Getting Started with

HPC Clusters, Slurm & tmux

A Beginner-Friendly Guide for Running Python on Any HPC Cluster

Amirhossein M.[1]

What Is This Guide About?

This guide will teach you — step by step — how to connect to and use an HPC (High-Performance Computing) cluster — a powerful shared computer you can access over the internet. You will learn how to:

- Connect to an HPC cluster from your laptop
- Navigate the terminal (the text-based command window)
- Run Python scripts directly or using tmux (so they keep running even if you disconnect)
- Submit jobs to Slurm — the system that schedules GPU and CPU work on the cluster

> **Why use a supercomputer?**
> Your laptop has limited CPU and GPU power. An HPC cluster gives you access to powerful GPUs for training machine learning models, running simulations, and processing large datasets — jobs that would take days on a laptop can finish in minutes.

Part 1: Connecting to Your HPC Cluster

Step 1 — Open a Terminal

A terminal (also called a command prompt or shell) is a window where you type commands instead of clicking buttons. Here is how to open one:

- Windows: Press the Windows key, search for "PowerShell" or "Windows Terminal", and open it.
- Mac: Press Cmd + Space, type "Terminal", and press Enter.
- Linux: Press Ctrl + Alt + T.

[1] a.mohammadi@uu.nl

Step 2 — Connect via SSH

SSH (Secure Shell) lets you log into the cluster remotely. Your institution will provide you with a hostname, username, and password. Type the following command and press Enter:

```
ssh username@hpc.university.edu
```

Replace `username` and `hpc.university.edu` with the credentials provided by your institution or instructor. You will be asked for your password — type it and press Enter (note: the cursor will not move while you type — this is normal).

> **Tip: First-time connection**
> The first time you connect, you may see a message asking if you trust the server. Type "yes" and press Enter.
> Common HPC hostnames look like: login.hpc.university.edu, cluster.dept.ac.uk, or hpc.institution.nl — check your welcome email.

Part 2: Navigating the Terminal

Once connected, you are in a text environment. Instead of clicking folders, you type commands. Here are the essential ones:

Command	What it does
`ls`	List files and folders in the current location
`ls -lh`	List files with sizes and details (human-readable)
`cd foldername`	Enter a folder (directory)
`cd ..`	Go up one level (back to the parent folder)
`cd ~`	Go directly to your home folder
`pwd`	Show where you currently are
`mkdir name`	Create a new folder called 'name'
`rm filename`	Delete a file (be careful — no undo!)
`rm -r foldername`	Delete an entire folder and its contents
`cat file.txt`	Display the contents of a text file
`head -n 20 file.txt`	Show only the first 20 lines of a file
`tail -n 20 file.txt`	Show only the last 20 lines of a file

Example walkthrough

```
ls              <- see what is in your home folder
mkdir my_project    <- create a new project folder
cd my_project       <- enter that folder
ls              <- it is empty for now
```

Part 2b: Copying and Moving Files — cp and mv

Two of the most useful commands for managing files on an HPC cluster are cp (copy) and mv (move/rename). Think of them like drag-and-drop — but in the terminal.

Copying Files and Folders — cp

cp makes a duplicate of a file. The original stays in place.

```
cp source.py destination.py          # copy a file, giving it a new name
cp myscript.py backup/myscript.py    # copy into a folder
cp -r my_project/ my_project_backup/ # copy an entire folder (-r means
recursive)
```

Command	What it does
`cp file.py copy.py`	Make a copy of file.py called copy.py
`cp file.py folder/`	Copy file.py into folder/
`cp -r folder/ backup/`	Copy an entire folder into backup/

Tip

Use cp -r to duplicate a whole project folder before making big changes — a handy safety net!

Moving and Renaming Files — mv

mv moves a file to a new location OR renames it. Unlike cp, the original is removed.

```
mv oldname.py newname.py             # rename a file
mv myscript.py scripts/              # move into a folder
mv results/ outputs/results/         # move a whole folder
```

Command	What it does
`mv old.py new.py`	Rename old.py to new.py

Command	What it does
`mv file.py folder/`	Move file.py into folder/
`mv folder/ newlocation/`	Move an entire folder

Warning — mv overwrites without asking!
If a file with the same name already exists at the destination, mv will silently overwrite it. Always double-check before moving.

Part 2c: Editing Files in the Terminal — vim

On an HPC cluster you cannot open a graphical text editor. Instead, you edit files directly in the terminal using vim (or the simpler nano). This section focuses on vim — it feels strange at first, but a few commands get you a long way.

vim has two modes
NORMAL mode — for navigating and running commands. This is where vim starts.
INSERT mode — for actually typing text. Press i to enter it.
You switch between them with Esc (back to Normal) and i (into Insert).

