Multi-Node CPU DDP `torchrun` on SLURM

(Assistant Lecturer: Eng. Ahmed Métwalli)

(Last updated: September 3, 2025)

Goal. In this lab, you will run distributed CPU training across multiple SLURM nodes using `torch.distributed` (Gloo backend), `torchrun`, and HuggingFace `Trainer`. You will stage dependencies for offline nodes, verify per-node environments, and interpret structured logs to confirm ranks, world size, and correct rendezvous.

The codes are updated: `train_tiny.py`, `side_shell_pr.sh`, and `req.sh`. This handout explains how they work together, why each design choice was made, and how to validate every step.

1 Architecture Overview (What Runs Where)

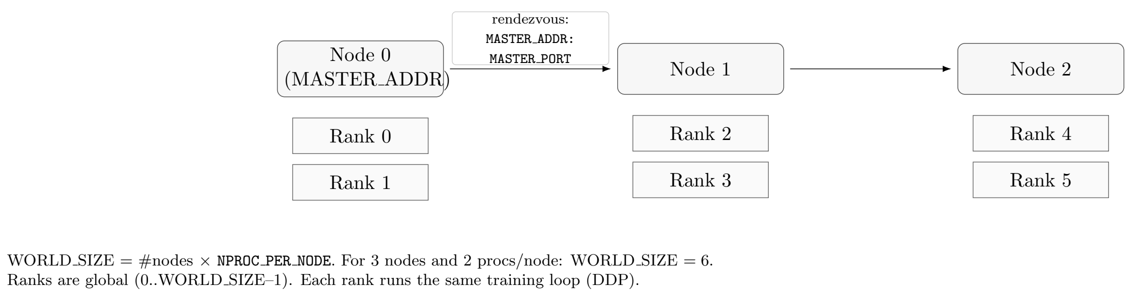

Figure 1: Ranks per node, rendezvous, and world size.

- *Hosts.* Each "Node" is a SLURM node. The leftmost node is chosen as the rendezvous server: we export `MASTER_ADDR` and `MASTER_PORT`; PyTorch's `c10d` store (Gloo on CPU) uses these to let all processes find each other (`init_method="env://"`).
- *Ranks beneath* On each node, `torchrun` spawns `NPROC_PER_NODE` worker processes. Each worker is a **rank** with a unique **global** `RANK` in 0..`WORLD_SIZE` − 1. In the example: Node 0 has ranks 0–1, Node 1 has 2–3, Node 2 has 4–5.
- *World size.* `WORLD_SIZE` = `nnodes` × `nproc_per_node`. For 3 nodes × 2 procs/node, `WORLD_SIZE` = 6.
- *Which IDs exist?* Each process gets:
  - `RANK` (global, unique across all nodes),
  - `LOCAL_RANK` (index of the process on its node: 0..`NPROC_PER_NODE` − 1),
  - `NODE_RANK` (which node this is: 0..`nnodes` − 1).

  These are set by `torchrun` via environment variables and consumed by `init_process_group`.
- *What runs on each rank?* `train_tiny.py` executes identically on every rank: it loads the tokenizer and data shard, builds the tiny BERT, and performs a forward/backward step. At the end of each step, DDP (via Gloo) **averages gradients** across all ranks, so the model stays in sync. Only rank 0 saves checkpoints.

- *Arrows between nodes.* They indicate the DDP control/data connections established after rendezvous at `MASTER_ADDR:MASTER_PORT`. The label sits above the first hop to remind you where rendezvous happens; the actual group spans all nodes.

Node	**node_rank**	**local_rank → global RANK (this figure)**
Node 0	0	$0 \to 0,\ 1 \to 1$
Node 1	1	$0 \to 2,\ 1 \to 3$
Node 2	2	$0 \to 4,\ 1 \to 5$

Table 1: *

Mapping of per-node processes to global ranks when `nnodes=3`, `nproc_per_node=2`.

What is a *rank*? A *rank* is a single Python **process** in the DDP job. DDP assigns:

- **Global** `RANK` $\in \{0, \ldots,$ `WORLD_SIZE`$-1\}$ — unique across the whole job. Rank 0 is the "leader" we use for logging/checkpoints.
- **Local** `LOCAL_RANK` $\in \{0, \ldots,$ `NPROC_PER_NODE` $- 1\}$ — index of the process on its node.
- **Node** `NODE_RANK` $\in \{0, \ldots,$ `nnodes` $- 1\}$ — which node you are on.

