Training with Parallelism

(Assistant Lecturer: Eng. Ahmed Métwalli) (Last updated: August 2, 2025)

7 CPU Parallelism in `salloc`: Tasks vs. Threads

On CPU-based clusters, performance depends heavily on **how you distribute work across cores**. SLURM supports several modes of parallel execution using combinations of `--ntasks`, `--cpus-per-task`, and `--nodes`. Here's how to choose the right one.

A. Task Parallelism (Multiple Processes)

Definition: You run *multiple independent tasks*, each as a separate process — often using MPI (Message Passing Interface), multiprocessing in Python, or parallel shell commands.

```
salloc --nodes=1 --ntasks=4 --cpus-per-task=1
srun ./my_program
```

Listing 1: Example: 4 tasks, each on 1 CPU

- Launches 4 separate processes.
- Ideal for simulations, map-reduce jobs, or embarrassingly parallel loops.
- Each process can run independently on different cores or even nodes.

B. Thread Parallelism (Single Task, Multi-threaded)

Definition: You run *one task* that spawns multiple threads internally — common in OpenMP, TensorFlow, NumPy, and MKL-powered code.

```
salloc --nodes=1 --ntasks=1 --cpus-per-task=8
srun ./my_openmp_app
```

Listing 2: Example: 1 task with 8 threads

- A single process runs, with 8 threads using shared memory.
- Ideal for BLAS libraries, PyTorch CPU inference, or large matrix ops.
- You must ensure your code uses multithreading (e.g., OpenMP or BLAS).
- Optionally export `OMP_NUM_THREADS=8` for OpenMP compliance.

C. Hybrid Parallelism (Multiple Tasks, Each with Threads)

Definition: You run *multiple tasks*, each of which uses multiple threads. This is typical in MPI + OpenMP programs or hybrid scientific workloads.

```
salloc --nodes=1 --ntasks=2 --cpus-per-task=4
srun ./hybrid_solver
```

Listing 3: Example: 2 tasks, 4 threads each

- Runs 2 processes, each with 4 CPU threads (total 8 CPUs).
- Best for distributed workloads where each MPI rank uses threading.
- Requires careful control of thread affinity and environment vars:

  ```
  export OMP_NUM_THREADS=4
  export MKL_NUM_THREADS=4
  ```

D. Comparison Table

Type	SLURM Flags	Use Case	Typical Scenario
Task Parallel	`--ntasks=N--cpus-per-task=1`	Independent workers	Grid search, Monte Carlo simulations, multiprocessing in Python.
Thread Parallel	`--ntasks=1--cpus-per-task=N`	Multithreaded libraries	Single-process apps using OpenMP, TensorFlow on CPU, NumPy or SciPy.
Hybrid	`--ntasks=M--cpus-per-task=N`	MPI + threads	Scientific codes combining distributed tasks and local multithreading.

Choosing the right model ensures better CPU utilization and avoids over- or under-subscription of cluster resources.

8 Why This Training Workflow Works (Step-by-Step Explainer)

The following setup enables reproducible, parallel CPU training using HuggingFace's Trainer API and PyTorch's native ‘torchrun‘ launcher. Below is a breakdown of each step and *why* it's done this way.

A. Preloading the Dataset and Tokenizer

```
python - <<'PY'
import os, datasets, transformers
os.environ["HF_HOME"] = os.path.expanduser("~/.cache/hf_tiny")
datasets.load_dataset("ag_news", split="train[:2000]",
                      cache_dir=f"{os.environ['HF_HOME']}/ds")
transformers.AutoTokenizer.from_pretrained(
    "google/bert_uncased_L-2_H-128_A-2",
    cache_dir=f"{os.environ['HF_HOME']}/tok")
print("Done pre-caching.")
PY
```

Listing 4: Pre-cache tokenizer and dataset to local HF cache

Why:

- HuggingFace downloads can bottleneck your first run or fail in offline clusters.
- We pre-cache datasets & tokenizers under `$HOME/.cache/hf_tiny` to prevent repeated downloads.