Opening and Closing Files

```
vim myscript.py       # open (or create) a file
```

Command	What it does
`:w`	Save (write) the file
`:q`	Quit vim (only works if no unsaved changes)
`:wq`	Save and quit
`:q!`	Quit WITHOUT saving (force quit)
`:w newname.py`	Save as a different filename

Moving Around (Normal Mode)

Command	What it does
`Arrow keys`	Move the cursor (works in most setups)
`gg`	Jump to the very top of the file

Command	What it does
`G`	Jump to the very bottom of the file
`0`	Jump to the start of the current line
`$`	Jump to the end of the current line
`/searchterm`	Search for a word (press n for next match)

Editing Text

Command	What it does
`i`	Enter Insert mode before the cursor
`o`	Insert a new line below and enter Insert mode
`Esc`	Return to Normal mode (stop typing)
`dd`	Delete the entire current line
`u`	Undo the last change
`Ctrl + r`	Redo (undo the undo)
`yy`	Copy (yank) the current line
`p`	Paste the copied line below the cursor

Minimal vim survival guide

1. Open a file: vim myscript.py
2. Start editing: press i
3. Make changes
4. Stop editing: press Esc
5. Save and quit: type :wq then press Enter

If you get stuck, press Esc a few times then type :q! to exit without saving.

nano — The Beginner-Friendly Alternative

If vim feels overwhelming, nano is much simpler. Commands are shown at the bottom of the screen.

```
nano myscript.py
```

Command	What it does
`Ctrl + O, Enter`	Save the file
`Ctrl + X`	Exit nano

Command	What it does
`Ctrl + K`	Cut (delete) the current line
`Ctrl + W`	Search for text

Part 3: Uploading Your Python Files

Before running a Python script on the cluster, you need to transfer it from your laptop. Open a NEW terminal on your local machine (not the cluster one) and use SCP:

```
scp myscript.py username@hpc.university.edu:~/my_project/
```

This copies myscript.py from your laptop into the my_project folder on the cluster. To copy an entire folder, add the -r flag:

```
scp -r my_project/ username@hpc.university.edu:~/
```

Part 4: Loading Python and Running a Quick Test

Most HPC clusters use a module system to manage software — this allows many versions of tools to coexist. Before you can use Python, you need to load it:

```
module load Python
python myscript.py
```

Important — Login Nodes vs Compute Nodes

When you SSH into the cluster, you land on a login node. This is a shared entry point, NOT meant for heavy computation. Only run tiny test commands here. For real workloads, always use Slurm (see Part 6).

Part 5: Using tmux for Persistent Sessions

Why Use tmux?

If you close your laptop or lose your Wi-Fi connection, any running program in your SSH session will be killed. tmux solves this by keeping your session alive on the server even when you disconnect.

Think of tmux like this...
Imagine leaving a TV on at home while you go out. When you come back, it is still running. tmux does the same for your terminal sessions on the cluster.

tmux — Step by Step

Start a new session (give it a memorable name):

```
tmux new -s mysession
```

You are now inside a tmux session. Run your script as normal:

```
python myscript.py
```

Detach from the session (your script keeps running):

```
Ctrl + B, then press D
```

Come back later and reconnect to your session:

```
tmux attach -t mysession
```

See all active sessions:

```
tmux ls
```

Close a session when you are done:

```
tmux kill-session -t mysession
```

When to use tmux vs Slurm
Use tmux for: quick tests, debugging, interactive work that takes a few minutes.
Use Slurm for: long training runs, batch jobs, anything needing a GPU.

Part 6: Submitting Jobs with Slurm

What Is Slurm?

Slurm is a job scheduler. Instead of running your code directly, you write a small script that describes your job (how long, how many CPUs, whether you need a GPU) and Slurm runs it for you on a compute node when resources are available.

The basic workflow is:

1. Write a job script (a text file with .slurm extension)
2. Submit it with sbatch
3. Wait for it to finish
4. Read the output file

Creating a Job Script

Use nano (a simple text editor in the terminal) to create a script:

```
nano job.slurm
```

Type the following and save with Ctrl+O, Enter, then Ctrl+X to exit:

```
#!/bin/bash
#SBATCH --job-name=my_first_job
#SBATCH --time=00:30:00
#SBATCH --nodes=1
#SBATCH --ntasks=1
#SBATCH --cpus-per-task=4
#SBATCH --mem=8G

module load Python
python myscript.py
```

Here is what each line means:

Command	What it does
`--job-name`	A friendly name to identify your job in the queue
`--time=00:30:00`	Maximum time allowed (HH:MM:SS). Job is killed if exceeded.
`--nodes=1`	Number of machines to use (almost always 1 for Python)
`--ntasks=1`	Number of parallel tasks (1 for a single Python script)
`--cpus-per-task=4`	How many CPU cores to allocate
`--mem=8G`	How much RAM to allocate (e.g. 8 gigabytes)

Command	What it does
`--partition=name`	The queue to submit to — clusters have different partitions (e.g. short, long, gpu). Check your cluster's docs.
`--account=name`	Your project/billing account — required on some HPC systems. Your welcome email will specify this.

