Fault Tolerance

Regardless of whether you own the ML training hardware or renting it by the hour, in this ever speeding up domain of ML, finishing the training in a timely matter is important. As such if while you were asleep one of the GPUs failed or the checkpoint storage run out of space which led to your training crashing, you'd have discovered upon waking that many training hours were lost.

Due the prohibitively high cost of ML hardware, it'd be very difficult to provide redundancy fail-over solutions as it's done in Web-services. Nevertheless making your training fault-tolerant is achievable with just a few simple recipes.

As most serious training jobs are performed in a SLURM environment, it'll be mentioned a lot, but most of this chapter's insights are applicable to any other training environments.

Always plan to have more nodes than needed

The reality of the GPU devices is that they tend to fail. Sometimes they just overheat and shut down, but can recover, at other times they just break and require a replacement.

The situation tends to ameliorate as you use the same nodes for some weeks/months as the bad apples get gradually replaced, but if you are lucky to get a new shipment of GPUs and especially the early GPUs when the technology has just come out, expect a sizeable proportion of those to fail.

Therefore, if you need 64 nodes to do your training, make sure that you have a few spare nodes and study how quickly you can replace failing nodes should the spares that you have not be enough.

It's hard to predict what the exact redundancy percentage should be, but 5-10% shouldn't be unreasonable. The more you're in a crunch to complete the training on time, the higher the safety margin should be.

Once you have the spare nodes available, validate that your SLURM environment will automatically remove any problematic nodes from the pool of available nodes so that it can automatically replace the bad nodes with the good ones.

If you use a non-SLURM scheduler validate that it too can do unmanned bad node replacements.

You also need at least one additional node for running various preventative watchdogs (discussed later in this chapter), possibly offloading the checkpoints and doing cleanup jobs.

Queue up multiple training jobs

The next crucial step is to ensure that if the training crashed, there is a new job lined up to take place of the previous one.

Therefore, when a training is started, instead of using:

```
sbatch train.slurm
```

You'd want to replace that with:

```
sbatch --array=1-10%1 train.slurm
```

This tells SLURM to book a job array of 10 jobs, and if one of the job completes normally or it crashes, it'll immediately schedule the next one.

footnote: `%1` in `--array=1-10%1` tells SLURM to launch the job array serially - one job at a time.

If you have already started a training without this provision, it's easy to fix without aborting the current job by using the `--dependency` argument:

```
sbatch --array=1-10%1 --dependency=CURRENTLY_RUNNING_JOB_ID train.slurm
```

So if your launched job looked like this:

```
$ squeue -u `whoami` -o "%.10i %9P %20j %.8T %.10M %.8l %.6D %.20S %R"
     JOBID PARTITION NAME             STATE       TIME   TIME_LIM    NODES  START_TIME     N
       87    prod    my-training-10b  RUNNING 2-15:52:19 1-16:00:00   64    2023-10-07T01:26
```

You will not that the current's `JOBID=87` and now you can use it in:

```
sbatch --array=1-10%1 --dependency=87 train.slurm
```

and then the new status will appear as:

```
$ squeue -u `whoami` -o "%.10i %9P %20j %.8T %.10M %.8l %.6D %.20S %R"
     JOBID PARTITION NAME             STATE       TIME   TIME_LIM    NODES  START_TIME     N
       87    prod    my-training-10b  RUNNING 2-15:52:19 1-16:00:00   64    2023-10-07T01:26
 88_[10%1]   prod    my-training-10b  PENDING       0:00 1-16:00:00   64
```

So you can see that an array of 10 jobs (`88_[10%1]`) was appended to be started immediately after the current job (`87`) completes or fails.

Granted that if the condition that lead to the crash is still there the subsequent job will fail as well. For example, if the storage device is full, no amount of restarts will allow the training to proceed. And we will discuss shortly how to avoid this situation.

But since the main reason for training crashes is failing GPUs, ensuring that faulty nodes are automatically removed and the new job starts with a new set of nodes makes for a smooth recovery from the crash.