B. Selecting the Best Partition Programmatically (Optional)

```
# See Cluster/Recommender.py
python Recommender.py
```

Listing 5: Python script to recommend best CPU partition

Why:

- Automatically scans SLURM partitions to find idle CPU nodes.
- Picks the most suitable one while avoiding GPU and downed nodes.
- Useful in dynamic environments where availability changes minute to minute.

C. Allocate Resources for CPU Parallelism

```
salloc -N1 -n1 -c2 -p parallel --time=00:05:00 --exclusive
```

Listing 6: Interactive allocation with 2 tasks on 1 node

Why:

- One node (`-N1`), 1 task (`-n1`), 2 CPUs per task (`-c2`).
- '–exclusive' ensures you're the only user on that node (important for benchmarking or isolation).
- Works well for 'torchrun' which will spawn 2 processes using `--nproc_per_node=2`.

D. Activate the Environment and Set Rendezvous Info

```
source ~/llamaenv_local/bin/activate
cd ~/mrmito/project
export HF_HOME=$HOME/.cache/hf_tiny
export PYTHONPATH=$PWD:$PYTHONPATH
export MASTER_ADDR=$(hostname)
export MASTER_PORT=$((20000 + RANDOM % 10000))
```

Listing 7: Prepare the training environment

Why:

- `llamaenv_local` contains project-specific Python packages.
- Setting `PYTHONPATH` allows local imports like `labs.tiny.train_tiny`.
- Torch distributed needs a rendezvous (host + port); here we auto-generate one.

E. Launch Distributed Training with `torchrun`

```
torchrun \
  --nproc_per_node=2 \
  --rdzv_backend=c10d \
  --rdzv_endpoint=${MASTER_ADDR}:${MASTER_PORT} \
  labs/tiny/train_tiny.py \
  --subset 2000 --epochs 3
```

Listing 8: Run HuggingFace training with torchrun

Why:

- Launches 2 processes (1 per CPU core).
- `train_tiny.py` uses HuggingFace's `Trainer`, which detects `torch.distributed` backend and synchronizes gradients.
- The tiny BERT config (L2-H128) ensures fast CPU convergence with minimal memory.

F. Inside `train_tiny.py`: Distributed Strategy

The training script auto-initializes 'torch.distributed', loads and tokenizes AG News data, builds a minimal BERT model, and trains using HuggingFace's Trainer.

Highlights:

- `torch.distributed.init_process_group` enables multi-process CPU gradient syncing.
- Rank 0 process saves the model + tokenizer at the end.
- Can scale to more nodes by adjusting `--nproc_per_node` and `salloc` parameters.

This modular workflow ensures reproducibility, optimal CPU utilization, and clean training behavior across SLURM-managed environments.

G. Model Architecture: TinyBERT for Fast CPU Training

The model used in this workflow is a minimal BERT-based classifier defined by a custom configuration via HuggingFace's `BertConfig`. This configuration is optimized for:

- **Speed:** Can be trained in under 5 minutes on a multi-core CPU.
- **Memory:** Requires less than 500MB of RAM per process.
- **Scale:** Allows students to test parallelism without needing GPUs.

Configuration Summary

Parameter	Value
`hidden_size`	64
`num_hidden_layers`	2
`num_attention_heads`	2
`intermediate_size`	256
`max_position_embeddings`	256
`vocab_size`	30522 (Google TinyBERT tokenizer)
`num_labels`	4 (AG News categories)

This compact model has fewer than **1 million trainable parameters**, compared to:

- **BERT-Base:** 110M parameters
- **BERT-Tiny (official):** 4M parameters
- **Our model:** ˜0.9M parameters

Why This Configuration?

- **Educational focus:** Training can be finished quickly and repeatedly — ideal for labs or capstones.
- **Parallel testing:** Model is small enough to not bottleneck CPUs when testing data/model/pipeline parallelism.
- **Compatibility:** Fully supported by HuggingFace's Trainer, tokenizer API, and PyTorch DDP backend.

What You Learn by Training This Model

- How HuggingFace tokenizers integrate with PyTorch datasets.
- How to define and train custom Transformers with minimal config.
- How distributed training works under 'torch.distributed'.
- How SLURM scheduling and CPU resource allocation affect training time.

This architecture serves as a practical baseline before moving on to larger models like `TinyLlama` in later modules or the Capstone Project.