Debugging PyTorch programs

Getting nodes to talk to each other

Once you need to use more than one node to scale your training, e.g., if you want to use DDP to train faster, you have to get the nodes to talk to each other, so that communication collectives could send data to each other. This is typically done via a comms library like NCCL. And in our DDP example, at the end of training step all GPUs have to perform an `all_reduce` call to synchronize the gradients across all ranks.

In this section we will discuss a very simple case of just 2 nodes (with 8 GPUs each) talking to each other and which can then be easily extended to as many nodes as needed. Let's say that these nodes have the IP addresses 10.0.0.1 and 10.0.0.2.

Once we have the IP addresses we then need to choose a port for communications.

In Unix there are 64k ports. The first 1k are reserved for common services so that any computer on the Internet could connect to any other computer knowing ahead of time which port to connect to. For example, port 22 is reserved for SSH. So that whenever you do `ssh example.com` in fact the program open a connection to `example.com:22`.

As there are thousands of services out there, the reserved 1k ports is not enough, and so various services could use pretty much any port. But fear not, when you get your Linux box on the cloud or an HPC, you're unlikely to have many preinstalled services that could use a high number port, so most ports should be available.

Therefore let's choose port 6000.

Now we have: `10.0.0.1:6000` and `10.0.0.2:6000` that we want to be able to communicate with each other.

The first thing to do is to open port `6000` for incoming and outgoing connections on both nodes. It might be open already or you might have to read up the instructions of your particular setup on how to open a given port.

Here are multiple ways that you could use to test whether port 6000 is already open.

```
telnet localhost:6000
nmap -p 6000 localhost
nc -zv localhost 6000
curl -v telnet://localhost:6000
```

Most of these should be available via `apt install` or whatever your package manager uses.

Let's use `nmap` in this example. If I run:

```
$ nmap -p 22 localhost
[...]
PORT   STATE SERVICE
22/tcp open  ssh
```

We can see the port is open and it tells us which protocol and service is allocated as a bonus.

Now let's run:

```
$ nmap -p 6000 localhost
[...]

PORT     STATE  SERVICE
6000/tcp closed X11
```

Here you can see port 6000 is closed.

Now that you understand how to test, you can proceed to test the `10.0.0.1:6000` and `10.0.0.2:6000`.

First ssh to the first node in terminal A and test if port 6000 is opened on the second node:

```
ssh 10.0.0.1
nmap -p 6000 10.0.0.2
```

if all is good, then in terminal B ssh to the second node and do the same check in reverse:

```
ssh 10.0.0.2
nmap -p 6000 10.0.0.1
```

If both ports are open you can now use this port. If either or both are closed you have to open these ports. Since most clouds use a proprietary solution, simply search the Internet for "open port" and the name of your cloud provider.

The next important thing to understand is that compute nodes will typically have multiple network interface cards (NICs). You discover those interfaces by running:

```
$ sudo ifconfig
```

One interface is typically used by users to connecting to nodes via ssh or for various other non-compute related services - e.g., sending an email or download some data. Often this interface is called `eth0`, with `eth` standing for Ethernet, but it can be called by other names.

Then there is the inter-node interface which can be Infiniband, EFA, OPA, HPE Slingshot, etc. (more information). There could be one or dozens of those interfaces.

Here are some examples of `ifconfig`'s output:

```
$ sudo ifconfig
enp5s0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 10.0.0.23  netmask 255.255.255.0  broadcast 10.0.0.255
        [...]
```

I removed most of the output showing only some of the info. Here the key information is the IP address that is listed after `inet`. In the example above it's `10.0.0.23`. This is the IP address of interface `enp5s0`.

If there is another node, it'll probably be `10.0.0.24` or `10.0.0.21` or something of sorts - the last segment will be the one with a different number.

Let's look at another example:

```
$ sudo ifconfig
ib0     Link encap:UNSPEC  HWaddr 00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00
        inet addr:172.0.0.50  Bcast: 172.0.0.255  Mask:255.255.255.0
        [...]
```

Here `ib` typically tells us it's an InfiniBand card, but really it can be any other vendor. I have seen OmniPath using `ib` for example. Again `inet` tells us the IP of this interface is `172.0.0.50`.

If you lost me, we want the IP addresses so that we could test if ip:port is open on each node in question.

Finally, going back to our pair of `10.0.0.1:6000` and `10.0.0.2:6000` let's do an `all_reduce` test using 2 terminals, where we choose `10.0.0.1` as the master host which will coordinate other nodes. For testing we will use this helper debug program torch-distributed-gpu-test.py.