In the SLURM lingo, the removed nodes are given a new status called `drained`. Here is an example of a hypothetical SLURM cluster:

```
$ sinfo
PARTITION AVAIL  TIMELIMIT  NODES  STATE NODELIST
prod*        up   infinite      4  drain node-[0-3]
prod*        up   infinite     47  alloc node-[4-51]
prod*        up   infinite     23   idle node-[52-73]
```

Here we have 47 nodes being used (`alloc`), 23 available (`idle`) and 4 unavailable (`drained`).

The sysadmin is expected to periodically check the drained nodes, fix or replace them and then make them again available to be used by changing their state to `idle`.

The other approach is to daisy-chain jobs via `--dependency` as explained here. Both of these approaches could also be combined.

How do you know when the job array or a daisy chain should not resume - well, normally the training loop will exit immediately if it knows the job is done. But you could also add features like kill switch which are even easier to use to prevent a job array from running.

Frequent checkpoint saving

Whenever the training job fails, many hours of training can be lost. This problem is mitigated by a frequent checkpoint saving. When the training is resumed it'll continue from the last checkpoint saved. If the failure occurred 12 hours after the last checkpoint has been saved, 12 hours of training is lost and needs to be re-done. This can be very expensive if the training uses hundreds of GPUs.

In theory one could save a checkpoint every 10 minutes and only ever lose 10 minutes of training time, but this too would dramatically delay the reaching of the finish line because large models can't be saved quickly and if the saving time starts to create a bottleneck for the training this approach becomes counterproductive.

Depending on your checkpointing methodology and the speed of your IO storage partition the saving of a large model can take from dozens of seconds to several

minutes. Therefore, the optimal approach to saving frequency lies somewhere in the middle.

The math is quite simple - measure the amount of time it takes to save the checkpoint, multiply it by how many times you'd want to save it and see how much of an additional delay the checkpoint saving will contribute to the total training time.

Use case: While training BLOOM-176B we had an incredibly fast GPFS over NVME filesystem and it took only 40 seconds to save a 2.3TB checkpoint written concurrently on 384 processes. We saved a checkpoint approximately every 3 hours. As we trained for about 3 months, that means that we saved about 720 checkpoints (`90 days * 24h / 3h`) - that is an additional 8 hours was spent just saving the checkpoints (`720 times * 40 secs / 3600 secs`) - or ~0.37% of the total training time (`8h / (90 days * 24 hours)`. Now say if the IO were to be 5 times slower, which is not uncommon on the cloud unless one pays for premium IO, that would have become 2% of the training time, which would be quite significant.

footnote: If you don't have a large local storage and you have to offload the checkpoints to the cloud, make sure that the 2 most frequent checkpoints remain local to allow for a quick resume. The reason for 2 and not 1, is that it's possible that the very last checkpoint got corrupted or didn't finish saving if a crash occurred during its saving.

Kill switch

In many SLURM environments users have no `sudo` access and when one user started a training and went to sleep, and then a problem has been discovered, the other users can't easily stop the training and restart it again.

This was the situation during BLOOM-176B training and we implemented a kill-switch to handle that. The mechanism is very simple. The training loop polls for a specific file to appear before starting a new iteration and if the file is there the program saves the checkpoint and exits, allowing users other than the one who started the previous training to change things and restart it again. An additional poll was added at the very beginning of `main` so that if there was a long job array queued by the user who is asleep they could be "burned through" quickly by getting each job exit quickly on start.

This is also discussed here.

This facility helps to minimize the amount of wasted training time.

Save switch

While mentioning the kill switch, it might be good to quickly mention its cousin, a save switch. Similarly to the kill switch the save switch is a variation of the former where instead of stopping the training, if the training loop discovers that a save-switch file appears - it will save a checkpoint, but will continue training. It'll also automatically remove the save-switch from the file-system, so that it won't accidentally start saving a checkpoint after every iteration.

This feature can be very useful for those who watch the training charts. If one sees an interesting or critical situation in the training loss or some other training metric one can quickly ask the training program to save the checkpoint of interest and be able to later reproduce the current situation at will.