They satisfy

$$\texttt{WORLD\_SIZE} = \texttt{nnodes} \times \texttt{nproc\_per\_node}, \qquad \texttt{RANK} = \texttt{NODE\_RANK} \times \texttt{NPROC\_PER\_NODE} + \texttt{LOCAL\_RANK}.$$

In our figure (`nnodes`=3, `nproc_per_node`=2) the global ranks are: Node 0 → RANK 0,1; Node 1 → RANK 2,3; Node 2 → RANK 4,5. All six ranks run identical training loops; after each backward pass, DDP (Gloo backend) averages gradients across all ranks so the models stay in sync. Only RANK 0 saves to `./tiny_out`.

Not to confuse: a rank is *not* a CPU core or a SLURM task. In this lab we request `--ntasks-per-node=1` with SLURM, and then `torchrun` creates `NPROC_PER_NODE=2` *ranks* (processes) per node; each rank may use a few CPU threads (e.g., `OMP_NUM_THREADS=2`).

2 Requirements & Offline Staging (`req.sh`)

2.1 What `req.sh` sets up and why

- **Offline cache root** at `$HOME/offline_repo_py311`: holds wheels or a vendorized `pkgs` tree so worker nodes do not need internet.
- **Pinned versions** in `requirements_local.txt`:
  - `torch==2.3.1+cpu` (CPU wheels), `transformers==4.41.2`, `datasets==2.19.0`,
  - `numpy==1.26.4` (*avoids* a known formatting issue with `datasets`+NumPy 2.x),
  - `pyarrow==16.1.0`, `tokenizers==0.19.1`, etc.
- **Two modes:** *(a) Online on login node* → directly install into `pkgs`; *(b) Offline cluster* → `pip download` wheels to `wheels/` then `pip install --no-index` into `pkgs`.

2.2 Run it once on the login node

```
bash req.sh
# Verify that you now have:
# $OFFLINE_ROOT/pkgs/ (many site-packages)
# $OFFLINE_ROOT/wheels/ (if you used the offline flow)
```

Listing 1: Prepare local offline/online dependencies

2.3 Why NumPy 1.26.4?

`datasets==2.19.0` can trip on NumPy 2.x formatting paths. Pinning to `1.26.4` keeps tokenization/arrow formatting stable. (Our launcher also contains a *defensive* patch that gracefully adapts if a node happens to have NumPy 2.x.)

3 The Training Script (`train_tiny.py`) — What to Notice

3.1 CPU-only safety patch

If CUDA is absent, some utilities (e.g., Accelerate) might try `torch.cpu.set_device()`. We *monkey-patch* it to a no-op on pure CPU nodes to prevent spurious errors:

```
# if not torch.cuda.is_available(): torch.cpu.set_device = lambda *_: None
```

Listing 2: Concept (already in your code, no action needed)

3.2 Distributed init and debugging prints

- Backend: `gloo` on CPU; `nccl` on GPU (not used here).
- Rendezvous: `env://` uses `MASTER_ADDR`, `MASTER_PORT`, `WORLD_SIZE`, `RANK`.
- **Debug line**: every process prints `[RANK r] WORLD_SIZE=w`.

3.3 Model & data

- Tokenizer: `google/bert_uncased_L-2_H-128_A-2`.
- Tiny BERT config (`hidden_size=64`, `layers=2`, `heads=2`, `intermediate=256`, `max_pos=256`).
- Dataset: `ag_news` (subset to `--subset 2000` for fast CPU runs).
- `Trainer` + DDP: `ddp_find_unused_parameters=False`, `logging_steps=10`.
- Rank 0 saves the model to `--out` (default `./tiny_out`).

4 The Launcher (`launch_tiny.sh`) — Phases & Rationale

4.1 Resource allocation (outer `salloc`)

If not already inside a job, the script re-execs itself under `salloc`:

- `--partition=torch` (*example partition name*),
- `--nodes=$JOB_NODES` (e.g., 10),
- `--ntasks-per-node=1` (one launcher task per node),
- `--cpus-per-task=4` (room for tokenization/BLAS),
- `--time=00:30:00`.