In terminal A:

```
$ ssh 10.0.0.1
$ python -m torch.distributed.run --role $(hostname -s): --tee 3 --nnodes 2 --nproc_per_node
 --master_addr 10.0.0.1 --master_port 6000 torch-distributed-gpu-test.py
```

In terminal B:

```
$ ssh 10.0.0.2
$ python -m torch.distributed.run --role $(hostname -s): --tee 3 --nnodes 2 --nproc_per_node
 --master_addr 10.0.0.1 --master_port 6000 torch-distributed-gpu-test.py
```

Note that I'm using the same `--master_addr 10.0.0.1 --master_port 6000` in both cases because we checked port 6000 is open and we use `10.0.0.1` as the coordinating host.

This approach of running things manually from each node is painful and so there are tools that automatically launch the same command on multiple nodes

pdsh

`pdsh` is one such solution - which is like `ssh` but will automatically run the same command on multiple nodes:

```
PDSH_RCMD_TYPE=ssh pdsh -w 10.0.0.1,10.0.0.2 \
"python -m torch.distributed.run --role $(hostname -s): --tee 3 --nnodes 2 --nproc_per_node
 --master_addr 10.0.0.1 --master_port 6000 torch-distributed-gpu-test.py"
```

You can see how I folded the 2 sets of commands into 1. If you have more nodes, just add more nodes as `-w` argument.

SLURM

If you use SLURM, it's almost certain that whoever set things up already have all the ports opened for you, so it should just work. But if it doesn't the information in this section should help debug things.

Here is how you'd use this with SLURM.

```
#!/bin/bash
#SBATCH --job-name=test-nodes        # name
#SBATCH --nodes=2                    # nodes
#SBATCH --ntasks-per-node=1          # crucial - only 1 task per dist per node!
#SBATCH --cpus-per-task=10           # number of cores per tasks
#SBATCH --gres=gpu:8                 # number of gpus
#SBATCH --time 0:05:00               # maximum execution time (HH:MM:SS)
#SBATCH --output=%x-%j.out           # output file name
#
export GPUS_PER_NODE=8
export MASTER_ADDR=$(scontrol show hostnames $SLURM_JOB_NODELIST | head -n 1)
export MASTER_PORT=6000
#
srun --jobid $SLURM_JOBID bash -c 'python -m torch.distributed.run \
--nproc_per_node $GPUS_PER_NODE --nnodes $SLURM_NNODES --node_rank $SLURM_PROCID \
--master_addr $MASTER_ADDR --master_port $MASTER_PORT \
torch-distributed-gpu-test.py'
```

If you have more than 2 nodes you just need to change the number of nodes and the above script will automatically work for any number of them.

MPI:

Another popular way is to use Message Passing Interface (MPI). There are a few open source implementations of it available.

To use this tool you first create a `hostfile` that contains your target nodes and the number of processes that should be run on each host. In the example of this section, with 2 nodes and 8 gpus each it'd be:

```
$ cat hostfile
10.0.0.1:8
10.0.0.2:8
```

and to run, it's just:

```
$ mpirun --hostfile  -np 16 -map-by ppr:8:node python my-program.py
```

Note that I used `my-program.py` here because torch-distributed-gpu-test.py was written to work with `torch.distributed.run` (also known as `torchrun`). With `mpirun` you will have to check your specific implementation to see which environment variable it uses to pass the rank of the program and replace `LOCAL_RANK` with it, the rest should be mostly the same.

Nuances: - You might have to explicitly tell it which interface to use by adding `--mca btl_tcp_if_include 10.0.0.0/24` to match our example. If you have many network interfaces it might use one that isn't open or just the wrong interface. - You can also do the reverse and exclude some interfaces. e.g. say you have `docker0` and `lo` interfaces - to exclude those add `--mca btl_tcp_if_exclude docker0,lo`.

`mpirun` has a gazillion of flags and I will recommend reading its manpage for more information. My intention was only to show you how you could use it. Also different `mpirun` implementations may use different CLI options.

Solving the Infiniband connection between multiple nodes

In one situation on Azure I got 2 nodes on a shared subnet and when I tried to run the 2 node NCCL test:

```
NCCL_DEBUG=INFO python -u -m torch.distributed.run --nproc_per_node=1 --nnodes 2 --rdzv_endp
```

I saw in the debug messages that Infiniband interfaces got detected:

```
node-2:5776:5898 [0] NCCL INFO NET/IB : Using [0]ibP111p0s0:1/IB [1]rdmaP1111p0s2:1/RoCE [RO
```

But the connection would then time out with the message:

```
node-2:5776:5902 [0] transport/net_ib.cc:1296 NCCL WARN NET/IB : Got completion from peer 10
0, vendor err 129 (Recv)
node-2:5776:5902 [0] NCCL INFO transport/net.cc:1134 -> 6
```

```
node-2:5776:5902 [0] NCCL INFO proxy.cc:679 -> 6
node-2:5776:5902 [0] NCCL INFO proxy.cc:858 -> 6 [Proxy Thread]
```

and nothing works. So here the Ethernet connectivity between 2 nodes works but not the IB interface.