The main use of this feature is around observing training loss spikes and divergences.

(note-to-self: better belongs to instabilities chapter)

Prevention

The easiest way to avoid losing training time is to prevent certain types of problems from happening. While one can't prevent a GPU from failing, other than ensuring that adequate cooling is provided, one can certainly ensure that there is enough of disk space remaining for the next few days of training. This is typically done by running scheduled watchdogs that monitor various resources and alert the operator of possible problems long before they occur.

Scheduled Watchdogs

Before we discuss the various watchdogs it's critical that you have a mechanism that allows you to run scheduled jobs. In the Unix world this is implemented by the `crontab` facility.

Here is an example of how `~/bin/watch-fs.sh` can be launched every hour:

```
0 * * * * ~/bin/watch-fs.sh
```

The link above explains how to configure a crontab job to run at various other frequencies.

To setup a crontab, execute `crontab -e` and check which jobs are scheduled `crontab -l`.

The reason I don't go into many details is because many SLURM environments don't provide access to the `crontab` facility. And therefore one needs to use other approaches to scheduling jobs.

The section on Crontab Emulation discusses how to implement crontab-like SLURM emulation and also Self-perpetuating SLURM jobs.

Notification facility

Then you need to have one or more notification facilities.

The simplest one is to use email to send alerts. To make this one work you need to ensure that you have a way to send an email from the SLURM job. If it isn't already available you can request this feature from your sysadmin or alternatively you might be able to use an external SMTP server provider.

In addition to email you could probably also setup other notifications, such as SMS alerting and/or if you use Slack to send slack-notifications to a channel of your choice.

Once you understand how to schedule watchdogs and you have a notification facility working let's next discuss the critical watchdogs.

Is-job-running watchdog

The most obvious watchdog is one which checks that there is a training SLURM job running or more are scheduled to run.

Here is an example slurm-status.py that was used during BLOOM-176B training. This watchdog was sending an email if a job was detected to be neither running nor scheduled and it was also piping its check results into the main training's log file. As we used Crontab Emulation, we simply needed to drop slurm-status.slurm into the `cron/cron.hourly/` folder and the previously launched SLURM crontab emulating scheduler would launch this check approximately once an hour.

The key part of the SLURM job is:

```
tools/slurm-status.py --job-name $WATCH_SLURM_NAME 2>&1 | tee -a $MAIN_LOG_FILE
```

which tells the script which job name to watch for, and you can also see that it logs into a log file.

For example, if you launched the script with:

```
tools/slurm-status.py --job-name my-training-10b
```

and the current status report shows:

```
$ squeue -u `whoami` -o "%.10i %9P %20j %.8T %.10M %.8l %.6D %.20S %R"
  JOBID    PARTITION NAME             STATE       TIME   TIME_LIM    NODES  START_TIME     N
    87     prod      my-training-10b  RUNNING 2-15:52:19 1-16:00:00  64    2023-10-07T01:26:
```

then all is good. But if `my-training-10b` job doesn't show the alert will be sent.

You can now adapt these scripts to your needs with minimal changes of editing the path and email addresses. And if it wasn't you who launched the job then

replace `whoami` with the name of the user who launched it. `whoami` only works if it was you who launched it.

Is-job-hanging watchdog

If the application is doing `torch.distributed` or alike and a hanging occurs during one of the collectives, it'll eventually timeout and throw an exception, which would restart the training and one could send an alert that the job got restarted.

However, if the hanging happens during another syscall which may have no timeout, e.g. reading from the disk, the application could easily hang there for hours and nobody will be the wiser.

Most applications do periodic logging, e.g., most training log the stats of the last N steps every few minutes. Then one could check if the log file has been updated during the expected time-frame - and if it didn't - send an alert. You could write your own, or use io-watchdog for that.

Low disk space alerts

The next biggest issue is running out of disk space. If your checkpoints are large and are saved frequently and aren't offloaded elsewhere it's easy to quickly run out of disk space. Moreover, typically multiple team members share the same cluster and it could be that your colleagues could quickly consume a lot of disk space. Ideally, you'd have a storage partition that is dedicated to your training only, but often this is difficult to accomplish. In either case you need to know when disk space is low and space making action is to be performed.