4.2 Rendezvous & world size

After allocation:

```
export MASTER_ADDR=$(scontrol show hostnames "$SLURM_NODELIST" | head -n1)
export MASTER_PORT=29500
export NNODES=${SLURM_NNODES:-$(scontrol show hostnames "$SLURM_NODELIST" | wc -l)}
```

Interpretation: The first hostname acts as rendezvous server. WORLD_SIZE = `NNODES` × `NPROC_PER_NODE`.

4.3 Preflight visibility check (critical!)

We must ensure each node sees:

1. the project directory (`$PROJECT_DIR`),
2. the offline packages (`$OFFLINE_ROOT/pkgs`).

The script runs:

```
VIS_REPORT=$(srun --export=ALL -N "$NNODES" -n "$NNODES" bash -lc '
  echo -n "$HOSTNAME "
  [[ -d "'"$PROJECT_DIR"'" ]] && printf "proj=ok " || printf "proj=missing "
  [[ -d "'"$OFFLINE_ROOT"'/pkgs" ]] && printf "pkgs=ok\n" || printf "pkgs=missing\n"
')
echo "$VIS_REPORT" | tee "$HOME/slurm_logs/preflight.$SLURM_JOB_ID.txt"
```

Listing 3: Node visibility report

Read this file first: `slurm_logs/preflight.$SLURM_JOB_ID.txt`. You should see lines like `hpc07 proj=ok pkgs=ok` for *every* node.

4.4 Conditional staging with `sbcast`

If any node reports `proj=missing` or `pkgs=missing`, the launcher:

1. tars `$PROJECT_DIR` and `pkgs/`,
2. broadcasts them to all nodes via `sbcast` into `/tmp`,

3. unpacks under a per-job temp root (`$SLURM_TMPDIR` or fallback).

At runtime, each task selects either the shared path or the staged copy:

```
RUN_TMP="${SLURM_TMPDIR:-/tmp/$USER/slurm_$SLURM_JOB_ID}"
RUN_OFFLINE_ROOT="$OFFLINE_ROOT"
RUN_PROJECT_DIR="$PROJECT_DIR"
[[ -d "$RUN_PROJECT_DIR" ]] || RUN_PROJECT_DIR="$RUN_TMP/$(basename "$PROJECT_DIR")"
[[ -d "$RUN_OFFLINE_ROOT/pkgs" ]] || RUN_OFFLINE_ROOT="$RUN_TMP"
```

Listing 4: Per-node runtime roots (inside launcher)

This guarantees all nodes have a working copy without NFS hiccups.

4.5 Environment hygiene (per-node)

Before Python starts, we export environment to make the run self-contained and deterministic:

- `PYTHONPATH=RUN_OFFLINE_ROOT/pkgs:$RUN_PROJECT_DIR:$PYTHONPATH`
- `HF_HOME=$HOME/.cache/hf`, `TRANSFORMERS_OFFLINE=1`, `HF_DATASETS_OFFLINE=1`
- `OMP_NUM_THREADS=2` (keep BLAS reasonable under DDP)
- `TOKENIZERS_PARALLELISM=false`
- `PYTORCH_DIST_BACKEND=gloo`

4.6 Sanity probe (per node)

We verify that Python can import `torch`, `transformers`, `datasets`, etc. The launcher prints a one-line summary:

```
srun ... python /tmp/_sanity.py
```

Listing 5: Expect lines like: NODE hpc09 OK -¿ PY 3.11 torch 2.3.1 tfm 4.41.2 ...

If a node fails here: check that it received staged copies; confirm Python is available; re-run with `--exclude=<node>` via `SALLOC_OPTS`.

4.7 The wrapper and optional NumPy 2.x patch

A tiny wrapper (`/tmp/_run_wrapper.py`) inserts `pkgs/` and project path into `sys.path`, and *if* NumPy 2.x is detected, applies a safe adapter to the `datasets` formatter. This keeps older `datasets` versions working even if a worker node has a newer NumPy by accident.

