishes, accounting information can be retrieved with `sacct`:

```
# Accounting info for a finished job
sacct -j <JOBID> --format=JobID,JobName,Partition,AllocCPUS,Elapsed,MaxRSS,State
```

This shows how many CPUs were allocated, how long the job ran, peak memory usage (`MaxRSS`), and the final state (e.g., COMPLETED, FAILED, TIMEOUT).

10.5.8 Minimal Resource Debug Job

The following SLURM script does nothing except report which resources it received. It is useful when testing new partitions or debugging allocation issues:

```bash
#!/bin/bash
#SBATCH --job-name=resource_debug
#SBATCH -N 1
#SBATCH -p kimq
#SBATCH --gres=gpu:1
#SBATCH --time=00:10:00

echo "Job ID: $SLURM_JOB_ID"
echo "Node list: $SLURM_NODELIST"
echo "CPUs/node: $SLURM_CPUS_ON_NODE"
echo "GPUs visible: $CUDA_VISIBLE_DEVICES"

echo "==== nvidia-smi ===="
nvidia-smi

echo "==== lscpu ===="
lscpu

echo "==== memory ===="
free -h
```

Aspect	**CPU**	**Single GPU**	**DataParallel (DP)**	**DistributedDataParallel (DDP)**
Hardware	CPU cores only	1 GPU	Multiple GPUs (1 node)	Multiple GPUs (1+ nodes)
Processes	1	1	1	1 per GPU
GPU usage	None	Full single GPU	Automatic batch split	Explicit per-rank GPU
Multi-node support	No	No	No	Yes
Performance	Slowest	Fast	Medium	Fastest
Scalability	Poor	Limited	Limited	Excellent
Communication	None	None	CPU-based gather/scatter	NCCL all-reduce
Gradient sync	N/A	N/A	Implicit (inefficient)	Explicit (efficient)
Memory efficiency	High	Medium	Low (replication overhead)	High
Fault isolation	N/A	N/A	Poor	Good
Code complexity	Very low	Low	Low	Medium
Launch method	`python`	`python`	`python`	`torchrun` / `srun`
Best use case	Debugging, tests	Small models	Prototyping	Large-scale training
Recommended for HPC	No	Limited	No	Yes

Table 4: Comparison of CPU, single-GPU, DataParallel (DP), and DistributedDataParallel (DDP) execution modes in PyTorch.

11 PyTorch DataParallel (DP) on SLURM

This section describes PyTorch **DataParallel (DP)**, a simpler multi-GPU approach compared to Distributed Data Parallel (DDP). DP uses a *single Python process* and automatically splits each batch across multiple GPUs on the same node.

Important: DP is suitable for quick experiments on a *single node* only. For serious training, large models, or multi-node jobs, prefer DDP (Section 12).

11.1 When to Use DataParallel

- Single node with multiple GPUs.

- Prototyping or debugging multi-GPU behavior.
- Minimal code changes from single-GPU training.

11.2 When *Not* to Use DataParallel

- Multi-node training (DP does not scale across nodes).
- Large models or long training runs (DDP is faster and more stable).
- When precise performance benchmarking matters.

11.3 Minimal DataParallel Training Skeleton

Save as `train_dp.py`. This example assumes a single node with multiple GPUs.

Listing 2: Minimal PyTorch DataParallel training skeleton (`train_dp.py`)

```python
import torch
import torch.nn as nn
from torch.utils.data import DataLoader

def build_model():
    # Replace with your FCN / VAE model
    return nn.Identity()

def build_dataset(split="train"):
    # Replace with your dataset
    raise NotImplementedError

def main():
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

    model = build_model()

    if torch.cuda.device_count() > 1:
        print(f"Using DataParallel on {torch.cuda.device_count()} GPUs")
        model = nn.DataParallel(model)

    model = model.to(device)

    train_ds = build_dataset("train")
    train_loader = DataLoader(
        train_ds,
        batch_size=16,
        shuffle=True,
        num_workers=4,
        pin_memory=True,
        drop_last=True,
    )

    optim = torch.optim.Adam(model.parameters(), lr=1e-3)

    model.train()
    for epoch in range(5):
        for batch in train_loader:
            X, Y = batch
            X = X.to(device, non_blocking=True)
            Y = Y.to(device, non_blocking=True)

            optim.zero_grad(set_to_none=True)
            Yhat = model(X)
            loss = torch.mean((Yhat - Y) ** 2)
            loss.backward()
            optim.step()

        print(f"Epoch {epoch} done. loss={loss.item():.6f}")

if __name__ == "__main__":
    main()
```