Now what should be the threshold at which the alerts are triggered. They need to be made not too soon as users will start ignoring these alerts if you start sending those at say, 50% usage. But also the percentage isn't always applicable, because if you have a huge disk space shared with others, 5% of that disk space could translate to many TBs of free disk space. But on a small partition even 25% might be just a few TBs. Therefore really you should know how often you write your checkpoints and how many TBs of disk space you need daily and how much disk space is available.

Use case: During BLOOM training we wrote a 2.3TB checkpoint every 3 hours, therefore we were consuming 2.6TB a day!

Moreover, often there will be multiple partitions - faster IO partitions dedicated to checkpoint writing, and slower partitions dedicated to code and libraries, and possibly various other partitions that could be in use and all of those need to be monitored if their availability is required for the training not crashing.

Here is another caveat - when it comes to distributed file systems not all filesystems can reliably give you a 100% of disk space you acquired. In fact with some of those types you can only reliably use at most ~80% of the allocated storage

space. The problem is that these systems use physical discs that they re-balance at the scheduled periods or triggered events, and thus any of these individual discs can reach 100% of their capacity and lead to a failed write, which would crash a training process, even though `df` would report only 80% space usage on the partition. We didn't have this problem while training BLOOM-176B, but we had it when we trained IDEFICS-80B - 80% there was the new 100%. How do you know if you have this issue - well, usually you discover it while you prepare for the training.

And this is not all. There is another issue of inodes availability and some storage partitions don't have very large inode quotas. Python packages are notorious for having hundreds to thousands of small files, which combined take very little total space, but which add up to tens of thousands of files in one's virtual environment and suddenly while one has TBs of free disk space available, but runs out of free inodes and discovering their training crashing.

Finally, many distributed partitions don't show you the disk usage stats in real time and could take up to 30min to update.

footnote: Use `df -ih` to see the inodes quota and the current usage.

footnote: Some filesystems use internal compression and so the reported disk usage can be less than reality if copied elsewhere, which can be confusing.

So here is fs-watchdog.py that was used during BLOOM-176B training. This watchdog was sending an email if any of the storage requirements thresholds hasn't been met and here is the corresponding fs-watchdog.slurm that was driving it.

If you study the watchdog code you can see that for each partition we were monitoring both the disk usage and inodes. We used special quota tools provided by the HPC to get instant stats for some partitions, but these tools didn't work for all partitions and there we had to fallback to using `df` and even a much slower `du`. As such it should be easy to adapt to your usecase.

Dealing with slow memory leaks

Some programs develop tiny memory leaks which can be very difficult to debug. Do not confuse those with the usage of MMAP where the program uses the CPU memory to read quickly read data from and where the memory usage could appear to grow over time, but this is not real as this memory gets freed when needed. You can read A Deep Investigation into MMAP Not Leaking Memory to understand why.

Of course, ideally one would analyze their software and fix the leak, but at times the leak could be coming from a 3rd party package or can be very difficult to diagnose and there isn't often the time to do that.

When it comes to GPU memory, there is the possible issue of memory fragmentation, where over time more and more tiny unused memory segments add up

and make the GPU appear to have a good amount of free memory, but when the program tries to allocate a large tensor from this memory it fails with the OOM error like:

```
RuntimeError: CUDA out of memory. Tried to allocate 304.00 MiB (GPU 0; 8.00 GiB total capaci
142.76 MiB already allocated; 6.32 GiB free; 158.00 MiB reserved in total by PyTorch)
```

In this example if there are 6.32GB free, how comes that 304MB couldn't be allocated.