Tip: Cluster-specific settings

Every HPC cluster has its own partition names, time limits, and account requirements.
Always check your cluster's documentation or welcome email for the correct values.
Common commands to explore your cluster: 'sinfo' (list all partitions) and 'sacctmgr show user' (see your accounts).

Useful Slurm Exploration Commands

Before submitting jobs, these commands help you understand what resources your cluster has:

Command	What it does
`sinfo`	List all partitions (queues) and how many nodes are idle/busy
`squeue`	Show all currently running and queued jobs on the cluster
`squeue -u username`	Show only your own jobs
`sacctmgr show user`	Show your account and billing group information
`module avail`	List all software modules available to load
`module list`	Show which modules you currently have loaded
`srun --partition=gpu --gpus=1 --pty bash`	Start an interactive GPU session for testing

Interactive sessions with srun

Instead of submitting a batch job, you can request an interactive shell on a compute node with srun.
This lets you run commands, test scripts, and use nvidia-smi in real time.
Example: srun --partition=gpu --gpus=1 --time=00:30:00 --pty bash
Type 'exit' when you are done to release the resources.

Submitting Your Job

```
sbatch job.slurm
```

You will see a message like: "Submitted batch job 12345". The number is your job ID — keep it handy.

Checking Job Status

```
squeue -u username
```

This shows all your jobs and their status. Common status codes:

Command	What it does
`PD`	Pending — waiting for resources to become available
`R`	Running — your job is currently executing
`CG`	Completing — job is finishing up
`(no entry)`	Done — your job finished (check the output file)

Reading the Output

When your job finishes, Slurm creates an output file named slurm-JOBID.out in your current folder:

```
cat slurm-12345.out
```

This file contains everything your Python script printed to the screen (stdout and stderr). If something went wrong, the error message will be here.

Cancelling a Job

If you need to stop a running or pending job:

```
scancel 12345
```

Part 7: Running GPU Jobs

GPU jobs follow the same pattern as CPU jobs, but you need to request the GPU partition and specify how many GPUs you need. Here is an example GPU job script:

```
#!/bin/bash
#SBATCH --job-name=gpu_training
#SBATCH --time=02:00:00
#SBATCH --partition=gpu
#SBATCH --nodes=1
#SBATCH --ntasks=1
#SBATCH --cpus-per-task=8
```

```
#SBATCH --mem=32G
#SBATCH --gpus=1

module load Python
module load CUDA

python train.py
```

The key differences from a CPU job are:

- --partition=gpu tells Slurm to use the GPU partition (the name may vary — check your cluster's documentation)
- --gpus=1 requests one GPU
- module load CUDA loads the GPU libraries needed by PyTorch/TensorFlow

Tip: Check GPU availability
GPU nodes are in high demand. If your job is stuck as PD (pending) for a long time, try requesting fewer GPUs or a shorter time limit.

Monitoring GPU Usage — nvidia-smi

Once your job is running on a GPU node, you can check how the GPU is being used with the nvidia-smi command. This is useful for debugging whether your code is actually using the GPU, and how much memory it is consuming.

Run a basic GPU status snapshot:

```
nvidia-smi
```

This shows a table with each GPU, its temperature, power draw, memory used vs total, and any running processes. Here is what a typical output looks like:

```
+-----------------------------------------------------------------------------
----+
| NVIDIA-SMI 525.85   Driver Version: 525.85   CUDA Version: 12.0
|
|-------------------------------+----------------------+----------------------
----+
| GPU  Name        Temp   Perf  Pwr:Usage/Cap |  Memory-Usage  | GPU-Util
|
|   0  A100-SXM4   42C    P0    68W / 400W    | 12543MiB/40960MiB | 87%
|
+-----------------------------------------------------------------------------
----+
```

Key columns to check:

Command	What it does
`GPU-Util`	Percentage of time the GPU is actively computing — should be high (>70%) during training
`Memory-Usage`	GPU RAM used vs total — if this hits 100%, your job will crash with OOM (Out of Memory) error
`Temp`	GPU temperature in Celsius — normal range is 30-85°C
`Pwr:Usage/Cap`	Power draw — useful for estimating job cost