There could be a variety of reason for this failing, but of the most likely one is when you're on the cloud and the 2 nodes weren't provisioned so that their IB is connected. So your Ethernet inter-node connectivity works, but it's too slow. Chances are that you need to re-provision the nodes so that they are allocated together. For example, on Azure this means you have to allocate nodes within a special availability set

Going back to our case study, once the nodes were deleted and recreated within an availability set the test worked out of the box.

The individual nodes are often not meant for inter-node communication and often the clouds have the concept of clusters, which are designed for allocating multiple nodes as a group and are already preconfigured to work together.

Prefixing logs with `node:rank`, interleaved asserts

In this section we will use `torchrun` (`torch.distributed.run`) during the demonstration and at the end of this section similar solutions for other launchers will be listed.

When you have warnings and tracebacks (or debug prints), it helps a lot to prefix each log line with its `hostname:rank` prefix, which is done by adding `--role $(hostname -s): --tee 3` to `torchrun`:

```
python -m torch.distributed.run --role $(hostname -s): --tee 3 --nnodes 1 --nproc_per_node
torch-distributed-gpu-test.py
```

Now each log line will be prefixed with `[hostname:rank]`

Note that the colon is important.

If you're in a SLURM environment the above command line becomes:

```
srun --jobid $SLURM_JOBID bash -c 'python -m torch.distributed.run \
--nproc_per_node $GPUS_PER_NODE --nnodes $SLURM_NNODES --node_rank $SLURM_PROCID \
--master_addr $MASTER_ADDR --master_port $MASTER_PORT \
--role $(hostname -s): --tee 3 \
torch-distributed-gpu-test.py'
```

Of course adjust your environment variables to match, this was just an example.

Important! Note, that I'm using a single quoted string of commands passed to `bash -c`. This way `hostname -s` command is delayed until it's run on each of the nodes. If you'd use double quotes above, `hostname -s` will get executed on the starting node and then all nodes will get the same hostname as the prefix, which defeats the purpose of using these flags. So if you use double quotes you need to rewrite the above like so:

```
srun --jobid $SLURM_JOBID bash -c "python -m torch.distributed.run \
--nproc_per_node $GPUS_PER_NODE --nnodes $SLURM_NNODES --node_rank \$SLURM_PROCID \
--master_addr $MASTER_ADDR --master_port $MASTER_PORT \
--role \$(hostname -s): --tee 3 \
torch-distributed-gpu-test.py"
```

`$SLURM_PROCID` is escaped too as it needs to be specific to each node and it's unknown during the launch of the slurm job on the main node. So there are 2 `\$` escapes in this version of the command.

This prefixing functionality is also super-helpful when one gets the distributed program fail and which often results in interleaved tracebacks that are very difficult to interpret. So by `grep`ing for one `node:rank` string of choice, it's now possible to reconstruct the real error message.

For example, if you get a traceback that looks like:

```
  File "/path/to/training/dataset.py", line 785, in __init__
  File "/path/to/training/dataset.py", line 785, in __init__
    if self.dataset_proba.sum() != 1:
AttributeError: 'list' object has no attribute 'sum'
    if self.dataset_proba.sum() != 1:
  File "/path/to/training/dataset.py", line 785, in __init__
  File "/path/to/training/dataset.py", line 785, in __init__
    if self.dataset_proba.sum() != 1:
    if self.dataset_proba.sum() != 1:
AttributeError: 'list' object has no attribute 'sum'
AttributeError: 'list' object has no attribute 'sum'
AttributeError: 'list' object has no attribute 'sum'
```

and when it's dozens of frames over 8 nodes it can't be made sense of, but the above `-tee` + `--role` addition will generate:

```
[host1:0]  File "/path/to/training/dataset.py", line 785, in __init__
[host1:1]  File "/path/to/training/dataset.py", line 785, in __init__
[host1:0]    if self.dataset_proba.sum() != 1:
[host1:0]AttributeError: 'list' object has no attribute 'sum'
[host1:1]    if self.dataset_proba.sum() != 1:
[host1:2]  File "/path/to/training/dataset.py", line 785, in __init__
[host1:3]  File "/path/to/training/dataset.py", line 785, in __init__
[host1:3]    if self.dataset_proba.sum() != 1:
[host1:2]    if self.dataset_proba.sum() != 1:
[host1:1]AttributeError: 'list' object has no attribute 'sum'
[host1:2]AttributeError: 'list' object has no attribute 'sum'
[host1:3]AttributeError: 'list' object has no attribute 'sum'
```

and you can `grep` this output for just one `host:rank` prefix, which gives us:

```
$ grep "[host1:0]" log.txt
[host1:0]  File "/path/to/training/dataset.py", line 785, in __init__
[host1:0]    if self.dataset_proba.sum() != 1:
[host1:0]AttributeError: 'list' object has no attribute 'sum'
```

and voila, you can now tell what really happened. And as I mentioned earlier there can be easily a hundred to thousands of interleaved traceback lines there.