One of the approaches my team developed during IDEFICS-80B training where there was some tiny CPU memory leak that would often take multiple days to lead to running out of CPU memory was to install a watchdog inside the training loop that would check the memory usage and if a threshold was reached it'd voluntarily exit the training loop. The next training job would then resume with all the CPU memory reclaimed.

footnote: The reality of machine learning trainings is that not all problems can be fixed with limited resources and often times a solid workaround provides for a quicker finish line, as compared to "stopping the presses" and potentially delaying the training for weeks, while trying to figure out where the problem is. For example we trained BLOOM-176B with `CUDA_LAUNCH_BLOCKING=1` because the training would hang without it and after multiple failed attempts to diagnose that we couldn't afford any more waiting and had to proceed as is. Luckily this environment variable that normally is used for debug purposes and which in theory should make some CUDA operations slower didn't actually make any difference to our throughput. But we have never figured out what the problem was and today it doesn't matter at all that we haven't, as we moved on with other projects which aren't impacted by that issue.

The idea is similar to the kill and save switches discussed earlier, but here instead of polling for a specific file appearance we simply watch how much resident memory is used. For example here is how you'd auto-exit if the OS shows only 5% of the virtual cpu memory remain:

```
import psutil
for batch in iterator:
    total_used_percent = psutil.virtual_memory().percent
    if total_used_percent > 0.95:
        print(f"Exiting early since the cpu memory is almost full: ({total_used_percent}%)"
        save_checkpoint()
        sys.exit()

    train_step(batch)
```

Similar heuristics could be used for setting a threshold for GPU memory usage, except one needs to be aware of cuda tensor caching and python garbage collection scheduling, so to get the actual memory usage you'd need to do first run the garbage collector then empty the cuda cache and only then you will get real

memory usage stats and then gracefully exit the training if the GPU is too close to being full.

```
import gc
import torch

for batch in iterator:
    gc.collect()
    torch.cuda.empty_cache()

    # get mem usage in GBs and exit if less than 2GB of free GPU memory remain
    free, total = map(lambda x: x/2**30, torch.cuda.mem_get_info());
    if free < 2:
        print(f"Exiting early since the GPU memory is almost full: ({free}GB remain)")
        save_checkpoint()
        sys.exit()

    train_step(batch)
```

footnote: don't do this unless you really have to, since caching makes things faster. Ideally figure out the fragmentation issue instead. For example, look up `max_split_size_mb` in the doc for `PYTORCH_CUDA_ALLOC_CONF` as it controls how memory is allocated. Some frameworks like Deepspeed solve this by pre-allocating tensors at start time and then reuse them again and again preventing the issue of fragmentation altogether.

footnote: this simplified example would work for a single node. For multiple nodes you'd need to gather the stats from all participating nodes and find the one that has the least amount of memory left and act upon that.

Dealing with forced job preemption

Earlier you have seen how the training can be gracefully stopped with a kill switch solution and it's useful when you need to stop or pause the training on demand.

On HPC clusters SLURM jobs have a maximum runtime. A typical one is 20 hours. This is because on HPCs resources are shared between multiple users/groups and so each is given a time slice to do compute and then the job is forcefully stopped, so that other jobs could use the shared resources.

footnote: this also means that you can't plan how long the training will take unless your jobs run with the highest priority on the cluster. If your priority is not the highest it's not uncommon to have to wait for hours and sometimes days before your job resumes.

One could, of course, let the job killed and hope that not many cycles were spent since the last checkpoint was saved and then let the job resume from this

checkpoint, but that's quite wasteful and best avoided.

The efficient solution is to gracefully exit before the hard tile limit is hit and the job is killed by SLURM.

First, you need to figure out how much time your program needs to gracefully finish. This typically requires 2 durations:

1. how long does it take for a single iteration to finish if you have just started a new iteration
2. how long does it take to save the checkpoint

If, for example, the iteration takes 2 minutes at most and the checkpoint saving another 2 minutes, then you need at least 4 minutes of that grace time. To be safe I'd at least double it. There is no harm at exiting a bit earlier, as no resources are wasted.

So, for example, let's say your HPC allows 100 hour jobs, and then your slurm script will say:

```
#SBATCH --time=100:00:00
```

Approach A. Tell the program at launch time when it should start the exiting process:

`srun ... torchrun ... --exit-duration-in-mins 5990`

100h is 6000 minutes and so here we give the program 10 mins to gracefully exit.

