Distributed training. Slurm.

Plan

- Distributed training with torch
- Usefull techniques
- Slurm

Distributed Data Parallel with Lightning

Trainer class API

Methods

init

```
Trainer.__init__(*, accelerator='auto', strategy='auto', devices='auto', num_nodes=1, precision=None, logger=None,
callbacks=None, fast_dev_run=False, max_epochs=None, min_epochs=None, max_steps=-1, min_steps=None,
max_time=None, limit_train_batches=None, limit_val_batches=None, limit_test_batches=None, limit_predict_batches=None,
overfit_batches=0.0, val_check_interval=None, check_val_every_n_epoch=1, num_sanity_val_steps=None,
log_every_n_steps=None, enable_checkpointing=None, enable_progress_bar=None, enable_model_summary=None,
accumulate_grad_batches=1, gradient_clip_val=None, gradient_clip_algorithm=None, deterministic=None, benchmark=None,
inference_mode=True, use_distributed_sampler=True, profiler=None, detect_anomaly=False, barebones=False,
plugins=None, sync_batchnorm=False, reload_dataloaders_every_n_epochs=0, default_root_dir=None)
```

[SOURCE]

Customize every aspect of training via flags.

Distributed Data Parallel

- Start **N** processes, 1 for each gpu
- Split data into **N** parts
- Compute forward and backward on each process
- Syncronize and average gradients

Will it work to just write "python train.py" with PyTorch?

Some useful techniques

- Mixed precision and bfloat16 training
- Activation checkpointing
- ZeRO

Slurm:
Simple Linux Utility For Resource Management

- Open-source
- Fault-tolerant
- Higly scalable
- Developed since 2002
- Workload manager on about 60% of the TOP500 supercomputers
- Easy-to-extend with plugins

- Allocating resources
- Scheduling jobs
- Monitoring resources

- **slurmctld** – centralized manager
- **slurmd** – daemons waiting for work
- **slurmdbd** – database daemon

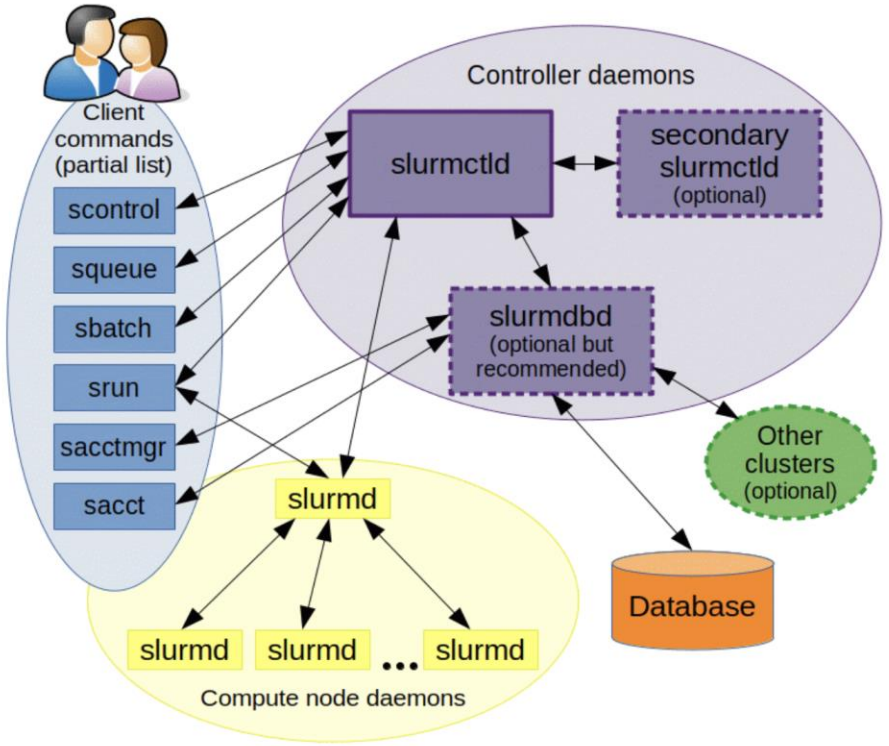

Figure 1. Slurm components

sinfo – View information about Slurm nodes and partitions

Partitions – logical groupings of compute nodes

Nodes can be partitioned by:

1. Hardware Type – for example cpu and gpu nodes
2. Access Permissions
3. Job durations

```
PARTITION   AVAIL  TIMELIMIT  NODES  STATE NODELIST
default*       up   infinite      8   idle node[01-08]
cpu            up   2-00:00:00  10   idle cpu-node[01-10]
gpu            up   1-00:00:00   4    mix gpu-node[01-04]
short          up   00:30:00     5   idle short-node[01-05]
long           up   7-00:00:00   3   alloc long-node[01-03]
senior         up   infinite     2   down senior-node[01-02]
```

scontrol :

1. view configuration and job/partition/node state

2. manage node states

3. manage jobs

Example:

```
malex26@buran:~/slurm_lection$ sudo scontrol update NodeName=buran State=DRAIN Reason="Scheduled maintance"
malex26@buran:~/slurm_lection$ scontrol show nodes buran
NodeName=buran Arch=x86_64 CoresPerSocket=1
   CPUAlloc=0 CPUEfctv=64 CPUTot=64 CPULoad=0.84
   AvailableFeatures=(null)
   ActiveFeatures=(null)
   Gres=(null)
   NodeAddr=buran NodeHostName=buran Version=21.08.5
   OS=Linux 5.15.0-87-generic #97-Ubuntu SMP Mon Oct 2 21:09:21 UTC 2023
   RealMemory=65536 AllocMem=0 FreeMem=398702 Sockets=64 Boards=1
   State=IDLE+DRAIN ThreadsPerCore=1 TmpDisk=0 Weight=1 Owner=N/A MCS_label=N/A
   Partitions=debug
   BootTime=2023-10-20T15:47:31 SlurmdStartTime=2023-10-21T01:03:22
   LastBusyTime=2023-10-21T04:27:18
   CfgTRES=cpu=64,mem=64G,billing=64
   AllocTRES=
   CapWatts=n/a
   CurrentWatts=0 AveWatts=0
   ExtSensorsJoules=n/s ExtSensorsWatts=0 ExtSensorsTemp=n/s
   Reason=Scheduled maintance [root@2023-10-21T04:27:53]

malex26@buran:~/slurm_lection$ sudo scontrol update NodeName=buran State=RESUME
malex26@buran:~/slurm_lection$ sinfo
PARTITION AVAIL  TIMELIMIT  NODES  STATE NODELIST
debug*       up   infinite      1   idle buran
malex26@buran:~/slurm_lection$
```

States:

1. **ALLOCATED –** is allocated for jobs
2. **IDLE –** is avaliable and waiting for jobs
3. **COMPLETING –** state between allocated and idle
4. **DOWN –** node is not available for use, mb failure
5. **DRAIN –** becoming unavailable for new jobs, but has running jobs to be completed
6. **DRAINED –** state after DRAIN, jobs completed
7. **UNKNOWN –** initial state of node
8. **MIXED –** has some resources allocated and some idle

Jobs - user-submitted workload that requests specified amount of cluster resources to execute a certain task / tasks

Most common requested resources:

- **Cpus (cores)**
- **Memory (RAM)**
- **GPU**
- **Nodes**
- **Wall time**

sbatch – submit job to cluster

Write your job script and job directives in file run_job.sh and execute "**sbatch run_job.sh**"

Job script Structure:

1. **SLURM Directives**

2. **Commands**

1. Job Name
2. Output and Error Files
3. Number of Nodes
4. Number of tasks
5. Number of CPUs
6. Number of GPUs
7. Amount of RAM
8. Wall Time
9. Partition
10. Email Notification

.....

job script generator

```
#!/bin/sh
#SBATCH --job-name=myJob
#SBATCH --output=myJob_output.txt
#SBATCH --error=myJob_error.txt
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=4
#SBATCH --cpus-per-task=4
#SBATCH --gpus-per-task=0
#SBATCH --mem=2G
#SBATCH --time=01:00:00
#SBATCH --partition=debug

echo "SLURM_JOB_ID: $SLURM_JOB_ID"
echo "SLURM_JOB_NODELIST: $SLURM_JOB_NODELIST"
echo "SLURM_NTASKS: $SLURM_NTASKS"
echo "SLURM_TASK_PID: $SLURM_TASK_PID"
echo "Hostname: $(hostname)"
echo "Task: $SLURM_TASKS_PER_NODE"
echo "Date: $(date)"
```

Job Dependencies

We can set up job dependencies with –dependency option. Useful when need a job to start only after another job has completed succesfully.

```
#!/bin/bash
#SBATCH --job-name=JobB
#SBATCH --nodes=1
#SBATCH --ntasks=32  # Using half of the CPUs
#SBATCH --mem=32G    # Using half of the memory
#SBATCH --time=00:01:50  # Running for 110 seconds
#SBATCH --output=jobB_output.txt
#SBATCH --error=jobB_error.txt

# The following line makes this job dependent on JobA
# NOTE: You will replace "JOBID_A" with the actual job ID of Job A after submitting it.

echo "Running Job B"
cat output_from_jobA.txt
sleep 50  # Sleep for 50 seconds
```

Job Arrays

Submit and manage collection of similar jobs.