11.4 SLURM Script for DataParallel (single node)

DP requires **one Python process only**. Do *not* use `torchrun` or multiple tasks.

Listing 3: SLURM script for DataParallel (`dp_1node.slurm`)

```
#!/bin/bash
#SBATCH --job-name=dp_1node
#SBATCH --output=dp_1node.out.%j
#SBATCH --error=dp_1node.err.%j
#SBATCH -N 1
#SBATCH -p kimq
#SBATCH --gres=gpu:4
#SBATCH --cpus-per-task=8
#SBATCH --time=04:00:00

echo "Host: \$(hostname)"
echo "CUDA_VISIBLE_DEVICES: \$CUDA_VISIBLE_DEVICES"

source ~/myEnvs/tf_env/bin/activate

export OMP_NUM_THREADS=\$SLURM_CPUS_PER_TASK

# Single process only
python train_dp.py

deactivate
```

11.5 DP vs DDP: Practical Comparison

Feature	DataParallel (DP)	DistributedDataParallel (DDP)
Processes	1 total	1 per GPU
Multi-node support	No	Yes
Performance	Slower	Faster
Scalability	Limited	Excellent
Failure isolation	Poor	Good
Recommended for	Prototyping	Production / research

Table 5: Comparison between DataParallel and DistributedDataParallel.

11.6 Common DP Pitfalls

- Do not combine DP with `torchrun` or `srun -n > 1`.
- Expect slower scaling beyond 2–4 GPUs.
- Avoid DP for multi-node jobs.
- Always move the model to GPU *after* wrapping with `DataParallel`.

11.7 Rule of Thumb

- **1 node, quick test** → DataParallel
- **Anything serious or multi-node** → DDP

12 PyTorch Distributed Data Parallel (DDP) on SLURM

This section shows a minimal, practical recipe for PyTorch DDP on an HPC cluster. DDP runs one Python process per GPU and synchronizes gradients efficiently.

12.1 What You Need (Checklist)

- Request **multiple GPUs** in SLURM (e.g., `-gres=gpu:4`).
- Launch with `torchrun` (recommended) or `srun` + environment variables.
- In Python: call `init_process_group`, set the per-rank GPU, wrap model with `DistributedDataParallel`.
- Use `DistributedSampler` for training (and validation if you want correct global metrics quickly).

12.2 Minimal DDP Training Skeleton (single file)

Save as `train_ddp.py`. Replace `build_model()`, `build_dataset()`, and the loss/optimizer with your own code.

