(troubleshooting)=

Troubleshooting

This document outlines some troubleshooting strategies you can consider. If you think you've discovered a bug, please [search existing issues](https://github.com/vllm-project/vllm/issues?q=is%3Aissue) first to see if it has already been reported. If not, please [file a new issue](https://github.com/vllm-project/vllm/issues/new/choose), providing as much relevant information as possible.

:::{note}
Once you've debugged a problem, remember to turn off any debugging environment variables defined, or simply start a new shell to avoid being affected by lingering debugging settings. Otherwise, the system might be slow with debugging functionalities left activated.
:::

Hangs downloading a model

If the model isn't already downloaded to disk, vLLM will download it from the internet which can take time and depend on your internet connection.
It's recommended to download the model first using the [huggingface-cli](https://huggingface.co/docs/huggingface_hub/en/guides/cli) and passing the local path to the model to vLLM. This way, you can isolate the issue.

Hangs loading a model from disk

If the model is large, it can take a long time to load it from disk. Pay attention to where you store the

model. Some clusters have shared filesystems across nodes, e.g. a distributed filesystem or a network filesystem, which can be slow.

It'd be better to store the model in a local disk. Additionally, have a look at the CPU memory usage, when the model is too large it might take a lot of CPU memory, slowing down the operating system because it needs to frequently swap between disk and memory.

:::{note}

To isolate the model downloading and loading issue, you can use the `--load-format dummy` argument to skip loading the model weights. This way, you can check if the model downloading and loading is the bottleneck.

:::

Out of memory

If the model is too large to fit in a single GPU, you will get an out-of-memory (OOM) error. Consider [using tensor parallelism](#distributed-serving) to split the model across multiple GPUs. In that case, every process will read the whole model and split it into chunks, which makes the disk reading time even longer (proportional to the size of tensor parallelism). You can convert the model checkpoint to a sharded checkpoint using <gh-file:examples/offline_inference/save_sharded_state.py>. The conversion process might take some time, but later you can load the sharded checkpoint much faster. The model loading time should remain constant regardless of the size of tensor parallelism.

Enable more logging

If other strategies don't solve the problem, it's likely that the vLLM instance is stuck somewhere. You can use the following environment variables to help debug the issue:

- `export VLLM_LOGGING_LEVEL=DEBUG` to turn on more logging.
- `export CUDA_LAUNCH_BLOCKING=1` to identify which CUDA kernel is causing the problem.
- `export NCCL_DEBUG=TRACE` to turn on more logging for NCCL.
- `export VLLM_TRACE_FUNCTION=1` to record all function calls for inspection in the log files to tell which function crashes or hangs.

Incorrect network setup

The vLLM instance cannot get the correct IP address if you have a complicated network config. You can find a log such as `DEBUG 06-10 21:32:17 parallel_state.py:88] world_size=8 rank=0 local_rank=0 distributed_init_method=tcp://xxx.xxx.xxx.xxx:54641 backend=nccl` and the IP address should be the correct one.
If it's not, override the IP address using the environment variable `export VLLM_HOST_IP=<your_ip_address>`.

You might also need to set `export NCCL_SOCKET_IFNAME=<your_network_interface>` and `export GLOO_SOCKET_IFNAME=<your_network_interface>` to specify the network interface for the IP address.

Error near `self.graph.replay()`

If vLLM crashes and the error trace captures it somewhere around `self.graph.replay()` in `vllm/worker/model_runner.py`, it is a CUDA error inside CUDAGraph.
To identify the particular CUDA operation that causes the error, you can add `--enforce-eager` to the command line, or `enforce_eager=True` to the {class}`~vllm.LLM` class to disable the CUDAGraph optimization and isolate the exact CUDA operation that causes the error.

(troubleshooting-incorrect-hardware-driver)=

Incorrect hardware/driver

If GPU/CPU communication cannot be established, you can use the following Python script and follow the instructions below to confirm whether the GPU/CPU communication is working correctly.

```python
# Test PyTorch NCCL
import torch
import torch.distributed as dist
dist.init_process_group(backend="nccl")
local_rank = dist.get_rank() % torch.cuda.device_count()
torch.cuda.set_device(local_rank)
data = torch.FloatTensor([1,] * 128).to("cuda")
dist.all_reduce(data, op=dist.ReduceOp.SUM)
torch.cuda.synchronize()
value = data.mean().item()
world_size = dist.get_world_size()
assert value == world_size, f"Expected {world_size}, got {value}"

print("PyTorch NCCL is successful!")

# Test PyTorch GLOO
gloo_group = dist.new_group(ranks=list(range(world_size)), backend="gloo")
cpu_data = torch.FloatTensor([1,] * 128)
dist.all_reduce(cpu_data, op=dist.ReduceOp.SUM, group=gloo_group)
```

```
value = cpu_data.mean().item()
assert value == world_size, f"Expected {world_size}, got {value}"

print("PyTorch GLOO is successful!")

if world_size <= 1:
    exit()

# Test vLLM NCCL, with cuda graph
from vllm.distributed.device_communicators.pynccl import PyNcclCommunicator

pynccl = PyNcclCommunicator(group=gloo_group, device=local_rank)
# pynccl is enabled by default for 0.6.5+,
# but for 0.6.4 and below, we need to enable it manually.
# keep the code for backward compatibility when because people
# prefer to read the latest documentation.
pynccl.disabled = False

s = torch.cuda.Stream()
with torch.cuda.stream(s):
    data.fill_(1)
    pynccl.all_reduce(data, stream=s)
    value = data.mean().item()
    assert value == world_size, f"Expected {world_size}, got {value}"

print("vLLM NCCL is successful!")
```

```
g = torch.cuda.CUDAGraph()
with torch.cuda.graph(cuda_graph=g, stream=s):
    pynccl.all_reduce(data, stream=torch.cuda.current_stream())

data.fill_(1)
g.replay()
torch.cuda.current_stream().synchronize()
value = data.mean().item()
assert value == world_size, f"Expected {world_size}, got {value}"

print("vLLM NCCL with cuda graph is successful!")

dist.destroy_process_group(gloo_group)
dist.destroy_process_group()
```

If you are testing with a single node, adjust `--nproc-per-node` to the number of GPUs you want to use:

```console
NCCL_DEBUG=TRACE torchrun --nproc-per-node=<number-of-GPUs> test.py
```

If you are testing with multi-nodes, adjust `--nproc-per-node` and `--nnodes` according to your setup and set `MASTER_ADDR` to the correct IP address of the master node, reachable from all nodes. Then, run:

```console
NCCL_DEBUG=TRACE torchrun --nnodes 2 --nproc-per-node=2 --rdzv_backend=c10d --rdzv_endpoint=$MASTER_ADDR test.py
```

If the script runs successfully, you should see the message `sanity check is successful!`.

If the test script hangs or crashes, usually it means the hardware/drivers are broken in some sense. You should try to contact your system administrator or hardware vendor for further assistance. As a common workaround, you can try to tune some NCCL environment variables, such as `export NCCL_P2P_DISABLE=1` to see if it helps. Please check [their documentation](https://docs.nvidia.com/deeplearning/nccl/user-guide/docs/env.html) for more information. Please only use these environment variables as a temporary workaround, as they might affect the performance of the system. The best solution is still to fix the hardware/drivers so that the test script can run successfully.

:::{note}
A multi-node environment is more complicated than a single-node one. If you see errors such as `torch.distributed.DistNetworkError`, it is likely that the network/DNS setup is incorrect. In that case, you can manually assign node rank and specify the IP via command line arguments:

- In the first node, run `NCCL_DEBUG=TRACE torchrun --nnodes 2 --nproc-per-node=2 --node-rank 0 --master_addr $MASTER_ADDR test.py`.
- In the second node, run `NCCL_DEBUG=TRACE torchrun --nnodes 2 --nproc-per-node=2 --node-rank 1 --master_addr $MASTER_ADDR test.py`.

Adjust `--nproc-per-node`, `--nnodes`, and `--node-rank` according to your setup, being sure to

execute different commands (with different `--node-rank`) on different nodes.

:::

(troubleshooting-python-multiprocessing)=

Python multiprocessing

`RuntimeError` Exception

If you have seen a warning in your logs like this:

```console
WARNING 12-11 14:50:37 multiproc_worker_utils.py:281] CUDA was previously
    initialized. We must use the `spawn` multiprocessing start method. Setting
    VLLM_WORKER_MULTIPROC_METHOD to 'spawn'. See
    https://docs.vllm.ai/en/latest/getting_started/troubleshooting.html#python-multiprocessing
    for more information.
```

or an error from Python that looks like this:

```console
RuntimeError:
        An attempt has been made to start a new process before the
        current process has finished its bootstrapping phase.

        This probably means that you are not using fork to start your
```

child processes and you have forgotten to use the proper idiom
in the main module:

if __name__ == '__main__':
freeze_support()
...

The "freeze_support()" line can be omitted if the program
is not going to be frozen to produce an executable.

To fix this issue, refer to the "Safe importing of main module"
section in https://docs.python.org/3/library/multiprocessing.html
```

then you must update your Python code to guard usage of `vllm` behind a `if __name__ == '__main__':` block. For example, instead of this:

```python
import vllm

llm = vllm.LLM(...)
```

try this instead:

```python
if __name__ == '__main__':
```
```

```
    import vllm

    llm = vllm.LLM(...)
```

`torch.compile` Error

vLLM heavily depends on `torch.compile` to optimize the model for better performance, which introduces the dependency on the `torch.compile` functionality and the `triton` library. By default, we use `torch.compile` to [optimize some functions](https://github.com/vllm-project/vllm/pull/10406) in the model. Before running vLLM, you can check if `torch.compile` is working as expected by running the following script:

```python
import torch

@torch.compile
def f(x):
    # a simple function to test torch.compile
    x = x + 1
    x = x * 2
    x = x.sin()
    return x

x = torch.randn(4, 4).cuda()
print(f(x))
```

If it raises errors from `torch/_inductor` directory, usually it means you have a custom `triton` library that is not compatible with the version of PyTorch you are using. See [this issue](https://github.com/vllm-project/vllm/issues/12219) for example.

Model failed to be inspected

If you see an error like:

```text
  File "vllm/model_executor/models/registry.py", line xxx, in _raise_for_unsupported
    raise ValueError(
ValueError: Model architectures ['<arch>'] failed to be inspected. Please check the logs for more details.
```

It means that vLLM failed to import the model file.

Usually, it is related to missing dependencies or outdated binaries in the vLLM build.

Please read the logs carefully to determine the root cause of the error.

Model not supported

If you see an error like:

```text
Traceback (most recent call last):
...
```

```
  File "vllm/model_executor/models/registry.py", line xxx, in inspect_model_cls
    for arch in architectures:
TypeError: 'NoneType' object is not iterable
```

or:

```text
  File "vllm/model_executor/models/registry.py", line xxx, in _raise_for_unsupported
    raise ValueError(
ValueError: Model architectures ['<arch>'] are not supported for now. Supported architectures: [...]
```

But you are sure that the model is in the [list of supported models](#supported-models), there may be some issue with vLLM's model resolution. In that case, please follow [these steps](#model-resolution) to explicitly specify the vLLM implementation for the model.

Known Issues

- In `v0.5.2`, `v0.5.3`, and `v0.5.3.post1`, there is a bug caused by [zmq](https://github.com/zeromq/pyzmq/issues/2000) , which can occasionally cause vLLM to hang depending on the machine configuration. The solution is to upgrade to the latest version of `vllm` to include the [fix](gh-pr:6759).
- To circumvent a NCCL [bug](https://github.com/NVIDIA/nccl/issues/1234) , all vLLM processes will set an environment variable `NCCL_CUMEM_ENABLE=0` to disable NCCL's `cuMem` allocator. It does not affect performance but only gives memory benefits. When external processes want to set up a NCCL connection with vLLM's processes, they should also set this environment variable,

otherwise, inconsistent environment setup will cause NCCL to hang or crash, as observed in the [RLHF integration](https://github.com/OpenRLHF/OpenRLHF/pull/604) and the [discussion](gh-issue:5723#issuecomment-2554389656) .