Useful nvidia-smi Variants

Command	What it does
`nvidia-smi`	One-time snapshot of all GPUs
`nvidia-smi -l 2`	Live view, refreshes every 2 seconds (Ctrl+C to stop)
`nvidia-smi --query-gpu=utilization.gpu,memory.used,memory.total --format=csv`	Print only GPU %, memory used, and total as CSV
`nvidia-smi -L`	List all available GPUs and their names
`watch -n 2 nvidia-smi`	Alternative live view using the watch command

When to run nvidia-smi

nvidia-smi can only be run on a compute node that has a GPU — not on the login node.
Use it inside a Slurm interactive session (srun --partition=gpu --gpus=1 --pty bash) or add it to your job script to log GPU stats to the output file.
If GPU-Util stays near 0% during training, your code may not be using the GPU — check that you are calling .to(device) or .cuda() on your model and tensors.

Part 8: Organising Your Project

A clean project structure makes it much easier to manage scripts, data, and outputs. Here is a recommended layout:

```
my_project/
   data/          <- put your datasets here
   scripts/       <- your Python files
   outputs/       <- results, logs, saved models
   job.slurm      <- your Slurm job script
   requirements.txt  <- Python packages your code needs
```

To install Python packages listed in requirements.txt:

```
pip install -r requirements.txt --user
```

Part 9: Using Git on the Cluster

Git is a version control system — it tracks changes to your code over time, lets you collaborate with others, and makes it easy to bring your project onto the cluster directly from GitHub or GitLab.

> **Why use Git on a cluster?**
> Instead of uploading files with scp every time you make a change, you can push from your laptop to GitHub and then pull on the cluster. This keeps everything in sync and gives you a full history of your work.

First-Time Setup

Tell Git who you are. You only need to do this once per machine:

```
git config --global user.name "Your Name"
git config --global user.email "you@example.com"
```

Cloning a Repository

Cloning downloads an existing project from GitHub/GitLab onto the cluster:

```
git clone https://github.com/yourname/yourproject.git
```

This creates a folder called yourproject/ with all the code inside. You can then cd into it and start working.

The Everyday Git Workflow

After making changes to your code, you go through three steps to save and share them:

5. Check what has changed

```
git status
```

6. Stage the files you want to save

```
git add myscript.py          # stage one file
git add .                    # stage ALL changed files
```

7. Commit — save a snapshot with a message

```
git commit -m "Add training loop for experiment 1"
```

8. Push — upload your commit to GitHub/GitLab

```
git push
```

Good commit messages matter
Write a short message that describes WHAT changed and WHY.
Good: "Fix learning rate scheduler bug"
Bad: "stuff" or "changes"

Getting the Latest Changes — pull

If someone else (or you, from your laptop) pushed new code, sync the cluster with:

```
git pull
```

This fetches and merges the latest changes from the remote repository into your current folder.

Checking History and Differences

Command	What it does
`git log --oneline`	Show a compact list of recent commits
`git diff`	Show what you changed but haven't staged yet
`git diff --staged`	Show what is staged and ready to commit
`git show COMMITID`	Show the changes introduced by a specific commit

Branches — Working Safely in Parallel

Branches let you experiment without breaking your main working code. Think of a branch as a separate copy of the project where you can try things out.

Command	What it does
`git branch`	List all branches
`git branch experiment1`	Create a new branch called experiment1
`git checkout experiment1`	Switch to that branch
`git checkout -b new-feature`	Create AND switch in one step
`git checkout main`	Switch back to the main branch
`git merge experiment1`	Merge experiment1 into the current branch

Recommended habit
Always create a new branch for each experiment or feature. Keep main clean and working. This way you can always go back to a known good state.

Ignoring Files — .gitignore

Some files should never be committed — large datasets, trained model weights, cached files. Create a .gitignore file in your project root to exclude them:

```
# inside .gitignore
data/
outputs/
*.pt
*.pth
__pycache__/
.ipynb_checkpoints/
```

Git will automatically skip any file or folder matching these patterns.

Undoing Mistakes

Command	What it does
`git restore file.py`	Discard unsaved changes to a file (back to last commit)
`git restore --staged file.py`	Unstage a file (undo git add)
`git revert COMMITID`	Create a new commit that undoes a past commit (safe)
`git stash`	Temporarily set aside uncommitted changes
`git stash pop`	Bring stashed changes back

Warning — avoid git reset --hard

git reset --hard permanently discards uncommitted changes. There is no undo. Use git restore or git stash instead when in doubt.