Listing 4: Minimal PyTorch DDP training skeleton (`train_ddp.py`)

```python
import os
import torch
import torch.nn as nn
import torch.distributed as dist
from torch.nn.parallel import DistributedDataParallel as DDP
from torch.utils.data import DataLoader
from torch.utils.data.distributed import DistributedSampler

# -------------------------
# 1) DDP setup / teardown
# -------------------------
def ddp_setup():
    """
    Works with: torchrun (sets RANK, LOCAL_RANK, WORLD_SIZE, MASTER_ADDR, MASTER_PORT
        ).
    """
    if dist.is_available() and not dist.is_initialized():
        dist.init_process_group(backend="nccl") # GPUs -> nccl

    rank = int(os.environ.get("RANK", "0"))
    local_rank = int(os.environ.get("LOCAL_RANK", "0"))
    world_size = int(os.environ.get("WORLD_SIZE", "1"))

    if torch.cuda.is_available():
        torch.cuda.set_device(local_rank)
        device = torch.device(f"cuda:{local_rank}")
    else:
        device = torch.device("cpu")

    return device, rank, local_rank, world_size

def ddp_cleanup():
    if dist.is_available() and dist.is_initialized():
        dist.destroy_process_group()

# -------------------------
# 2) Your builders (replace)
# -------------------------
def build_model():
    # Replace with your FCN/VAE model builder
    return nn.Identity()

def build_dataset(split="train"):
    # Replace with your dataset (e.g., CleanImageFolder / PairedImageFolder)
    # Must return a torch.utils.data.Dataset
    raise NotImplementedError

# -------------------------
# 3) Main training loop
# -------------------------
def main():
    device, rank, local_rank, world = ddp_setup()
    is_ddp = (world > 1)

    model = build_model().to(device)
    if is_ddp:
        model = DDP(model, device_ids=[local_rank], output_device=local_rank)

```

12.3 SLURM Script for DDP (single node, 4 GPUs)

This launches **one process per GPU** on the same node.

Listing 5: SLURM script for DDP on 1 node (`ddp_1node.slurm`)

```
#!/bin/bash
#SBATCH --job-name=ddp_1node
#SBATCH --output=ddp_1node.out.%j
#SBATCH --error=ddp_1node.err.%j
#SBATCH -N 1
#SBATCH -p kimq
#SBATCH --gres=gpu:4
#SBATCH --cpus-per-task=4
#SBATCH --time=04:00:00

echo "Host: \$(hostname)"
echo "Node list: \$SLURM_NODELIST"
echo "CUDA_VISIBLE_DEVICES: \$CUDA_VISIBLE_DEVICES"

source ~/myEnvs/tf_env/bin/activate

# One process per GPU:
export OMP_NUM_THREADS=\$SLURM_CPUS_PER_TASK

# torchrun is recommended.
# --standalone works for single-node. For multi-node, use rendezvous settings shown
    below.
srun torchrun \
  --standalone \
  --nproc_per_node=4 \
  train_ddp.py

deactivate
```

12.4 SLURM Script for DDP (2 nodes, 4 GPUs per node)

For multi-node DDP, you need a rendezvous endpoint (master address/port). A common pattern is to use the first node in `$SLURM_NODELIST` as the master.

Listing 6: SLURM script for DDP on 2 nodes (`ddp_2node.slurm`)

```
#!/bin/bash
#SBATCH --job-name=ddp_2node
#SBATCH --output=ddp_2node.out.%j
#SBATCH --error=ddp_2node.err.%j
#SBATCH -N 2
#SBATCH -p kimq
#SBATCH --gres=gpu:4
#SBATCH --cpus-per-task=4
#SBATCH --time=06:00:00

source ~/myEnvs/tf_env/bin/activate

# Pick master node and port
MASTER_ADDR=\$(scontrol show hostnames "\$SLURM_NODELIST" | head -n 1)
MASTER_PORT=29500

echo "MASTER_ADDR=\$MASTER_ADDR"
echo "MASTER_PORT=\$MASTER_PORT"

export OMP_NUM_THREADS=\$SLURM_CPUS_PER_TASK

# Total processes = nodes * gpus_per_node = 2 * 4 = 8
srun torchrun \
  --nnodes=2 \
  --nproc_per_node=4 \
  --rdzv_backend=c10d \
  --rdzv_endpoint=\$MASTER_ADDR:\$MASTER_PORT \
  train_ddp.py

deactivate
```

12.5 Common DDP Pitfalls (Fast Debug List)

- **Do not use `DataParallel` with DDP.** Use one or the other.
- **Call `DistributedSampler(...)` for training** and **call** `set_epoch(epoch)` each epoch.
- **Set the per-process GPU** via `LOCAL_RANK` and `torch.cuda.set_device(local_rank)`.
- **Print/checkpoint only on rank 0** to avoid duplicates and file corruption.
- If validation metrics must be global, either:
  - use `DistributedSampler` for validation and `all_reduce` metric sums, or
  - run validation only on rank 0 (simpler but slower).