Also, if you have just one node, you can just pass `-tee 3` and there is no need to pass `--role`.

If `hostname -s` is too long, but you have each host with its own sequence number like:

```
[really-really-really-long-hostname-5:0]
[really-really-really-long-hostname-5:1]
[really-really-really-long-hostname-5:2]
```

you can of course make it shorter by replacing `hostname -s` with `hostname -s | tr -dc '0-9'`, which would lead to much shorter prefixes:

```
[5:0]
[5:1]
[5:2]
```

And, of course, if you're doing debug prints, then to solve this exact issue you can use `printflock`.

Here is how you accomplish the same feat with other launchers:

- `srun` in SLURM: add `--label`
- `openmpi`: add `--tag-output`
- `accelerate`: you can just pass the same `-tee` + `--role` flags as in `torchrun`

Dealing with Async CUDA bugs

When using CUDA, failing pytorch programs very often produce a python traceback that makes no sense or can't be acted upon. This is because due to CUDA's async nature - when a CUDA kernel is executed, the program has already moved on and when the error happened the context of the program isn't there. The async functionality is there to make things faster, so that while the GPU is churning some `matmul` the program on CPU could already start doing something else.

At other times some parts of the system will actually tell you that they couldn't generate the correct traceback, as in this error:

```
[E ProcessGroupNCCL.cpp:414] Some NCCL operations have failed or timed out. Due to the
asynchronous nature of CUDA kernels, subsequent GPU operations might run on corrupted/
```

```
incomplete data. To avoid this inconsistency, we are taking the entire process down.
```

There are a few solutions.

If the failure is instant and can be reproduced on CPU (not all programs work on CPU), simply re-rerun it after hiding your GPUs. This is how you do it:

```
CUDA_VISIBLE_DEVICES="" python my-pytorch-program.py
```

The env var `CUDA_VISIBLE_DEVICES` is used to manually limit the visibility of GPUs to the executed program. So for example if you have 8 gpus and you want to run program1.py with first 4 gpus and program2.py with the remaining 2 gpus you can do:

```
CUDA_VISIBLE_DEVICES="0,1,2,3" python my-pytorch-program1.py
CUDA_VISIBLE_DEVICES="4,5,6,7" python my-pytorch-program2.py
```

and the second program won't be the wiser that it's not using GPUs 0-3.

But in the case of debug we are hiding all GPUs, by setting `CUDA_VISIBLE_DEVICES=""`.

Now the program runs on CPU and you will get a really nice traceback and will fix the problem in no time.

But, of course, if you your program requires multiple GPUs this won't work. And so here is another solution.

Rerun your program after setting this environment variable:

```
CUDA_LAUNCH_BLOCKING=1 python my-pytorch-program.py
```

This variable tells pytorch (or any other CUDA-based program) to turn its async nature off everywhere and now all operations will be synchronous. So when the program crashes you should now get a perfect traceback and you will know exactly what ails your program.

In theory enabling this variable should make everything run really slow, but in reality it really depends on your software. We did the whole of BLOOM-176B training using `CUDA_LAUNCH_BLOCKING=1` with `Megatron-Deepspeed`](https://github.com/bigscience-workshop/Megatron-DeepSpeed) and had zero slowdown - we had to use it as pytorch was hanging without it and we had no time to figure the hanging out.

So, yes, when you switch from async to sync nature, often it can hide some subtle race conditions, so there are times that a hanging disappears as in the example I shared above. So measure your throughput with and without this flag and sometimes it might actual not only help with getting an in-context traceback but actually solve your problem altogether.

Note: NCCL==2.14.3 coming with `pytorch==1.13` hangs when `CUDA_LAUNCH_BLOCKING=1` is used. So don't use it with that version of pytorch. The issue has been fixed in `nccl>=2.17` which should be included in `pytorch==2.0`.

segfaults and getting a backtrace from a core file

It's not uncommon for a complex pytorch program to segfault and drop a core file. Especially if you're using complex extensions like NCCL.

The corefile is what the program generates when it crashes on a low-level - e.g. when using a python extension - such as a CUDA kernel or really any library that is coded directly in some variant of C or another language and made accessible in python through some binding API. The most common cause of a segfault is when such software accesses memory it has not allocated. For example, a program may try to free memory it hasn't allocated. But there could be many other reasons.