And when you start the program you create a timer and then before every new iteration starts you check if the time limit is reached. If it is you save the checkpoint and exit.

case study: you can see how this was set in the BLOOM training job and then acted upon here:

```
        # Exiting based on duration
        if args.exit_duration_in_mins:
            train_time = (time.time() - _TRAIN_START_TIME) / 60.0
            done_cuda = torch.cuda.IntTensor(
                [train_time > args.exit_duration_in_mins])
            torch.distributed.all_reduce(
                done_cuda, op=torch.distributed.ReduceOp.MAX)
            done = done_cuda.item()
            if done:
                if not saved_checkpoint:
                    save_checkpoint_and_time(iteration, model, optimizer,
                                             lr_scheduler)
                print_datetime('exiting program after {} minutes'.format(train_time))
                sys.exit()
```

As you can see since the training is distributed we have to synchronize the exiting event across all ranks

You could also automate the derivation, by retrieving the `EndTime` for the running job:

```
$ scontrol show -d job $SLURM_JOB_ID | grep Time
   RunTime=00:00:42 TimeLimit=00:11:00 TimeMin=N/A
   SubmitTime=2023-10-26T15:18:01 EligibleTime=2023-10-26T15:18:01
   AccrueTime=2023-10-26T15:18:01
   StartTime=2023-10-26T15:18:01 EndTime=2023-10-26T15:18:43 Deadline=N/A
```

and then comparing with the current time in the program and instead setting the graceful exit period. There are other timestamps and durations that can be retrieved as it can be seen from the output.

Approach B.1. Sending a custom signal X minutes before the end

In your sbatch script you could set:

```
#SBATCH --signal=USR1@600
```

and then SLURM will send a `SIGUSR1` signal to your program 10min before job's end time.

footnote: normally SLURM schedulers send a `SIGCONT`+`SIGTERM` signal about 30-60 seconds before the job's time is up, and just as the time is up it will send a `SIGCONT`+`SIGTERM`+`SIGKILL` signal if the job is still running. `SIGTERM` can be caught and acted upon but 30 seconds is not enough time to gracefully exit a large model training program.

Let's demonstrate how the signal sending and trapping works. In terminal A, run:

```
python -c "
import time, os, signal

def sighandler(signum, frame):
    print('Signal handler called with signal', signum)
    exit(0)

signal.signal(signal.SIGUSR1, sighandler)
print(os.getpid())
time.sleep(1000)
"
```

it will print the pid of the process, e.g., `4034989` and will go to sleep (emulating real work). In terminal B now send `SIGUSR1` signal to the python program in terminal A with:

```
kill -s USR1 4034989
```

The program will trap this signal, call the `sighandler` which will now print that it was called and exit.

```
Signal handler called with signal 10
```

`10` is the numerical value of `SIGUSR1`.

So here is the same thing with the SLURM setup:

```
$ cat sigusr1.slurm
#SBATCH --job-name=sigusr1
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=1
#SBATCH --time=0:03:00
#SBATCH --partition=mypartition
#SBATCH --output=%x-%j.out
#SBATCH --signal=USR1@170

srun python -c "
import time, os, signal

def sighandler(signum, frame):
    print('Signal handler called with signal', signum)
    exit(0)

signal.signal(signal.SIGUSR1, sighandler)
print(os.getpid())
time.sleep(1000)
"
```

In the SLURM script we told SLURM to send the program a signal 170 seconds before its end and the job itself was set to run for 180 secs (3 mins).

When this job has been scheduled:

```
sbatch sigusr1.slurm
```

10 seconds (`180-170`) after the job started, it will exit with the log:

```
58307
Signal handler called with signal 10
```

which means the job had a pid `58307` and it caught `SIGUSR1` (`10`) and it exited.

Now that you understand how this machinery works, instead of immediate `exit(0)` you can set exit-asap flag, finish the currently run iteration, check that the flag is up, save the checkpoint and exit. This is very similar to the code shown in Approach A above.