12.6 SLURM Job Templates for Different Resource Requirements

This subsection shows full job scripts for common use cases.

12.7 Example Commands for File Transfer

This subsection provides practical, copy-pasteable examples for transferring files and directories between your local machine and the CRADLE cluster. These commands are typically run from a local terminal, not inside a SLURM job.

12.7.1 Copying a Single File from Local Machine to CRADLE

```
scp my_script.py \
    yourUserName@login.cradle.university.edu:~/projects/
```

This copies `my_script.py` from your current local directory into `~/projects/` on the cluster.

12.7.2 Copying a Single File from CRADLE to Local Machine

```
scp yourUserName@login.cradle.university.edu:~/projects/results.txt \
    .
```

The dot (`.`) represents your current local directory.

12.7.3 Copying an Entire Directory to CRADLE

To recursively copy a directory (for example, a project folder):

```
scp -r myProjectFolder/ \
    yourUserName@login.cradle.university.edu:~/projects/
```

This creates `~/projects/myProjectFolder/` on the cluster.

12.7.4 Copying an Entire Directory from CRADLE to Local Machine

```
scp -r yourUserName@login.cradle.university.edu:~/projects/myProjectFolder/ \
    ~/Desktop/
```

This downloads the entire project directory to your local desktop.

12.7.5 Transferring Job Output After Completion

After a SLURM job finishes, you often want to retrieve output files such as logs or trained models.

```
scp yourUserName@login.cradle.university.edu:~/slurm_outputs/myJob.out.123456 \
    .
```

Here, `123456` is the SLURM job ID.

12.7.6 Copying Files Using Wildcards

You can use wildcards to transfer multiple files at once:

```
scp yourUserName@login.cradle.university.edu:~/slurm_outputs/*.out.* \
    .
```

This copies all SLURM output files in the directory.

12.7.7 Recommended Workflow with SLURM Jobs

A typical workflow looks like this:

1. Copy scripts to CRADLE using `scp` or `git clone`.
2. Submit the job using `sbatch`.
3. Monitor the job with `squeue`.
4. Copy results back to your local machine using `scp`.

Important Note: File transfers should generally be performed from the *login node*, not from inside running compute jobs, unless explicitly required.

12.7.8 Single Node, CPU-Only Job

```bash
#!/bin/bash
#SBATCH --job-name=cpu_only_job
#SBATCH --output=cpu_only_job.out.%j
#SBATCH --error=cpu_only_job.err.%j
#SBATCH -N 1
#SBATCH -p compute # CPU-only partition
#SBATCH --time=01:00:00 # 1 hour

echo "Host: $(hostname)"
echo "Using CPU-only environment"
source ~/myEnvs/cpu_env/bin/activate

echo "Running CPU-only Python script"
python ~/projects/my_cpu_script.py

echo "Deactivating environment"
deactivate
```

12.7.9 Single Node, Single GPU Job

```bash
#!/bin/bash
#SBATCH --job-name=single_gpu_job
#SBATCH --output=single_gpu_job.out.%j
#SBATCH --error=single_gpu_job.err.%j
#SBATCH -N 1
#SBATCH -p kimq # GPU partition
#SBATCH --gres=gpu:1 # 1 GPU
#SBATCH --time=02:00:00 # 2 hours

echo "Host: $(hostname)"
echo "CUDA_VISIBLE_DEVICES: $CUDA_VISIBLE_DEVICES"

echo "Activating GPU TensorFlow environment"
source ~/myEnvs/tf_env/bin/activate

echo "Running single-GPU training script"
python ~/projects/train_single_gpu.py

echo "Deactivating environment"
deactivate
```