When a segfault event happens Python can't do anything, as the proverbial carpet is pulled out from under its feet, so it can't generate an exception or even write anything to the output.

In these situation one must go and analyse the libC-level calls that lead to the segfault, which is luckily saved in the core file.

If your program crashed, you will often find a file that will look something like: `core-python-3097667-6`

Before we continue make sure you have `gdb` installed:

```
sudo apt-get install gdb
```

Now make sure you know the path to the python executable that was used to run the program that crashed. If you have multiple python environment you have to activate the right environment first. If you don't `gdb` may fail to unpack the core file.

So typically I'd go:

```
conda activate my-env
gdb python core-python-3097667-6
```

- adjust `my-env` to whatever env you use, or instead of conda use whatever way you use to activate your python environment - and perhaps you're using the system-wise python and then you don't need to activate anything.
- adjust the name of the core file to the file you have gotten - it's possible that there are many - pick the latest then.

Now `gdb` will churn for a bit and will give you a prompt where you type: `bt`. We will use an actual core file here:

```
(gdb) bt
#0  0x0000147539887a9f in raise () from /lib64/libc.so.6
#1  0x000014753985ae05 in abort () from /lib64/libc.so.6
#2  0x000014751b85a09b in __gnu_cxx::__verbose_terminate_handler() [clone .cold.1] () from /
#3  0x000014751b86053c in __cxxabiv1::__terminate(void (*)()) () from /lib64/libstdc++.so.6
#4  0x000014751b860597 in std::terminate() () from /lib64/libstdc++.so.6
```

```
#5  0x000014751b86052e in std::rethrow_exception(std::__exception_ptr::exception_ptr) () fro
#6  0x000014750bb007ef in c10d::ProcessGroupNCCL::WorkNCCL::handleNCCLGuard() ()
   from .../python3.8/site-packages/torch/lib/libtorch_cuda_cpp.so
#7  0x000014750bb04c69 in c10d::ProcessGroupNCCL::workCleanupLoop() ()
   from.../python3.8/site-packages/torch/lib/libtorch_cuda_cpp.so
#8  0x000014751b88cba3 in execute_native_thread_routine () from /lib64/libstdc++.so.6
#9  0x000014753a3901cf in start_thread () from /lib64/libpthread.so.0
#10 0x0000147539872dd3 in clone () from /lib64/libc.so.6
```

and there you go. How do you make sense of it?

Well, you go from the bottom of the stack to the top. You can tell that a `clone` call was made in `libc` which then called `start_thread` in `libpthread` and then if you keep going there are a bunch of calls in the torch libraries and finally we can see that the program terminated itself, completing with `raise` from `libc` which told the Linux kernel to kill the program and create the core file.

This wasn't an easy to understand backtrace.

footnote: Yes, python calls it a *traceback* and elsewhere it's called a *backtrace* - it's confusing, but it's more or less the same thing.

Actually I had to ask pytorch devs for help and received:

- PyTorch `ProcessGroup` watchdog thread caught an asynchronous error from NCCL
- This error is an `"unhandled system error"` which in this particular case turned out to be an IB-OPA error
- The `ProcessGroup`'s `WorkCleanUp` thread rethrew the error so that the main process would crash and the user would get notified (otherwise this async error would not surface)

Trust me there are times when even if you're inexperienced the backtrace can give you enough of a hint to where you should look for troubleshooting.

But fear not - most of the time you won't need to understand the traceback. Ideally you'd just attach the core file to your filed Issue. But it can easily be 5GB large. So the developers that will be trying to help you will ask you to generate a `gdb` backtrace and now you know how to do that.

I didn't promise it'll be easy, I just showed you where to start.

Now another useful details is that many programs these days run multiple threads. And `bt` only shows the main thread of the process. But, often, it can be helpful to see where other threads in the process were when segfault has happened. For that you simply type 2 commands at the `(gdb)` prompt:

```
(gdb) thread apply all bt
(gdb) bt
```

and this time around you typically will get a massive report, one backtrace per thread.

py-spy

This is a super-useful tool for analysing hanging programs. For example, when a you have a resource deadlock or there is an issue with a network connection.

You will find an exhaustive coverage of this tool here.

strace

Similar to py-spy, `strace` is a super-useful tool which traces any running application at the low-level system calls - e.g. `libC` and alike.

For example, run:

```
strace python -c "print('strace')"
```

and you will see everything that is done at the system call level as the above program runs.

But usually it's more useful when you have a stuck program that spins all CPU cores at 100% but nothing happens and you want to see what's it doing. In this situation you simply attached to the running program like so:

```
strace --pid PID
```

where you get the PID for example from the output of `top` or `ps`. Typically I just copy-n-paste the PID of the program that consumes the most CPU - `top` usually shows it at the very top of its listing.