4.8 The distributed launch

Finally, we call:

```
python -m torch.distributed.run \
  --nnodes="$NNODES" \
  --nproc_per_node="$NPROC_PER_NODE" \
  --rdzv_backend=c10d \
  --rdzv_endpoint="$MASTER_ADDR:$MASTER_PORT" \
  --rdzv_id="$SLURM_JOB_ID" \
```

```
  --node_rank="$SLURM_NODEID" \
  /tmp/_run_wrapper.py --subset 2000 --epochs 3
```

Listing 6: Key torch.distributed.run arguments

Interpretation: `node_rank` is assigned by SLURM, `WORLD_SIZE = nnodes * nproc_per_node`, and rendezvous is shared via the master host:port.

5 Running the Lab

5.1 One-time setup

```
bash req.sh
```

Listing 7: Install deps into offline cache (login node)

5.2 Launch (interactive job)

```
bash ~/launch_tiny.sh
# Optional: exclude flaky nodes
# SALLOC_OPTS="--exclude=hpc42" bash ~/launch_tiny.sh
```

Listing 8: Start the distributed job (10 nodes, 2 procs/node)

5.3 What to watch on screen

1. Allocation line: `[alloc] job=... nodes=...`
2. Rendezvous line: `NNODES=... MASTER_ADDR=... MASTER_PORT=...`
3. Preflight report: each host prints `proj=ok pkgs=ok` (or triggers staging).
4. Sanity probe: each host prints `NODE <name> OK -> PY 3.11 torch 2.3.1 ...`
5. Torchrun banner: `nnodes=?, nproc_per_node=?, node_rank=?, rdzv=?`
6. Training logs: every rank periodically logs via HF Trainer; each process prints `[RANK r] WORLD_SIZE=w`.

6 Logs and How To Read Them Carefully

6.1 Files generated

All stdout/stderr from `srun` phases are captured to:

```
$HOME/slurm_logs/<jobname>.<jobid>.<nodename>.<taskid>.out
```

Plus one preflight summary: `slurm_logs/preflight.$SLURM_JOB_ID.txt`.

6.2 Minimal checklist

1. **Preflight:** Open the preflight file; ensure *every* node is `proj=ok pkgs=ok`.
   If not: confirm that staging ran (you should see "staging via sbcast" on screen), then check per-node `.out` to see lines like `$HOSTNAME staged -> /tmp/...`.

2. **Sanity probe:** In each node's `.out`, find the `NODE <name> OK -> PY ... torch ... tfm ... datasets ...`. All must say `OK`.

3. **World size:** In any training `.out`, grep for `[RANK`. Confirm that the largest rank equals `WORLD_SIZE-1`.
   Expected: `WORLD_SIZE = NNODES * NPROC_PER_NODE`.

4. **Saving:** Only rank 0 will save artifacts; look for `Saving model to ./tiny_out`.

6.3 Useful greps

```
# Show preflight summary
cat ~/slurm_logs/preflight.$SLURM_JOB_ID.txt

# List all node-task outputs for this job
ls -1 ~/slurm_logs/*.$SLURM_JOB_ID.*.out

# Check ranks and world size
grep -h "\[RANK" ~/slurm_logs/*.$SLURM_JOB_ID.*.out | sort -u

# Find any failure signatures
grep -HiE "Traceback|ERROR|RuntimeError|OSError" ~/slurm_logs/*.$SLURM_JOB_ID.*.out
```

Listing 9: Quick log triage

7 Design Choices (Deep Dive)

7.1 Why Gloo + `env://` on CPU

Gloo is PyTorch's reliable CPU backend. Using `env://` keeps launch simple under SLURM: we export rendezvous variables once and let `torchrun` propagate them to all ranks.

7.2 Why `--ntasks-per-node=1` but `NPROC_PER_NODE=2`

We place one SLURM task per node (clean lifecycle and logging), then let `torchrun` spawn 2 processes per node (total ranks per node = 2). This makes failure handling (`--kill-on-bad-exit=1`) predictable and reduces SLURM scheduling overhead.

7.3 Why `OMP_NUM_THREADS=2`

On CPU, too many BLAS threads per process can degrade performance under DDP. With 2 ranks/node and light models, `OMP_NUM_THREADS=2` balances tokenization and linear algebra without oversubscription.

7.4 Why conditional staging with `sbcast`

Some nodes may not see shared storage or may mount it inconsistently. Broadcasting a tarball to `/tmp` ensures a consistent, job-scoped snapshot with low latency.