12.7.10 Single Node, Multiple GPUs (e.g., 4 GPUs)

```bash
#!/bin/bash
#SBATCH --job-name=multi_gpu_job
#SBATCH --output=multi_gpu_job.out.%j
#SBATCH --error=multi_gpu_job.err.%j
#SBATCH -N 1
#SBATCH -p kimq # GPU partition
#SBATCH --gres=gpu:4 # 4 GPUs on the same node
#SBATCH --time=04:00:00 # 4 hours

echo "Host: $(hostname)"
echo "CUDA_VISIBLE_DEVICES: $CUDA_VISIBLE_DEVICES"

echo "Activating GPU TensorFlow environment"
source ~/myEnvs/tf_env/bin/activate

echo "Running multi-GPU training script (e.g., data parallel)"
python ~/projects/train_multi_gpu.py --num-gpus 4

echo "Deactivating environment"
deactivate
```

12.7.11 Multiple Nodes, CPU-Only (e.g., MPI or Distributed CPU Job)

```bash
#!/bin/bash
#SBATCH --job-name=cpu_multi_node_job
#SBATCH --output=cpu_multi_node_job.out.%j
#SBATCH --error=cpu_multi_node_job.err.%j
#SBATCH -N 2 # 2 nodes
#SBATCH -p compute # CPU-only partition
#SBATCH --time=03:00:00 # 3 hours

echo "SLURM_NODELIST: $SLURM_NODELIST"
echo "SLURM_NNODES: $SLURM_NNODES"

echo "Activating CPU-only environment"
source ~/myEnvs/cpu_env/bin/activate

# Example: launch an MPI or distributed job
# (replace with your actual launcher, e.g., srun mpirun, torchrun, etc.)
echo "Running distributed CPU job"
srun python ~/projects/distributed_cpu_job.py

echo "Deactivating environment"
deactivate
```

12.7.12 Multiple Nodes, Multiple GPUs (Distributed GPU Training)

```bash
#!/bin/bash
#SBATCH --job-name=multi_node_multi_gpu
#SBATCH --output=multi_node_multi_gpu.out.%j
#SBATCH --error=multi_node_multi_gpu.err.%j
#SBATCH -N 2 # 2 nodes
#SBATCH -p kimq # GPU partition
#SBATCH --gres=gpu:4 # 4 GPUs per node (total 8 GPUs)
#SBATCH --time=06:00:00 # 6 hours

echo "SLURM_NODELIST: $SLURM_NODELIST"
echo "SLURM_NNODES: $SLURM_NNODES"
echo "CUDA_VISIBLE_DEVICES: $CUDA_VISIBLE_DEVICES"

echo "Activating GPU TensorFlow environment"
source ~/myEnvs/tf_env/bin/activate

# Example using srun to launch a distributed training job
# Replace with your actual distributed launcher (e.g., torchrun, horovodrun)
echo "Running multi-node, multi-GPU training"
srun python ~/projects/train_distributed_gpu.py \
    --nodes $SLURM_NNODES \
    --gpus-per-node 4

echo "Deactivating environment"
deactivate
```

12.8 Quick Mapping: Which Example to Use?

- **Create new CPU environment**: Listing in Section 10.1.
- **Clone shared GPU TF environment**: Listing in Section 10.2.
- **CPU-only job**: Section 12.7.8.
- **Single-GPU job**: Section 12.7.9.
- **Single node, multiple GPUs**: Section 12.7.10.
- **Multi-node CPU job**: Section 12.7.11.
- **Multi-node, multi-GPU job**: Section 12.7.12.

13 Summary

In this tutorial, you have seen:

- How to write and submit a basic SLURM job script that runs a TensorFlow program.
- How to interpret key `#SBATCH` directives: number of nodes, partition, GPU requests, and time limits.
- Why shared environments on HPC clusters are read-only and how to safely extend them by cloning to your home directory.
- How to create your own Python virtual environments from scratch and how to connect them to SLURM job scripts.

This workflow lets you run fast, GPU-enabled jobs using cluster-optimized software while keeping your custom Python packages isolated and safe.

Index