Same as `py-spy` you may need `sudo` perms to attached to an already running process - it all depends on your system setup. But you can always start a program with `strace` as I have shown in the original example.

Let's look at a small sub-snippet of the output of `strace python -c "print('strace')"`

```
write(1, "strace\n", 7strace
)                  = 7
```

Here we can see that a write call was executed on filedescriptor `1`, which almost always is `stdout` (`stdin` being 0, and `stderr` being 2).

If you're not sure what a filedescriptor is pointing to, normally you can tell from `strace`'s output itself. But you can also do:

```
ls -l /proc/PID/fd
```

where PID is the pid of the currently running program you're trying to investigate.

For example, when I run the above while running a pytest test with gpus, I got (partial output):

```
l-wx------ 1 stas stas 64 Mar  1 17:22 5 -> /dev/null
lr-x------ 1 stas stas 64 Mar  1 17:22 6 -> /dev/urandom
lrwx------ 1 stas stas 64 Mar  1 17:22 7 -> /dev/nvidiactl
lrwx------ 1 stas stas 64 Mar  1 17:22 8 -> /dev/nvidia0
lr-x------ 1 stas stas 64 Mar  1 17:22 9 -> /dev/nvidia-caps/nvidia-cap2
```

so you can see that a device `/dev/null` is open as FD (file descriptor) 5, `/dev/urandom` as FD 6, etc.

Now let's go look at another snippet from our `strace` run.

```
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or directory)
```

Here it tried to see if file `/etc/ld.so.preload` exists, but as we can see it doesn't - this can be useful if some shared library is missing - you can see where it's trying to load it from.

Let's try another one:

```
openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libpthread.so.0", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\2\1\1\0\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\0\0\0\0\0\0\0\0"..., 832) = 832
newfstatat(3, "", {st_mode=S_IFREG|0644, st_size=21448, ...}, AT_EMPTY_PATH) = 0
mmap(NULL, 16424, PROT_READ, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x7f8028807000
mmap(0x7f8028808000, 4096, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1
mmap(0x7f8028809000, 4096, PROT_READ, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x2000) = 0x7
mmap(0x7f802880a000, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x
close(3)
```

here we can see that it opens `/lib/x86_64-linux-gnu/libpthread.so.0` and assigns it FD 3, it then reads 832 chars from FD 3, (we can also see that the first chars are ELF - which stands for a shared library format), then memory maps it and closes that file.

In this following example, we see a python cached file is opened, its filepointer is moved to 0, and then it's read and closed.

```
openat(AT_FDCWD, "/home/stas/anaconda3/envs/py38-pt113/lib/python3.8/__pycache__/abc.cpytho
fstat(3, {st_mode=S_IFREG|0664, st_size=5329, ...}) = 0
lseek(3, 0, SEEK_CUR)                   = 0
lseek(3, 0, SEEK_CUR)                   = 0
fstat(3, {st_mode=S_IFREG|0664, st_size=5329, ...}) = 0
brk(0x23bf000)                          = 0x23bf000
read(3, "U\r\r\n\0\0\0\0\24\216\177c\211\21\0\0\343\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0"..., 5330
read(3, "", 1)                          = 0
close(3)
```

It's important to notice that file descriptors are re-used, so we have seen the same FD 3 twice, but each time it was open to a different file.

If your program is for example trying to reach to the Internet, you can also tell these calls from `strace` as the program would be reading from a socket file descriptor.

So let's run an example on a program that downloads files from the HF hub:

```
strace python -c 'import sys; from transformers import AutoConfig; AutoConfig.from_pretraine
```

here is some relevant to this discussion snippet:

```
socket(AF_INET6, SOCK_STREAM|SOCK_CLOEXEC, IPPROTO_TCP) = 3
setsockopt(3, SOL_TCP, TCP_NODELAY, [1], 4) = 0
ioctl(3, FIONBIO, [1])                  = 0
connect(3, {sa_family=AF_INET6, sin6_port=htons(443), sin6_flowinfo=htonl(0), inet_pton(AF_I
", &sin6_addr), sin6_scope_id=0}, 28) = -1 EINPROGRESS (Operation now in progress)
poll([{fd=3, events=POLLOUT|POLLERR}], 1, 10000) = 1 ([{fd=3, revents=POLLOUT}])
getsockopt(3, SOL_SOCKET, SO_ERROR, [0], [4]) = 0
[...]
write(3, "\26\3\3\0F\20\0\0BA\4\373m\244\16\354/\334\205\361j\225\356\202m*\305\332\275\251\
read(3, 0x2f05c13, 5)                   = -1 EAGAIN (Resource temporarily unavailable)
poll([{fd=3, events=POLLIN}], 1, 9903)  = 1 ([{fd=3, revents=POLLIN}])
read(3, "\24\3\3\0\1", 5)               = 5
read(3, "\1", 1)                        = 1
read(3, "\26\3\3\0(", 5)                = 5
read(3, "\0\0\0\0\0\0\0\0\344\v\273\225`\4\24m\234~\371\332%l\364\254\34\3472<\0356s\313"...
ioctl(3, FIONBIO, [1])                  = 0
poll([{fd=3, events=POLLOUT}], 1, 10000) = 1 ([{fd=3, revents=POLLOUT}])
write(3, "\27\3\3\1.\0\374$\361\217\337\377\264g\215\364\345\256\260\211$\326pkR\345\276,\32
ioctl(3, FIONBIO, [1])                  = 0
read(3, 0x2ef7283, 5)                   = -1 EAGAIN (Resource temporarily unavailable)
poll([{fd=3, events=POLLIN}], 1, 10000) = 1 ([{fd=3, revents=POLLIN}])
```

You can see where that again it uses FD 3 but this time it opens a INET6 socket instead of a file. You can see that it then connects to that socket, polls, reads and writes from it.

There are many other super useful understandings one can derive from using this tool.

BTW, if you don't want to scroll up-down, you can also save the output to a file:

```
strace -o strace.txt python -c "print('strace')"
```

Now, since you're might want to strace the program from the very beginning, for example to sort out some race condition on a distributed filesystem, you will want to tell it to follow any forked processes. This what the `-f` flag is for:

```
strace -o log.txt -f python -m torch.distributed.run --nproc_per_node=4 --nnodes=1 --tee 3 t
```

So here we launch 4 processes and will end up running `strace` on at least 5 of them - the launcher plus 4 processes (each of which may spawn further child processes).

It will conveniently prefix each line with the pid of the program so it should be easy to tell which system was made by which process.

But if you want separate logs per process, then use `-ff` instead of `-f`.

The `strace` manpage has a ton of other useful options.

Invoke pdb on a specific rank in multi-node training

Once pytorch 2.2 is released you will have a new handy debug feature:

```
import torch.distributed as dist
[...]

def mycode(...):

   dist.breakpoint(0)
```

This is the same as `ForkedPdb` (below) but will automatically break for you on the rank of your choice - rank0 in the example above. Just make sure to call `up;;n` right away when the breakpoint hits to get into your normal code.

Here is what it does underneath:

```
import sys
import pdb

class ForkedPdb(pdb.Pdb):
    """
    PDB Subclass for debugging multi-processed code
    Suggested in: https://stackoverflow.com/questions/4716533/how-to-attach-debugger-to-a-py
    """
    def interaction(self, *args, **kwargs):
        _stdin = sys.stdin
        try:
            sys.stdin = open('/dev/stdin')
            pdb.Pdb.interaction(self, *args, **kwargs)
        finally:
            sys.stdin = _stdin


def mycode():

    if dist.get_rank() == 0:
        ForkedPdb().set_trace()
    dist.barrier()
```

so you can code it yourself as well.

And you can use that `ForkedPdb` code for normal forked applications, minus the `dist` calls.

Floating point math discrepancies on different devices

It's important to understand that depending on which device the floating point math is performed on the outcomes can be different. For example doing the same floating point operation on a CPU and a GPU may lead to different outcomes, similarly when using 2 different GPU architectures, and even more so if these are 2 different types of accelerators (e.g. NVIDIA vs. AMD GPUs).

Here is an example of discrepancies I was able to get doing the same simple floating point math on an 11 Gen Intel i7 CPU and an NVIDIA A100 80GB (PCIe) GPU:

```
import torch

def do_math(device):
    inv_freq = (10 ** (torch.arange(0, 10, device=device) / 10))
    print(f"{inv_freq[9]:.20f}")
    return inv_freq.cpu()

a = do_math(torch.device("cpu"))
b = do_math(torch.device("cuda"))

torch.testing.assert_close(a, b, rtol=0.0, atol=0.0)
```

when we run it we get 2 out of 10 elements mismatch:

```
7.94328212738037109375
7.94328308105468750000
[...]
AssertionError: Tensor-likes are not equal!

Mismatched elements: 2 / 10 (20.0%)
Greatest absolute difference: 9.5367431640625e-07 at index (9,)
Greatest relative difference: 1.200604771156577e-07 at index (9,)
```