7.5 Defensive NumPy patch

If a worker accidentally has NumPy 2.x, an adapter in the wrapper rebinds a `datasets` formatter method to keep `np.asarray` semantics. This avoids mid-run crashes due to heterogeneous node images.

8 Verification, Metrics, and Expected Outputs

8.1 Rank banner (from `train_tiny.py`)

```
[RANK 0] WORLD_SIZE=20
[RANK 7] WORLD_SIZE=20
...
```

Here, 10 nodes $\times$ 2 procs/node $\Rightarrow$ WORLD_SIZE = 20.

8.2 HF Trainer logs

Every `logging_steps` steps (e.g., 10), each rank prints a short report. You may see throughput stabilize after the first few steps as tokenizers warm up.

8.3 Saved artifacts (rank 0)

- `tiny_out/config.json`, `pytorch_model.bin`, `tokenizer.json`, etc.
- Check modified time to confirm it was produced by the current run.

9 Troubleshooting (Most Common Issues)

Symptom	Fix / Explanation
Some nodes show `proj=missing` or `pkgs=missing`	This is normal on clusters without shared storage. The launcher will stage via `sbcast`. Confirm per-node `staged -> /tmp/...` messages.
`Traceback` about `datasets` and NumPy 2.x	Pin NumPy to 1.26.4 in `req.sh`. The wrapper also applies a safety patch; ensure the wrapper ran (it prints a "patch applied" line).
`Address already in use` on `MASTER_PORT`	Change `MASTER_PORT` in the launcher (e.g., any unreserved port > 1024).
Ranks hang at init	Check that `MASTER_ADDR` resolves from all nodes. Verify firewalls. Ensure `rdzv_endpoint` matches the allocation.
Slow training or CPU spikes	Lower `OMP_NUM_THREADS` to 1, or reduce `NPROC_PER_NODE`. Confirm `--cpus-per-task` leaves headroom.
Model not saved	Only rank 0 saves. Look for `[RANK 0]` logs. Ensure disk write permissions in `--out`.

10 Lab Tasks & Checkoff

1. Run `req.sh`. Show your `pkgs/` tree exists and includes `torch`, `transformers`, `datasets`, `numpy`.
2. Launch training on 3 nodes with `NPROC_PER_NODE=2`. Capture:
   - Preflight summary (`proj=ok pkgs=ok` per node).
   - At least one node's sanity probe line.
   - A snippet with `[RANK r] WORLD_SIZE=w` proving correct world size.
3. Open `tiny_out/` and list saved files (rank 0).
4. (Optional) Repeat with 5 nodes and compare time/epoch.

11 Reference: Key Variables at a Glance

Name	Set In	Meaning
MASTER_ADDR	launcher	Hostname for rendezvous (first node).
MASTER_PORT	launcher	TCP port for rendezvous (e.g., 29500).
NNODES	launcher	Number of allocated nodes.
NPROC_PER_NODE	launcher	# of ranks per node (torchrun).
WORLD_SIZE	implicit	NNODES * NPROC_PER_NODE.
node_rank	torchrun	Rank of this node among nodes (0-based).
RANK	torchrun	Global rank of a process (0..WORLD_SIZE-1).
RUN_PROJECT_DIR	launcher	Per-node project path (shared or staged).
RUN_OFFLINE_ROOT	launcher	Per-node offline root (shared or staged).
OMP_NUM_THREADS	launcher	BLAS threading per process (e.g., 2).
TRANSFORMERS_OFFLINE	launcher	Disables net calls in `transformers`.
HF_DATASETS_OFFLINE	launcher	Disables net calls in `datasets`.

12 Appendix: Commands You Will Actually Type

A. Prepare dependencies

```
bash req.sh
```

B. Start a 10-node, 2-proc/node run

```
bash ~/launch_tiny.sh
# or exclude one bad node:
# SALLOC_OPTS="--exclude=hpc42" bash ~/launch_tiny.sh
```

C. Inspect logs

```
cat ~/slurm_logs/preflight.$SLURM_JOB_ID.txt
ls ~/slurm_logs/*.$SLURM_JOB_ID.*.out | wc -l
grep -h "\[RANK" ~/slurm_logs/*.$SLURM_JOB_ID.*.out | sort -u | head
```

Questions? E-mail ametwalli@aast.edu