

关于集群分布式torchrun命令踩坑记录（自用）

原创

萌新玉玉玉

于 2023-05-19 16:17:35 发布

阅读量3.3w

收藏 77

点赞数 23


版权

文章标签：

分布式

pytorch

深度学习

 GitCode 开源社区

文章已被社区收录

加入社区

项目场景：

在训练或者微调模型的过程中，单节点的显存溢出，或者单节点的显卡较少，**算力** 有限。需要跨节点用多个节点多块显卡来运行这项任务。这里就需要使用分布式命令，将这项任务分布到多个节点上来处理。

问题描述

在此我仅记录我在运行分布式过程中遇到的一些问题。

首先，对于pytorch**深度学习** 框架的分布式进程是有一套标准的流程的，简单来讲就是先通过dist进行初始化，再将模型进行分布式分配。具体所用的API为：

```
1 | import torch.distributed as dist
2 | from torch.nn.parallel import DistributedDataParallel as DDP
```

对于预训练（或者微调）的脚本，我参考了官方文档中的测试代码，测试代码实际非常简单，搭建了一个非常小的网络，同时对其使用分布式进程，非常适合拿来测试。链接为：[官方文档](#)

dist.init_process_group()是PyTorch中用于初始化分布式训练的函数之一。它用于设置并行训练环境，连接多个进程以进行数据和模型的分布式处理。我们通过init_process_group()函数这个方法来进行初始化，其参数包括以下内容

1. backend（必需参数）：指定分布式后端的类型，可以是以下选项之一：
- ‘tcp’：使用TCP协议进行通信。

‘gloo’：使用Gloo库进行通信。

‘mpi’：使用MPI（Message Passing Interface）进行通信。

‘nccl’：使用NCCL库进行通信（适用于多GPU的分布式训练）。

‘hccl’：使用HCCL库进行通信（适用于华为昇腾AI处理器的分布式训练）。
2. init_method（可选参数）：指定用于初始化分布式环境的方法。它可以是以下选项之一：
- ‘env://’：使用环境变量中指定的方法进行初始化。

‘file://’：使用本地文件进行初始化。

‘tcp://’：

：使用TCP地址和端口进行初始化。

‘gloo://’：

：使用Gloo地址和端口进行初始化。

‘mpi://’：

：使用MPI地址和端口进行初始化。
3. rank（可选参数）：指定当前进程的排名（从0开始）。
4. world_size（可选参数）：指定总共使用的进程数。
5. timeout（可选参数）：指定初始化的超时时间。
6. group_name（可选参数）：指定用于连接的进程组名称。

这里由于服务器采用的slurm系统，我们开始计划使用mpi去实现分布式分发，同时torch的初始化也支持mpi，原始想法是通过mpirun来进行分布式计算。但是，**如果要使用mpi来实现分布式功能，必须要通过github上的源代码进行编译，通过conda和pip进行下载的pytorch自身是不携带mpi的**

通过上面的参数，可以看到backend是有多重通信方式的，常用的有gloo和mpi和nccl，但是这三者是有区别的，这里我们可以参考官方文档：[官方](#)

文档

Backend	gloo		mpi		nccl	
Device	CPU	GPU	CPU	GPU	CPU	GPU
send	✓	✗	✓	?	✗	✓
recv	✓	✗	✓	?	✗	✓
broadcast	✓	✓	✓	?	✗	✓
all_reduce	✓	✓	✓	?	✗	✓
reduce	✓	✗	✓	?	✗	✓
all_gather	✓	✗	✓	?	✗	✓
gather	✓	✗	✓	?	✗	✓
scatter	✓	✗	✓	?	✗	✓
reduce_scatter	✗	✗	✗	✗	✗	✓
all_to_all	✗	✗	✓	?	✗	✓
barrier	✓	✗	✓	?	✗	✓

CSDN @萌新玉玉玉

这里我们直接放出结论，以供参考：

- 对于分布式 GPU 训练，使用 NCCL 后端。
- 对于分布式 CPU 训练，使用 Gloo 后端。
- 如果你的主机是 GPU 主机并且具有 InfiniBand 互连： 使用 NCCL，因为它是目前唯一支持 InfiniBand 和 GPUDirect 的后端。
- 如果你的主机是 GPU 主机并且具有以太网互连： 使用 NCCL，因为它目前提供了最好的分布式 GPU 训练性能，特别是对于多进程单节点或多节点分布式训练。
- 如果你遇到 NCCL 的任何问题，使用 Gloo 作为备选选项。（注意，Gloo 目前运行速度比 NCCL 慢）
- 如果你的主机是 CPU 主机并且具有 InfiniBand 互连： 如果你的 InfiniBand 启用了 IP over IB，使用 Gloo，否则，使用 MPI。我们计划在即将发布的版本中为 Gloo 添加 InfiniBand 支持。
- 如果你的主机是 CPU 主机并且具有以太网互连： 使用 Gloo，除非你有特定的理由使用 MPI。

我们可以根据文档的提示，得出，MPI是最不推荐使用的一种方法，对于英伟达的显卡，最优先的还是使用NCCL方法。和Mpi相匹配的有一种torch官方自带的方法，在torch2.0之前使用的API叫：**torch.distributed.launch**在使用时显示未来的版本将会弃用这个API，取而代之的是**torchrun**。因此我们将命令由mpi改为torchrun方法，在dist初始化使用nccl后端通信。

这里我们可以参考torchrun的官方运行方法：[官方文档](#)

经过近两周的调试与踩坑，先是在测试节点上对官方测试脚本进行分布式测试，运行成功后再将相同的环境和文件迁移到计算节点上，再在计算节点上对分布式命令进行测试，期间还测试了root用户和子用户对torchrun命令是否会有影响。

假设我们有三个节点，node02，node03，node04，每个节点上有四张GPU。现在我们将官方测试文档中的代码写为test_mpi.py。最终通过torchrun实现的命令如下：

```
1 | torchrun --nproc_per_node=4 --nnodes=3 --node_rank=0 --master_addr=192.168.0.101 --master_port=29500 test_mpi.py
```

我们没有必要和torchrun的官方文档一样去设置**--rdzv-backend** 和**--rdzv-id**，因为这不是必须的，用默认的即可。我们只需要设置的参数只有上面这几个。具体参数介绍如下：

--nproc_per_node=4：指定每个节点（机器）上的进程数，这里是4个。意味着每个机器将启动4个进程来参与分布式训练。

- --nnodes=3：指定总共的节点数，这里是3个。意味着总共有3个机器参与分布式训练。
- --node_rank=0：指定当前节点（机器）的排名，这里是0。排名从0开始，用于在分布式环境中区分不同的节点。
- --master_addr=192.168.0.101：指定主节点的IP地址，这里是192.168.0.101。主节点用于协调分布式训练过程。
- --master_port=29500：指定主节点的端口号，这里是29500。主节点使用指定的端口来与其他节点进行通信。

通过设置这些参数，该命令将在3个节点的分布式环境中启动4个进程，并指定192.168.0.101作为主节点进行协调和通信。这里的主节点地址我随便写的，可以根据实际情况进行修改。主节点的地址的--node_rank必须设置为0，也就是上述这行命令，必须要先在主节点上线运行。

举个例子，假如我的主节点是node02，那么我就要先在node02节点的终端上运行上述torchrun命令，同时--master_addr要为node02的ip地址（查看IP地址可以通过：ip addr），然后node03，node04的顺序就不重要了，在其节点的终端上将--node_rank=0改为--node_rank=1和--node_rank=2运行即可。

这里出现第一个问题，即是，通讯超时（具体表现为：**ERROR:torch.distributed.elastic.multiprocessing.api:failed (exitcode: -11)**）。假如我们的节点之前ping方法没有问题，同时节点并没有处于被占用的情况，那么分析超时就比较困难了。我会在之后的解决方法中，提供我是如何解决的。

```
WARNING:torch.distributed.elastic.multiprocessing.api:Sending process 7035 closing signal SIGTERM
WARNING:torch.distributed.elastic.multiprocessing.api:Sending process 7037 closing signal SIGTERM
WARNING:torch.distributed.elastic.multiprocessing.api:Sending process 7040 closing signal SIGTERM
ERROR:torch.distributed.elastic.multiprocessing.api:failed (exitcode: -11) local_rank: 3 (pid: 7043) o
```

在命令确认无误之后，我们需要将这个命令，写成脚本提交到后台，挂在服务器上运行，而不是在终端上一直在线处理。

由于我们服务器使用的slurm系统，slurm系统自带一套可以提交作业的方法。于是就要将这个命令进行sbatch脚本打包。打包的bash脚本如下所示：

```
1  #!/bin/bash
2  #SBATCH --job-name=pytorch_job      # 创建一个任务名
3  #SBATCH -N 3                        # 需要使用的节点数
4  #SBATCH --ntasks-per-node=4        # 每个节点上的任务数
5  #SBATCH --output=job_output.out     # 标准输出文件
6  #SBATCH --error=job_error.err       # 标准错误文件
7  #SBATCH --odelist=node02,node03,node04 # 指定节点列表
8
9  # 加载任何必要的模块，例如：
10 # module load python
11 # module load pytorch
12 # source .....
13
14 export TORCH_DISTRIBUTED_DEBUG=INFO
15 export NCCL_IB_DISABLE=1
16
17
18 # 设置主节点
19 # 节点列表
20 NODELIST=$(scontrol show hostname $SLURM_JOB_NODELIST)
21 # 对第一个节点赋值为主节点
22 MASTER_NODE=$(head -n 1 <<< "$NODELIST")
23 # 计数器
24 NODE_COUNT=0
25 # 一共的节点数
26 NODE_NUM=$(echo $NODELIST | tr " " "\n" | wc -l)
27
28 # 打印
29 echo $SLURM_NODEID
30 echo $NODELIST
31 echo $MASTER_NODE
32 echo $NODE_NUM
```

```
31
32   for NODE in $NODELIST; do
33       if [ "$NODE" == "$MASTER_NODE" ]; then
34           srun --nodes=1 --ntasks=1 -w $NODE torchrun --nproc_per_node=4 --nnodes=$NODE_NUM --node_rank=0 --master_
35       else
36           ((NODE_COUNT++))
37           srun --nodes=1 --ntasks=1 -w $NODE torchrun --nproc_per_node=4 --nnodes=$NODE_NUM --node_rank=$NODE_COUNT
38       fi
39   done
40   wait
41
42
```

脚本的逻辑为：通过srun在启动的每个节点上运行torchrun命令，运行的同时还需要进行判断，判断提交的节点是不是主节点，如果是主节点则node_rank的值要为0，如果不是主节点则node_rank的值为1,2.....其实并不推荐使用sbatch嵌套srun（）

这里出现**第二个问题**，假如不是不是在主节点第一个运行命令，则会发生超时，具体情况如下：

TimeoutError: The client socket has timed out after 900s while trying to connect to
crunch error: node07: task 0: Failed with exit code 1
我会在之后的解决方法中，提供我是如何解决的。

原因分析：

对于上述的两种超时问题，首先要做的是在节点之间进行ping操作确认，确认不是服务器本身的问题，则考虑是不是节点之间的通信问题。因为NCCL也是有内部通信的，NVIDIA 的NCCL库支持多种传输方式，以便在不同的硬件和网络配置中实现最优的通信 性能 。以下是一些主要的传输方式：

InfiniBand (IB)：如前所述，InfiniBand是一种高性能、低延迟的网络传输技术，常用于高性能计算（HPC）和数据中心。

TCP/IP：这是最常见的网络通信协议，可以在任何支持TCP/IP的网络（包括常见的以太网）上运行。

Shared Memory (SHM)：在同一台机器上的GPU之间，NCCL可以使用共享内存进行通信。这通常比通过网络传输更快。

CUDA Inter-Process Communication (IPC)：对于同一节点上的多个GPU，NCCL可以使用CUDA IPC进行通信。这是一种允许不同CUDA进程共享GPU内存的机制，可以提高通信效率。

NVLink：NVLink是NVIDIA的一种高速互连技术，用于连接GPU和GPU，或GPU和CPU。它提供了比传统PCIe更高的带宽，适用于需要高速GPU间通信的应用。

这些传输方式可以根据具体的硬件配置和通信需求进行选择 and 配置。

```
icmp_seq=1 ttl=64 time=0.123 ms
icmp_seq=2 ttl=64 time=0.050 ms
icmp_seq=3 ttl=64 time=0.056 ms
icmp_seq=4 ttl=64 time=0.049 ms
icmp_seq=5 ttl=64 time=0.062 ms
icmp_seq=6 ttl=64 time=0.050 ms
icmp_seq=7 ttl=64 time=0.049 ms
icmp_seq=8 ttl=64 time=0.044 ms
icmp_seq=9 ttl=64 time=0.045 ms
```

解决方案：

我们可以在之前的脚本中，添加环境变量，进入调试模型，查看具体的原因：

```
1   export NCCL_DEBUG=INFO
2   export NCCL_DEBUG_SUBSYS=ALL
3   export TORCH_DISTRIBUTED_DEBUG=INFO
```

对于第一个问题，再次运行我们的命令，即可获得NCCL的INFO信息，我们详细对比信息可以发现，在主节点上，我们使用的通信方式是NET/IB，如下图所示：

```
NCCL INFO NET/IB : Using [0]mlx4_0:1/IB
NCCL INFO Using network IB
```

而在其他节点，我们使用的方式是 NET/Socket

```
84 [0] NCCL INFO NET/Socket : Using [0]p6p
84 [0] NCCL INFO Using network Socket
```

NET/IB 和 NET/Socket 是两种不同的网络通信接口。NET/IB 通常指的是 InfiniBand，这是一种高性能、低延迟的网络通信接口，常用于高性能计算和数据中心。而 NET/Socket 则是一种更常见的网络接口，它在各种网络环境中都可以使用。如果你的两个节点一个使用 NET/IB，另一个使用 NET/Socket，那么这两个节点之间的通信可能会受到影响。因为 NCCL 默认使用最快的可用传输方法，如果两个节点的网络接口不同，那么可能

无法建立有效的通信。具体情况可能根据你的网络环境和配置进行测试。这里建议使用同一种通信方式。
我们将IB禁用即可：

```
1 | export NCCL_IB_DISABLE=1
```

对于第二个问题，我们只需要写判断语句，确保主节点运行node_rank=0的命令即可，在上述给出的代码我已经写好了判断语句。