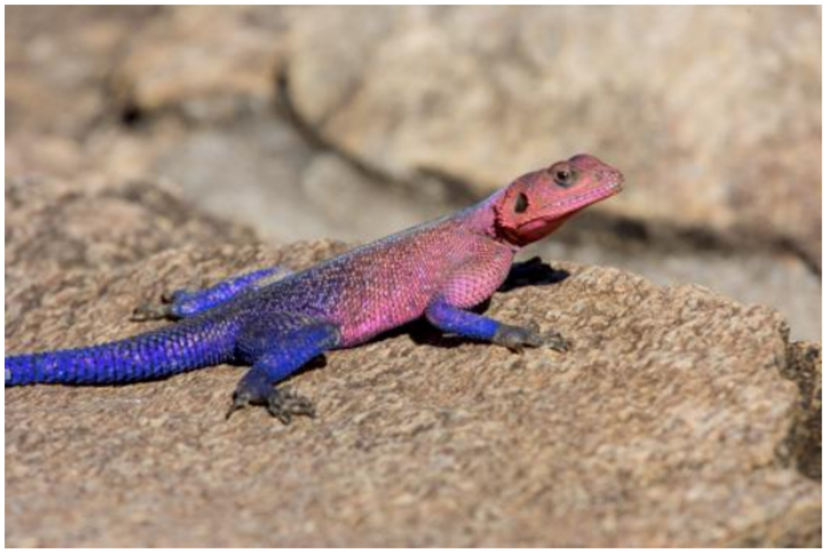
This was a simple low-dimensional example, but in reality the tensors are much bigger and will typically end up having more mismatches.

Now you might say that the `1e-6` discrepancy can be safely ignored. And it's often so as long as this is a final result. If this tensor from the example above is now fed through a 100 layers of `matmuls`, this tiny discrepancy is going to compound and spread out to impact many other elements with the final outcome being quite different from the same action performed on another type of device.

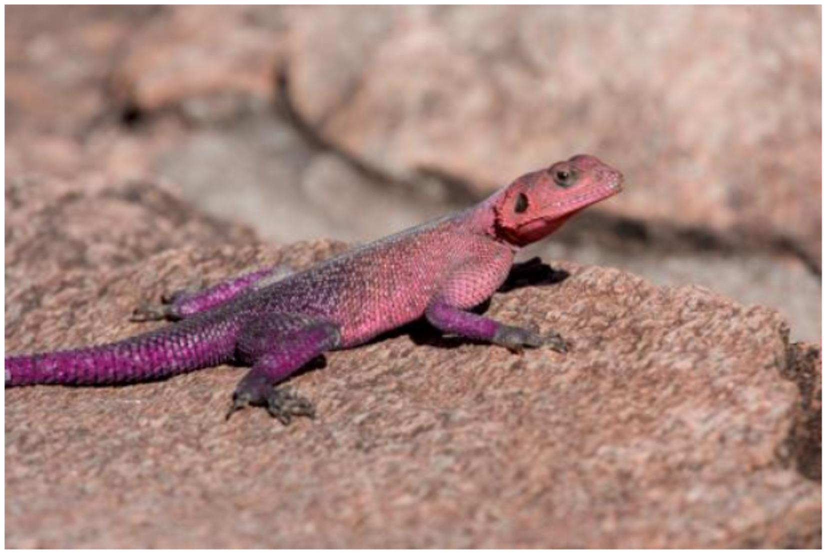
For example, see this discussion - the users reported that when doing Llama-2-7b inference they were getting quite different logits depending on how the model was initialized. To clarify the initial discussion was about Deepspeed potentially being the problem, but in later comments you can see that it was reduced to just which device the model's buffers were initialized on. The trained weights aren't an issue they are loaded from the checkpoint, but the buffers are recreated from scratch when the model is loaded, so that's where the problem emerges.

It's uncommon that small variations make much of a difference, but sometimes the difference can be clearly seen, as in this example where the same image is produced on a CPU and an MPS device.

To better illustrate the problem that motivated me to post this issue, I can present some output from CURL. This first image is from the CPU:



This is from the mps device:



The difference in output comes down to a cascade of tiny differences in fp values that avalanche into large differences. I think these calculations need to be *exactly* the same on the cpu and mps device to ensure correctness. Otherwise, I don't see how the mps device can be relied on.

This snapshot and the commentary come from this PyTorch Issue thread.

If you're curious where I pulled this code from - this is a simplified reduction of this original code in modeling_llama.py:

```
class LlamaRotaryEmbedding(nn.Module):
    def __init__(self, dim, max_position_embeddings=2048, base=10000, device=None):
        super().__init__()

        self.dim = dim
        self.max_position_embeddings = max_position_embeddings
        self.base = base
        inv_freq = 1.0 / (self.base ** (torch.arange(0, self.dim, 2).float().to(device) / se
        self.register_buffer("inv_freq", inv_freq, persistent=False)
```