Git Quick Reference Summary

Command	What it does
`git clone URL`	Download a repository
`git status`	See what has changed
`git add .`	Stage all changes
`git commit -m "message"`	Save a snapshot
`git push`	Upload commits to GitHub/GitLab
`git pull`	Download and merge latest changes
`git log --oneline`	View recent commits
`git branch name`	Create a branch
`git checkout name`	Switch to a branch
`git merge name`	Merge a branch into current
`git restore file`	Discard changes to a file
`git stash / git stash pop`	Shelve and restore uncommitted changes

Part 10: Common Mistakes and How to Avoid Them

Watch out for these!

Running heavy code on the login node — always use sbatch or tmux on a compute node instead.
Forgetting --partition=gpu for GPU jobs — without this, Slurm won't give you a GPU.
Setting --time too short — if your job runs out of time, it is killed mid-way. Add some buffer.
Not checking the .out file when something goes wrong — all error messages are stored there.
Closing the terminal without detaching from tmux — always press Ctrl+B then D before closing.
Running nvidia-smi on the login node — it only works on GPU compute nodes.
GPU-Util showing 0% during training — means your model is not on the GPU; check .to(device) calls.

Quick Reference — All Commands

Connecting & Uploading Files

Command	What it does
`ssh user@hpc.university.edu`	Connect to the HPC cluster
`scp file.py user@hpc.university.edu:~/`	Upload a file to the cluster
`scp -r folder/ user@hpc.university.edu:~/`	Upload a whole folder to the cluster

Terminal Navigation

Command	What it does
`ls / ls -lh`	List files (with details)
`cd foldername / cd ..`	Enter folder / go up a level
`cd ~`	Go to home folder
`pwd`	Show current location
`mkdir name`	Create a folder
`rm file / rm -r folder`	Delete file / delete folder
`cat file.txt`	Show file contents
`head -n 20 file`	Show first 20 lines
`tail -n 20 file`	Show last 20 lines

Copying & Moving Files

Command	What it does
`cp file.py copy.py`	Copy a file
`cp -r folder/ backup/`	Copy an entire folder
`mv old.py new.py`	Rename a file
`mv file.py folder/`	Move a file into a folder

vim Editor

Command	What it does
`vim file.py`	Open a file in vim
`i`	Enter Insert mode (start typing)
`Esc`	Return to Normal mode
`:wq`	Save and quit
`:q!`	Quit without saving
`dd`	Delete current line
`u`	Undo
`/term`	Search for text

nano Editor

Command	What it does
`nano file.py`	Open a file in nano
`Ctrl+O, Enter`	Save
`Ctrl+X`	Exit
`Ctrl+W`	Search

Git

Command	What it does
`git clone URL`	Download a repository
`git status`	See what changed
`git add . / git add file`	Stage all / specific file
`git commit -m "msg"`	Save a snapshot
`git push / git pull`	Upload / download changes
`git log --oneline`	View recent commits
`git branch name`	Create a branch
`git checkout name`	Switch branch
`git merge name`	Merge branch into current
`git restore file`	Discard changes to a file
`git stash / git stash pop`	Shelve / restore uncommitted work

tmux Sessions

Command	What it does
`tmux new -s name`	Start a new session called 'name'
`Ctrl+B, D`	Detach (leave session running)
`tmux attach -t name`	Reconnect to a session
`tmux ls`	List all sessions
`tmux kill-session -t name`	Delete a session

Slurm Job Management

Command	What it does
`sbatch job.slurm`	Submit a job
`squeue -u username`	Check your job status
`scancel JOBID`	Cancel a job
`cat slurm-JOBID.out`	Read job output
`sinfo`	List all partitions and their availability
`sacctmgr show user`	Show your accounts/billing groups
`srun --pty bash`	Start an interactive compute session
`module load Python`	Load Python module
`module load CUDA`	Load GPU libraries
`module avail`	List all available software modules

GPU Monitoring — nvidia-smi

Command	What it does
`nvidia-smi`	Snapshot of all GPUs (usage, memory, temperature)
`nvidia-smi -l 2`	Live view, refreshes every 2 seconds
`nvidia-smi -L`	List all available GPUs and their names
`watch -n 2 nvidia-smi`	Alternative live view using watch

Command	What it does
`nvidia-smi --query-gpu=utilization.gpu,memory.used,memory.total --format=csv`	Print GPU %, memory used/total as CSV

You are ready to go!

Start small: connect to the cluster, upload a simple Python script, and submit a CPU job first. Once that works, move to GPU jobs. If anything goes wrong, the .out file is your best friend. Good luck!