- **SLURM_ARRAY_JOB_ID** – job id for array
- **SLURM_ARRAY_TASK_ID –** id for the task currently running from the array
- **SLURM_PROCID –** id for the task within the job
- **SLURM_ARRAY_TASK_COUNT –** count of total tasks in the array
- ...

Job Arrays

Submit and manage collection of similar jobs.

```
#!/bin/bash
#SBATCH --job-name=JobArray
#SBATCH --nodes=1
#SBATCH --ntasks=1  # 1 task for each job in the array
#SBATCH --mem=10G    # Allocating 4GB of memory for each task
#SBATCH --time=00:00:10  # Running for 10 seconds
#SBATCH --output=job_array_%A_%a_output.txt  # %A is array job ID; %a is array index
#SBATCH --error=job_array_%A_%a_error.txt
#SBATCH --array=1-10  # This creates 5 jobs with array indices from 1 to 10

echo "Running job array task with ID $SLURM_ARRAY_TASK_ID"

# Simulating CPU-intensive operation based on the task ID
end=$((SECONDS+5))
while [ $SECONDS -lt $end ]; do
    echo "scale=5000; 4*a(1) * $SLURM_ARRAY_TASK_ID" | bc -l > /dev/null
done

echo "Finished job array task with ID $SLURM_ARRAY_TASK_ID"
```

salloc - interactive resource allocation

Example: salloc –nodes=1 –ntasks=4 –mem=8G

squeue - show a list of all jobs
scancel – utility for cancelling jobs

```
malex26@buran:~/slurm_lection/array_job$ squeue
             JOBID PARTITION     NAME     USER ST       TIME  NODES NODELIST(REASON)
         72_[7-10]     debug JobArray  malex26 PD       0:00      1 (Resources)
              72_1     debug JobArray  malex26  R       0:01      1 buran
              72_2     debug JobArray  malex26  R       0:01      1 buran
              72_3     debug JobArray  malex26  R       0:01      1 buran
              72_4     debug JobArray  malex26  R       0:01      1 buran
              72_5     debug JobArray  malex26  R       0:01      1 buran
              72_6     debug JobArray  malex26  R       0:01      1 buran
malex26@buran:~/slurm_lection/array_job$
```

Checkpointing, Requeuing and Preemption

Preemptable job: use –qos=standby flag

Reque if preempted: --requeue

Checkpoint logic can be implemented in your application, or you can search for checkpointing slurm plugins

Enroot

- A simple, yet powerful tool to turn traditional container/OS images into unprivileged sandboxes.
- Enroot can be thought of as an enhanced unprivileged chroot(1). It uses the same underlying technologies as containers but removes much of the isolation they inherently provide while preserving filesystem separation.
- This approach is generally preferred in high-performance environments or virtualized environments where portability and reproducibility is important, but extra isolation is not warranted.
- Built-it NVIDIA-GPU support
- Standalone (no daemon)

Enroot

Easy to create enroot image (.sqsh file) from docker image:

- enroot import docker://pytorch/pytorch:latest

Also easy to start container:

- enroot create pytorch+pytorch+latest.sqsh

Integrating with slurm tasks:

tools > $ segmenting.sh

```
1  #!/bin/bash
2  #SBATCH --job-name=segmenting
3  #SBATCH --output=/external/nfs/malex26/tmp/slurm_logs/%j.log
4  #SBATCH --nodes=1
5  #SBATCH --ntasks-per-node=1
6  #SBATCH --gpus-per-task=0
7  #SBATCH --cpus-per-task=128
8
9  srun --container-image="/external/nfs/malex26/docker_images/py39_torch2_cuda117.sqsh" \
10   --container-mounts="/external/nfs:/external/nfs" \
11   --container-workdir="/external/nfs/malex26/tmp/slurm_workdir" \
12   python /external/nfs/malex26/tools/segmenting.py
13
```

Recap

- Distributed training with torch
- Usefull techniques
- Slurm

Enroot

Easy to create enroot image (.sqsh file) from docker image:

- enroot import docker://pytorch/pytorch:latest

Also easy to start container:

- enroot create pytorch+pytorch+latest.sqsh

Integrating with slurm tasks: