(3) --- Horovodrun背后做了什么

0x01 背景知识

首先介绍一些相关背景知识。

1.1 分布式体系

在设计并行计算机时，最直接的方式就是多个计算单元共享一个内存。共享内存的编程在数据交换和访问上有较大的优势，程序编写起来更加简单。但在扩展性上有较大的瓶颈。

另一种方式为 分布式内存。即每个计算单元有单独的内存，计算单元之间的数据访问通过互联网络去传输。这一架构在可移植性和扩展上会强很多，但消息的传递会成为程序设计中的难点。

将这两点结合，即是分布式共享内存并行计算机的架构，也是当今最常用的体系结构。

1.2 并行任务通信

并行任务通信一般分为P2P（Point-to-point communication）和 Collective communication。

- P2P通信这种模式只有一个sender和一个receiver，即点到点通信。
- Collective communication含多个sender多个receive。

Collective communication包含一些常见的原语

- broadcast
- reduce，allreduce
- scatter，scatter reduce
- gather，allgather
- ring-base collectives
- ring-allreduce

传统Collective communication假设通信节点组成的topology是一颗fat tree，这样通信效率最高。但实际的通信topology可能比较复杂，并不是一个fat tree。因此一般用ring-based Collective communication。

1.3 MPI

MPI(Message Passing Interface) 是一种可以支持点对点和广播的通信协议，具体实现的库有很多，使用比较流行的包括 Open Mpi， Intel MPI 等等。

MPI 是一种消息传递编程模型。消息传递指用户必须通过显式地发送和接收消息来实现处理器间的数据交换。在这种并行编程中，每个控制流均有自己独立的地址空间，不同的控制流之间不能直接访问彼此的地址空间，必须通过显式的消息传递来实现。这种编程方式是大规模并行处理机(MPP)和机群(Cluster)采用的主要编程方式。由于消息传递程序设计要求用户很好地分解问题，组织不同控制流间的数据交换，并行计算粒度大，特别适合于大规模可扩展并行算法。

MPI 是基于进程的并行环境。进程拥有独立的虚拟地址空间和处理器调度，并且执行相互独立。MPI 设计为支持通过网络连接的机群系统，且通过消息传递来实现通信，消息传递是 MPI 的最基本特色。

1.4 Open-MPI

OpenMPI 是一种高性能消息传递库，最初是作为融合的技术和资源从其他几个项目（FT-MPI， LA-MPI， LAM/MPI， 以及 PACX-MPI），它是 MPI-2 标准的一个开源实现，由一些科研机构和企业一起开发和维护。因此，OpenMPI 能够从高性能社区中获得专业技术、工业技术和资源支持，来创建最好的 MPI 库。OpenMPI 提供给系统和软件供应商、程序开发者和研究人员很多便利。易于使用，并运行本身在各种各样的操作系统，网络互连，以及一批/调度系统。

1.5 MPI 使用问题

因为MPI是分布式内存编程，在后面的开发中涉及节点间信息的传递。往往数据和程序是在多个节点上，所以需要保证执行命令时各节点之间信息的交换。

具体使用之中，就有两个问题：

- 这个多台机器Open-MPI是如何发现并建立连接的呢？
- 多机多卡在训练过程中，传输环如何建立，这个也是决定了训练效率，那么Open-MPI如何去做呢？

关于第一个问题：

设置SSH免密登录可以免去操作中密码的输入。各节点生成私钥和公钥后需要认证，此时可以保证本机免密登录。将各个子节点的公钥文件发送给主节点，然后分别加入到主节点的认证文件中，此时可以保证主节点对各个子节点的免密登录。最后将认证文件传回到每个子节点，从而保证各个子节点对其他节点之间的免密登录。

在 Open-MPI 启动的时候，可以指定`--hostfile`或者`--host`去指定运行要运行任务的 IP 或 Hostname，这样 Open-MPI 就会试图通过 ssh 免秘钥的方式试图去链接对方机器，并执行一系列命令，主要是为了同步环境变量、当前路径以及下发启动命令。

当然用户也可以通过其他方式给远程机器下发命令，这个可以通过环境变量`OMPI_MCA_plm_rsh_agent`指定。

关于第二个问题：

当所有的机器建立好连接了，准备开始计算了，为了能够最高效的去通信，Open-MPI中集成了组件——hwloc。该组件主要是为了单机硬件资源拓扑构建，进而构建最短路径通信。

0x02 入口点

很多机器学习框架都会采用如下套路：shell脚本（可选），python端 和 C++端。

- Shell脚本是启动运行的入口，负责解析参数，确认并且调用训练程序；
- Python是用户的接口，引入了C++库，封装了API，负责运行时和底层C++交互；
- C++实现底层训练逻辑；

所以我们先看看 hordovodrun 脚本。

2.1 如何运行

官方给出的 Hovorod 运行范例之一如下：

```
horovodrun -np 2 -H localhost:4 --gloo python
/horovod/examples/tensorflow2/tensorflow2_mnist.py
```

这里 -np 指的是进程的数量，localhost:4表示localhost节点上4个GPU。

注意，如果虚拟机只有一个核。想要强行地达到并行的效果，可以使用 -np参数，它会自动帮你把一个核心切成多份处理器，每一个分布式处理就是一个slot。

因此，我们可以从 horovodrun 这个命令入手看看。

2.2 horovodrun

入口文件可以从 setup.py 看到，其就被映射成 horovod.runner.launch:run_commandline。

```
entry_points={
    'console_scripts': [
        'horovodrun = horovod.runner.launch:run_commandline'
    ]
}
```

所以我们看看 run_commandline

2.3 run_commandline

该命令位于：horovod-master/horovod/runner/launch.py，我们摘录重要部分。

```
def run_commandline():
    args = parse_args()
    _run(args)
```

于是进入到 _run 函数。可以看到，Horovod 会依据是否是弹性训练来选择不同的路径。我们在此系列中，会首先分析 非弹性训练 _run_static。

```
def _run(args):
    # if hosts are not specified, either parse from hostfile, or default as
    # localhost
    if not args.hosts and not args.host_discovery_script:
        if args.hostfile:
            args.hosts = hosts.parse_host_files(args.hostfile)
        else:
            # Set hosts to localhost if not specified
            args.hosts = 'localhost:{np}'.format(np=args.np)

    # Convert nics into set
    args.nics = set(args.nics.split(',')) if args.nics else None

    if _is_elastic(args):
        return _run_elastic(args)
    else:
        return _run_static(args) # 我们先看这里
```

2.4 非弹性训练 _run_static

在 _run_static 之中做了如下操作：

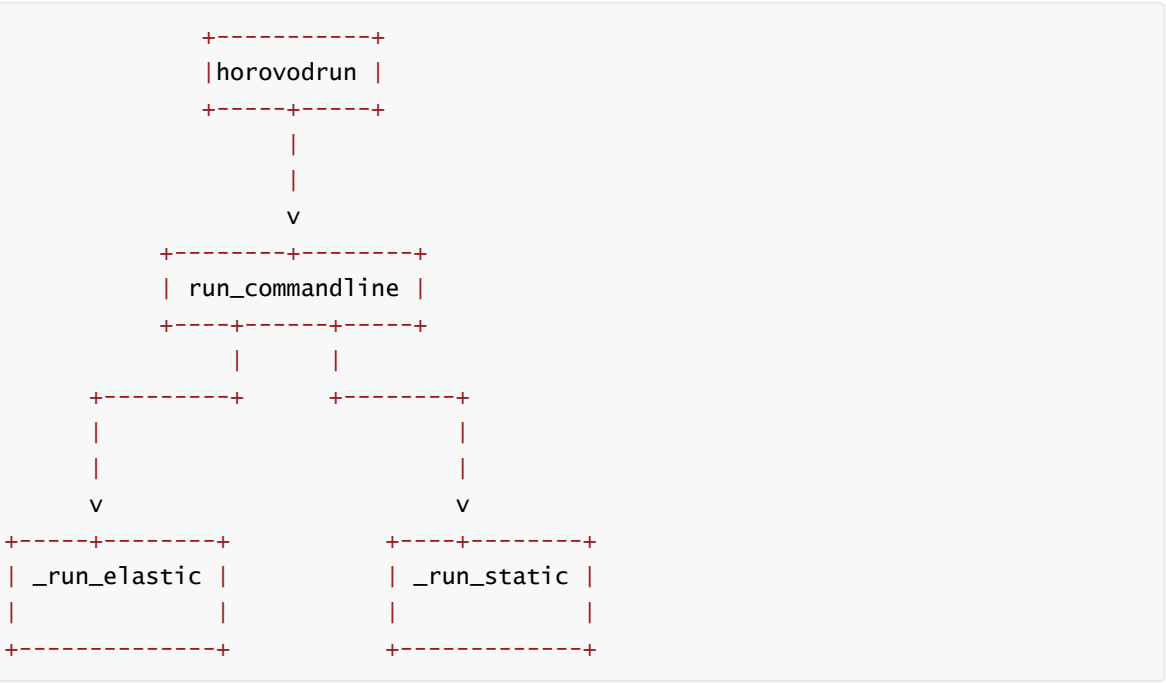
- 首先解析各种参数，得到 settings；
- 会调用 `driver_service.get_common_interfaces` 获取网卡以及其他host的信息，依据这些信息会进行slot分配，这部分很复杂，具体我们会有专文讲解（下一篇）。
- 这里有一个问题：为什么要得到 host, slot, rank 之间的关系信息？由于工程上的考虑，底层 C++ 世界中对于 rank 的角色做了区分：rank 0 是 master，rank n 是 worker，所以这些信息需要决定并且传递给 C++世界；
- 会根据是否在参数中传递运行函数来决定采取何种路径，一般默认没有运行参数，所以会执行 _launch_job 来启动训练 job；

具体代码如下：

```
def _run_static(args):

    settings = hvd_settings.Settings(verbose=2 if args.verbose else 0,
                                     ssh_port=args.ssh_port,
                                     ssh_identity_file=args.ssh_identity_file,
                                     extra_mpi_args=args.mpi_args,
                                     tcp_flag=args.tcp_flag,
                                     binding_args=args.binding_args,
                                     key=secret.make_secret_key(),
                                     start_timeout=tmout,
                                     num_proc=args.np,
                                     hosts=args.hosts,
                                     output_filename=args.output_filename,
                                     run_func_mode=args.run_func is not None,
                                     nics=args.nics,...)
      # 首先解析各种参数，得到 settings
    fn_cache = None
    if not args.disable_cache:
        params = ''
        if args.np:
            params += str(args.np) + ' '
        if args.hosts:
            params += str(args.hosts) + ' '
        if args.ssh_port:
            params += str(args.ssh_port)
        if args.ssh_identity_file:
            params += args.ssh_identity_file
        parameters_hash = hashlib.md5(params.encode('utf-8')).hexdigest()
        fn_cache = cache.Cache(CACHE_FOLDER, CACHE_STALENESS_THRESHOLD_MINUTES,
                               parameters_hash)

    # 获取网卡以及其他host的信息，依据这些信息会进行slot分配
    all_host_names, _ = hosts.parse_hosts_and_slots(args.hosts)
    remote_host_names = network.filter_local_addresses(all_host_names)

    nics = driver_service.get_common_interfaces(settings, all_host_names,
                                                remote_host_names, fn_cache)

    if args.run_func:
        # get the driver IPv4 address
        driver_ip = network.get_driver_ip(nics)
        run_func_server = KVStoreServer(verbose=settings.verbose) # 启动内部KV服务
器
        run_func_server_port = run_func_server.start_server()
        put_data_into_kvstore(driver_ip, run_func_server_port,
                              'runfunc', 'func', args.run_func) # 把'func',
args.run_func存储成KV

        command = [sys.executable, '-m', 'horovod.runner.run_task',
str(driver_ip), str(run_func_server_port)]

        try:
            _launch_job(args, settings, nics, command)
            results = [None] * args.np
            for i in range(args.np):
                results[i] = read_data_from_kvstore(driver_ip,
run_func_server_port,'runfunc_result', str(i))
```

```
            return results
        finally:
            run_func_server.shutdown_server()
    else:
        command = args.command
        _launch_job(args, settings, nics, command) # 我们重点讲解这里
        return None
```

目前逻辑如下：



至此，我们已经分析完成 horovod 的入口，下面会分析具体如何启动 Job。

0x03 运行训练 Job

3.1 _launch_job

_launch_job 会根据配置或者安装情况来进行具体调用。我们看到有三种可能：gloo, mpi, js。

jsrun的资料很难找，所以我们重点看看 gloo, mpi 这两种。

```
def _launch_job(args, settings, nics, command):
    env = os.environ.copy()
    config_parser.set_env_from_args(env, args)

    def gloo_run_fn():
        driver_ip = network.get_driver_ip(nics)
        gloo_run(settings, nics, env, driver_ip, command)

    def mpi_run_fn():
        mpi_run(settings, nics, env, command)

    def js_run_fn():
        js_run(settings, nics, env, command)

    run_controller(args.use_gloo, gloo_run_fn,
                   args.use_mpi, mpi_run_fn,
                   args.use_jsrun, js_run_fn,
```

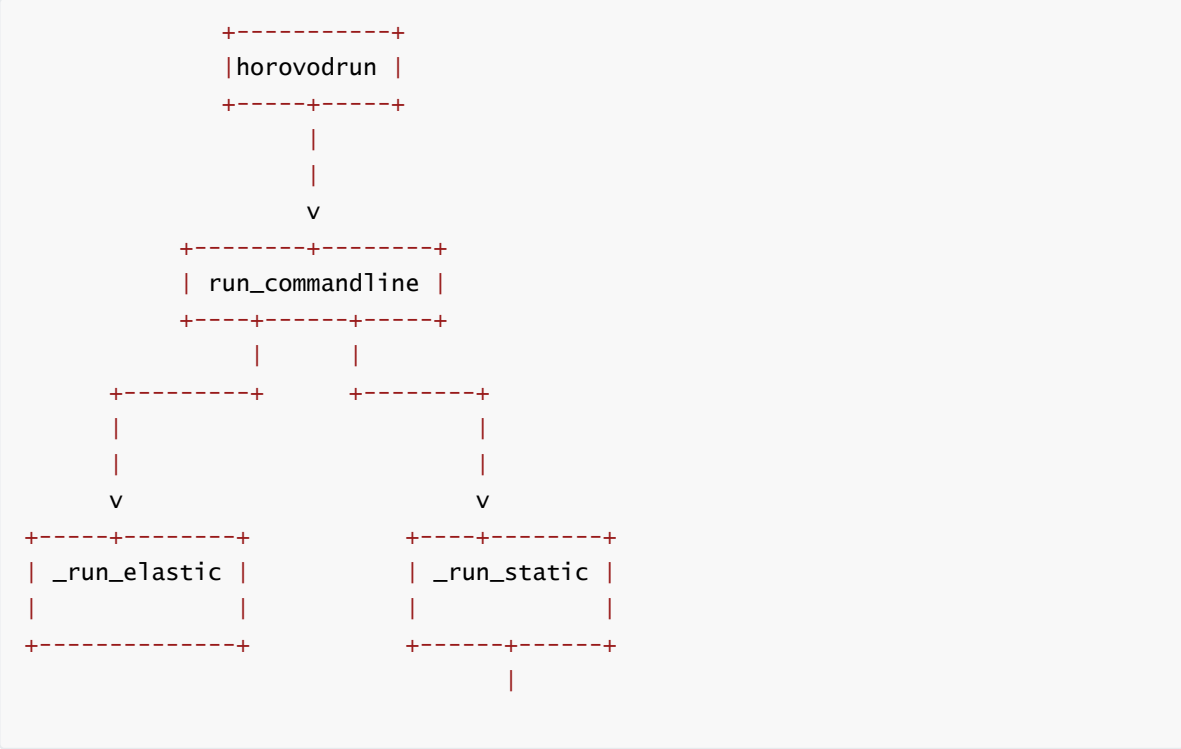
```
                    args.verbose)
```

3.2 run_controller

run_controller 依然是一个中介函数，具体导入 gloo 或者 mpi。

```
def run_controller(use_gloo, gloo_run, use_mpi, mpi_run, use_jsrun, js_run,
verbosity):
    if use_gloo:
        gloo_run()
    elif use_mpi:
        mpi_run()
    elif use_jsrun:
        js_run()
    else:
        if mpi_built(verbose=verbose):
            if lsf.LSFUtils.using_lsf() and is_jsrun_installed():
                js_run()
            else:
                mpi_run()
        elif gloo_built(verbose=verbose):
            gloo_run()
```

目前逻辑如下：



于是我们下面就分为两个分支介绍：gloo & mpi。

0x04 Gloo 实现

4.1 Gloo 简介

Gloo 是 facebook出品的一个类似MPI的集合通信库（https://github.com/facebookincubator/gloo）。

集合通信库的主要特征是：大体上会遵照 MPI 提供的接口规定，实现了包括点对点通信（SEND,RECV等），集合通信（ REDUCE，BROADCAST，ALLREDUCE等）等相关接口，然后根据自己硬件或者是系统的需要，在底层实现上进行了相应的改动，保证接口的稳定和性能。

Gloo 为CPU和GPU提供了集合通信程序的优化实现。 它特别适用于GPU，因为它可以执行通信而无需使用GPUDirect 将数据传输到CPU的内存。 它还能够使用 NCCL 执行快速的节点内通信，并实现其自己的节点间例程算。你不需要考虑内存数据的拷贝，只需要实现逻辑就可以。

Gloo 支持集体通信（collective Communication），并对其进行了优化。由于 GPU 之间可以直接进行数据交换，而无需经过 CPU 和内存，因此，在 GPU 上使用 gloo后端速度更快。

Horovod 为什么会选择 Gloo？个人认为除了其功能的全面性和性能之外，基于它可以二次开发是一个亮点，比如下面我们所说的 Rendezvous 功能就被 Horovod 用来实现弹性训练（我们后文有专门讲解）。

Gloo 和 MPI 都起到了同样类似作用：

- 一方面Horovod内集成了基于 Gloo 的AllReduce，类似于NCCL，都是用作梯度规约；
- 另一方面，Gloo 可以用来启动多个进程（Hovorod里用Rank表示），实现并行计算；

具体如下：

4.2 Rendezvous 功能

4.2.1 Rendezvous 概念

在 Gloo 的文档中，如此说:

```
The rendezvous process needs to happen exactly once per Gloo context.
It makes participating Gloo processes exchange details for setting up their
communication channels. For example, when the TCP transport is used, processes
exchange IP address and port number details of listening sockets.

Rendezvous can be executed by accessing a key/value store that is accessible by
all participating processes. Every process is responsible for setting a number
of keys and will wait until their peers have set their keys. The values stored
against these keys hold
the information that is passed to the transport layer.
```

大致意思是:

Gloo 在每一个 Gloo context 之中有一个 rendezvous process，Gloo 利用它来交换通讯需要的细节。

Rendezvous 具体实现是可以依靠访问一个 KVstore 来完成。具体细节就是通过 KVstore 来进行交互。

以 Horovod 为例:

- Horovod 在进行容错 AllReduce 训练时，除了启动 worker 进程外，还会启动一个 driver 进程。这个 driver 进程用于帮助 worker 调用 gloo 构造 AllReduce 通信环。
- driver 进程中会创建一个带有 KVStore 的 RendezvousServer，driver 会将参与通信的 worker 的 ip 等信息存入 KVstore 中。
- 然后 worker 就可以调用 gloo 来访问 RendezvousServer 构造通信环了。

4.2.2 RendezvousServer

具体代码如下，可以看到是启动了RendezvousHTTPServer（就是继承拓展了 HTTPServer）：

```
class RendezvousServer:
    def __init__(self, verbose=0):
        self._httpd = None
        self._listen_thread = None
        self._verbose = verbose

    # Rendezvous function finds a available port, create http socket,
    # and start listening loop to handle request
    # self.httpd.init needs to be called after server start
    def start(self, handler_cls=RendezvousHandler): # 下面马上介绍
        self._httpd, port = find_port(
            lambda addr: RendezvousHTTPServer(
                addr, handler_cls, self._verbose))

        # start the listening loop
        self._listen_thread = in_thread(target=self._httpd.serve_forever)

        return port

    def init(self, host_alloc_plan):
        self._httpd.init(host_alloc_plan)

    def stop(self):
        self._httpd.shutdown()
        self._listen_thread.join()
```

4.2.3 KVStore

KVStore 是由 KVStoreHandler 来体现，RendezvousHandler 继承了 KVStoreHandler，进而被 RendezvousServer 作为 handler 使用。

KVStoreHandler 精简版代码如下：

```
class KVStoreHandler(SimpleHTTPRequestHandler):

    # Override PUT handler
    def do_PUT(self):
        paths = self.path.split('/')
        _, scope, key = paths

        # Get body length
        content_length = int(self.headers['Content-Length'])
        value = self.rfile.read(content_length)
        self._put_value(scope, key, value)
        self.send_status_code(OK)

    def _put_value(self, scope, key, value):
        with self.server.cache_lock:
            scope_dict = self.server.cache.setdefault(scope, {})
            scope_dict[key] = value
```

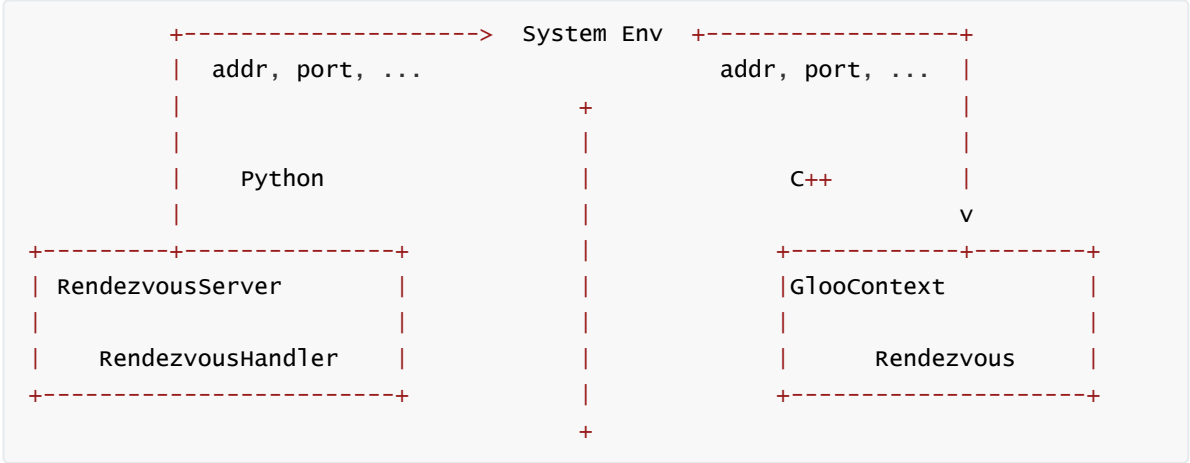
4.2.4 底层使用

Rendezvous 具体如何使用？简要的说：

- Python世界构建了一个 RendezvousServer，其地址配置在环境变量（或者其他方式）中。
- 在 C++ 世界中，比如 horovod/common/gloo/gloo_context.h，horovod/common/gloo/gloo_context.cc 之中有使用。即得到 Python 配置的 RendezvousServer 的地址端口等，然后构建 gloo 所需的 context。

```
#define HOROVOD_HOSTNAME "HOROVOD_HOSTNAME"
#define HOROVOD_RANK "HOROVOD_RANK"
#define HOROVOD_SIZE "HOROVOD_SIZE"
#define HOROVOD_LOCAL_RANK "HOROVOD_LOCAL_RANK"
#define HOROVOD_LOCAL_SIZE "HOROVOD_LOCAL_SIZE"
#define HOROVOD_CROSS_RANK "HOROVOD_CROSS_RANK"
#define HOROVOD_CROSS_SIZE "HOROVOD_CROSS_SIZE"
#define HOROVOD_ELASTIC "HOROVOD_ELASTIC"

  ctx = Rendezvous(HOROVOD_GLOO_GLOBAL_PREFIX,
                   rendezvous_addr_env, rendezvous_port,
                   rank, size, dev, timeout);

  local_ctx = Rendezvous(HOROVOD_GLOO_LOCAL_PREFIX + hostname,
                         rendezvous_addr_env, rendezvous_port,
                         local_rank, local_size, dev, timeout);

  cross_ctx = Rendezvous(HOROVOD_GLOO_CROSS_PREFIX + std::to_string(local_rank),
                         rendezvous_addr_env, rendezvous_port,
                         cross_rank, cross_size, dev, timeout);
```

逻辑如下，C++世界会从python世界的获取到RendezvousServer的 IP，port：



4.3 Horovd 的 gloo 入口

gloo_run 是 horovod 之中，gloo 模块的 相关入口。

注释说的很清楚：每一个 thread 将使用 ssh 命令在远程host之上启动训练job。

```
def gloo_run(settings, nics, env, server_ip, command):
    # Each thread will use ssh command to launch the job on each remote host. If an
    # error occurs in one thread, entire process will be terminated. Otherwise,
    # threads will keep running and ssh session.
    exec_command = _exec_command_fn(settings)
    launch_gloo(command, exec_command, settings, nics, env, server_ip)
```

就是用 launch_gloo 来运行 exec_command。

此时 command 参数类似 `"['python', 'train.py']"`。

4.4 构建可执行环境

gloo_run 的第一部分是 `exec_command = _exec_command_fn(settings)`，就是基于各种配置来生成可以执行命令环境。如果是远程，就得生成相关远程可运行命令环境（包括切换目录，远程执行等等）。

4.4.1 _exec_command_fn

具体又可以分为两部分：

- 利用 get_remote_command 来生成相关远程可运行环境，比如在训练脚本前面加上 `'ssh -o PasswordAuthentication=no -o StrictHostKeyChecking=no'`；
- 调整输入输出，利用 safe_shell_exec.execute 来实现安全执行能力；

具体如下：

```
def _exec_command_fn(settings):
    """
    executes the jobs defined by run command on hosts.
    :param hosts_alloc: list of dict indicating the allocating info.
    For example,
        [{'Hostname':'worker-0', 'Rank': 0, 'Local_rank': 0, 'Cross_rank':0,
            'Size':2, 'Local_size':1, 'Cross_size':2},
```

```
        {'Hostname':'worker-1', 'Rank': 1, 'Local_rank': 0, 'Cross_rank':1,
            'Size':2, 'Local_size':1, 'Cross_size':2}
        ]
    :type hosts_alloc: list(dict)
    :param remote_host_names: names that are resolved to one of the addresses
    of remote hosts interfaces.
    :param _run_command: command to execute
    """
    def _exec_command(command, slot_info, events):
        index = slot_info.rank
        host_name = slot_info.hostname
        host_address = network.resolve_host_address(host_name)
        local_addresses = network.get_local_host_addresses()
        # 需要构建远程命令
        if host_address not in local_addresses:
            local_command = quote('cd {pwd} > /dev/null 2>&1 ; {command}'
                                  .format(pwd=os.getcwd(), command=command))
            command = get_remote_command(local_command,
                                         host=host_name,
                                         port=settings.ssh_port,

identity_file=settings.ssh_identity_file)

        # Redirect output if requested
        # 调整输入输出，利用 safe_shell_exec.execute 来实现安全执行能力
        stdout = stderr = None
        stdout_file = stderr_file = None
        if settings.output_filename:
            padded_rank = _pad_rank(index, settings.num_proc)
            output_dir_rank = os.path.join(settings.output_filename, 'rank.
{rank}'.format(rank=padded_rank))
            if not os.path.exists(output_dir_rank):
                os.mkdir(output_dir_rank)

            stdout_file = open(os.path.join(output_dir_rank, 'stdout'), 'w')
            stderr_file = open(os.path.join(output_dir_rank, 'stderr'), 'w')

            stdout = MultiFile([sys.stdout, stdout_file])
            stderr = MultiFile([sys.stderr, stderr_file])

            # 实现安全执行能力
            exit_code = safe_shell_exec.execute(command,
                                                index=index,
                                                stdout=stdout,
                                                stderr=stderr,
                                                events=events,...)

        return exit_code, time.time()

    return _exec_command
```

4.4.2 get_remote_command

本函数是针对远程 host，获取如何在其上运行的方式。这个函数是比较新加入的，具体和 kubeflow mpi operator 也相关，以后有机会再分析。

```
SSH_COMMAND_PREFIX = 'ssh -o PasswordAuthentication=no -o
StrictHostKeyChecking=no'

def get_ssh_command(local_command, host, port=None, identity_file=None,
timeout_s=None):
    port_arg = f'-p {port}' if port is not None else ''
    identity_file_arg = f'-i {identity_file}' if identity_file is not None else
''
    timeout_arg = f'-o ConnectTimeout={timeout_s}' if timeout_s is not None else
''
    return f'{SSH_COMMAND_PREFIX} {host} {port_arg} {identity_file_arg}
{timeout_arg} {local_command}'


def get_remote_command(local_command, host, port=None, identity_file=None,
timeout_s=None):
    return f'{env_util.KUBEFLOW_MPI_EXEC} {host} {local_command}' if
env_util.is_kubeflow_mpi() \
        else get_ssh_command(local_command, host, port, identity_file,
timeout_s)
```

大致逻辑如下：

```
command   :  python train.py
             +
             |
             v
  +---------+-------------+
  |                       |
  |  get_remote_command   |
  |                       |
  +---------+-------------+
             |
             v
 ssh -o ... python train.py
             +
             |
             v
  +---------+--------------+
  |safe_shell_exec.execute |
  +------------------------+
```

4.5 使用 gloo 执行命令

获取到了可执行环境 exec_command 与 执行命令 command 之后，就可以使用 gloo 来执行命令了。

每个 command 都是被 exec_command 来执行。

launch_gloo 来获取命令，各种配置信息，网卡信息（nics，比如 {'lo'}），host信息等，然后开始运行，就是开始运行我们的训练代码了，具体是：

- 建立 RendezvousServer，这个会被底层 Gloo C++ 环境使用到；
- host_alloc_plan = get_host_assignments 来根据host进行分配slot，就是horovod的哪个rank应该在哪个host上的哪个slot之上运行；
- get_run_command 获取到可执行命令；
- slot_info_to_command_fn 来得到在slot之上可执行的 slot command；

- 依据 slot_info_to_command_fn 构建 args_list，这个 list 之中，每一个arg就是一个 slot command；
- 多线程执行，在每一个 exec_command 之上执行每一个 arg（slot command）；

代码如下：

```
def launch_gloo(command, exec_command, settings, nics, env, server_ip):
    """
    Launches the given command multiple times using gloo.
    Each command is launched via exec_command.

    :param command: command to launch
    :param exec_command: means to execute a single command
    :param settings: settings for the distribution
    :param nics: common interfaces
    :param env: environment to use
    :param server_ip: ip to use for rendezvous server
    """
    # Make the output directory if it does not exist
    if settings.output_filename:
        _mkdir_p(settings.output_filename)

    # start global rendezvous server and get port that it is listening on
    # 建立 RendezvousServer，这个会被底层 Gloo C++ 环境使用到
    rendezvous = RendezvousServer(settings.verbose)

    # allocate processes into slots
    # 来根据host进行分配slot，就是horovod的哪个rank应该在哪个host上的哪个slot之上运行
    hosts = parse_hosts(settings.hosts)
    host_alloc_plan = get_host_assignments(hosts, settings.num_proc)

    # start global rendezvous server and get port that it is listening on
    global_rendezv_port = rendezvous.start()
    rendezvous.init(host_alloc_plan)
    # 获取到可执行命令
    run_command = get_run_command(command, server_ip, nics, global_rendezv_port)

    # 得到在slot之上可执行的 slot command
    slot_info_to_command = _slot_info_to_command_fn(run_command, env)
    event = register_shutdown_event()
    # 依据 slot_info_to_command_fn 构建 args_list，这个 list 之中，每一个arg就是一个
slot command
    args_list = [[slot_info_to_command(slot_info), slot_info, [event]]
                 for slot_info in host_alloc_plan]

    # If an error occurs in one thread, entire process will be terminated.
    # Otherwise, threads will keep running.
    # 多线程执行，在每一个 exec_command 之上执行每一个 arg（slot command）
    res = threads.execute_function_multithreaded(exec_command,
                                                 args_list,
                                                 block_until_all_done=True)

    for name, value in sorted(res.items(), key=lambda item: item[1][1]):
        exit_code, timestamp = value
```

4.5.1 slot分配方案

上面提到了 Horovod 在 slot 之上执行任务，我们需要看看 slot 是如何分配的。

4.5.1.1 从输入参数解析

由下面代码可知，slot 是通过 parse_hosts 自动解析出来。

```
def parse_hosts(hosts_string):
    """Parse a string of comma-separated hostname:slots mappings into a list of
HostItem objects.
    :param hosts_string: list of addresses and number of processes on each host.
        For example:
            - 'worker-0:2,worker-1:2'
            - '10.11.11.11:4,10.11.11.12:4'
    :return: a list of HostInfo objects describing host to slot mappings
    :rtype: list[HostInfo]
    """
    return [HostInfo.from_string(host_string) for host_string in
hosts_string.split(',')]
```

具体 HostInfo.from_string 信息如下：

```
class HostInfo:
    def __init__(self, hostname, slots):
        self.hostname = hostname
        self.slots = slots
    @staticmethod
    def from_string(host_string):
        hostname, slots = host_string.strip().split(':')
        return HostInfo(hostname, int(slots))
```

4.5.1.2 分配方案

get_host_assignments 会依据 host 和 process capacities (slots) 来给 Horovod 之中的进程分配，即给出一个 horovod rank 和 slot 的对应关系。设置了几个 np，就有几个 slot。

给出的分配方案类似如下，这样就知道了哪个rank对应于哪个host上的哪个slot：

```
[
  SlotInfo(hostname='h1', rank=0, local_rank=0, cross_rank=0, size=2,
local_size=2, coress_size=1),
    SlotInfo(hostname='h2', rank=1, local_rank=0, cross_rank=0, size=2,
local_size=2, coress_size=1),
]
```

代码如下：

```
def get_host_assignments(hosts, min_np, max_np=None):
    """Assign hosts with process capacities (slots) to ranks in the Horovod
process.
    This function will try to allocate as many as possible processes on the same
host to leverage local network.
    :param hosts: list of HostInfo objects describing host and slot capacity
    :type hosts: list[HostInfo]
    :param min_np: minimum number of processes to be allocated
```

```
    :param max_np: (optional) maximum number of processes to be allocated
    :return: a list of the allocation of process on hosts in a `SlotInfo`
object.
    :rtype: list[SlotInfo]
    """
    host_ranks = []
    cross_ranks = collections.defaultdict(dict)
    rank = 0
    # 依据 hosts 信息构建 rank, local rank, cross rank(hierarchical allreduce所需要)
    for host_info in hosts:
        ranks = []
        for local_rank in range(host_info.slots):
            if rank == max_np:
                break

            ranks.append(rank)
            rank += 1

            cross_ranks_at_local = cross_ranks[local_rank]
            cross_ranks_at_local[host_info.hostname] = len(cross_ranks_at_local)

        host_ranks.append((host_info, ranks))

    world_size = rank

    # 给出一个 horovod rank 和 slot 的对应关系。返回一个alloc_list，每个SlotInfo包括各种
rank信息
    alloc_list = []
    for host_info, ranks in host_ranks:
        local_size = len(ranks)
        for local_rank, rank in enumerate(ranks):
            cross_ranks_at_local = cross_ranks[local_rank]
            cross_rank = cross_ranks_at_local[host_info.hostname]
            cross_size = len(cross_ranks_at_local)

            alloc_list.append(
                SlotInfo(
                    hostname=host_info.hostname,
                    rank=rank,
                    local_rank=local_rank,
                    cross_rank=cross_rank,
                    size=world_size,
                    local_size=local_size,
                    cross_size=cross_size))

    return alloc_list
```

4.5.2 得到运行命令

get_run_command 是从环境变量中得到 Gloo 的变量，然后加到 command 之上。此步完成之后，得到类似如下命令：

```
HOROVOD_GLOO_RENDEZVOUS_ADDR=1.1.1.1 HOROVOD_GLOO_RENDEZVOUS_PORT=2222
HOROVOD_CPU_OPERATIONS=gloo HOROVOD_GLOO_IFACE=lo HOROVOD_CONTROLLER=gloo python
train.py
```

可以把这个格式缩写为：{horovod_gloo_env} command。代码为：

```
def create_run_env_vars(server_ip, nics, port, elastic=False):
    # 从环境变量中得到 Gloo 的变量
    run_envs = {
        'HOROVOD_GLOO_RENDEZVOUS_ADDR': server_ip,
        'HOROVOD_GLOO_RENDEZVOUS_PORT': port,
        'HOROVOD_CONTROLLER': "gloo",
        'HOROVOD_CPU_OPERATIONS': "gloo",
        'HOROVOD_GLOO_IFACE': list(nics)[0],    # TODO: add multiple ifaces in
future
        'NCCL_SOCKET_IFNAME': ','.join(nics),
    }
    if elastic:
        run_envs["HOROVOD_ELASTIC"] = "1"
    return run_envs

def get_run_command(command, server_ip, nics, port, elastic=False):
    env_vars = create_run_env_vars(server_ip, nics, port, elastic)
    env_string = " ".join(
        [f"{k}={str(v)}" for k, v in env_vars.items()])
    run_command = (
        '{env_string} '
        '{command}'  # expect a lot of environment variables
        .format(env_string=env_string,
                command=' '.join(quote(par) for par in command)))
    return run_command
```

4.5.3 得到slot运行命令

得到运行命令之后，这里会结合 horovod env 和 env，以及slot 分配情况 进一步修改为适合 gloo 运行的方式。就是可以在具体每一个slot上运行的命令。

可以把这个格式缩写为：{horovod_gloo_env} {horovod_rendez_env} {env} run_command。

此步完成之后，得到类似如下：

```
HOROVOD_HOSTNAME=1.1.1.1 HOROVOD_RANK=1 HOROVOD_SIZE=2 HOROVOD_LOCAL_RANK=1
SHELL=/bin/bash PATH=XXXX USER=xxx PWD=xxx SSH_CONNECTION="1.1.1.1 11 2.2.2.2
22" HOME=xxx SSH_CLIENZT=xxxx
HOROVOD_GLOO_IFACE=lo NCCL_SOCKET_IFNAME=lo
HOROVOD_GLOO_RENDEZVOUS_ADDR=1.1.1.1 HOROVOD_GLOO_RENDEZVOUS_PORT=2222
HOROVOD_CPU_OPERATIONS=gloo HOROVOD_GLOO_IFACE=lo HOROVOD_CONTROLLER=gloo python
train.py
```

具体代码如下：

```
def _slot_info_to_command_fn(run_command, env):
    # TODO: Workaround for over-buffered outputs. Investigate how mpirun avoids
this problem.
    env = copy.copy(env)  # copy env so we do not leak env modifications
    env['PYTHONUNBUFFERED'] = '1'

    def slot_info_to_command(slot_info):
        """
        Given a slot_info, creates a command used by gloo to launch a single
job.
```

```
        :param slot_info: host and slot to execute the run command on
        :return:
        """
        env_vars = create_slot_env_vars(slot_info)
        horovod_rendez_env = " ".join(
            [f"{k}={str(v)}" for k, v in env_vars.items()])

        return '{horovod_env} {env} {run_command}' .format(
            horovod_env=horovod_rendez_env,
            env=' '.join(['%s=%s' % (key, quote(value)) for key, value in
env.items()
                          if env_util.is_exportable(key)]),
            run_command=run_command)

    return slot_info_to_command
```

4.5.4 多线程调用命令

这就是启动了多线程进行调用。gloo_run 的注释说的很清楚：在调用 execute_function_multithreaded 时，每一个thread将使用 ssh 命令在远程host之上启动训练job。

回忆下之前我们在“构建可执行环境” 中提到：利用 get_remote_command 来生成相关远程可运行环境，比如在训练脚本前面加上 `'ssh -o PasswordAuthentication=no -o StrictHostKeyChecking=no'`。大家就理解了如何在远端执行。

在**本地运行**，则命令大致为：

```
cd /code directory > /dev/null 2 >&1
HOROVOD_HOSTNAME=1.1.1.1 HOROVOD_RANK=1 HOROVOD_SIZE=2 HOROVOD_LOCAL_RANK=1
SHELL=/bin/bash PATH=XXXX USER=xxx PWD=xxx SSH_CONNECTION="1.1.1.1 11 2.2.2.2
22" HOME=xxx SSH_CLIENZT=xxxx
HOROVOD_GLOO_IFACE=lo NCCL_SOCKET_IFNAME=lo
HOROVOD_GLOO_RENDEZVOUS_ADDR=1.1.1.1 HOROVOD_GLOO_RENDEZVOUS_PORT=2222
HOROVOD_CPU_OPERATIONS=gloo HOROVOD_GLOO_IFACE=lo HOROVOD_CONTROLLER=gloo python
train.py
```

在**远端运行**，命令就需要加上 ssh 信息，大致为：

```
ssh -o PasswordAuthentication=no -o StrictHostKeyChecking=no 1.1.1.1
cd /code directory > /dev/null 2 >&1
HOROVOD_HOSTNAME=1.1.1.1 HOROVOD_RANK=1 HOROVOD_SIZE=2 HOROVOD_LOCAL_RANK=1
SHELL=/bin/bash PATH=XXXX USER=xxx PWD=xxx SSH_CONNECTION="1.1.1.1 11 2.2.2.2
22" HOME=xxx SSH_CLIENZT=xxxx
HOROVOD_GLOO_IFACE=lo NCCL_SOCKET_IFNAME=lo
HOROVOD_GLOO_RENDEZVOUS_ADDR=1.1.1.1 HOROVOD_GLOO_RENDEZVOUS_PORT=2222
HOROVOD_CPU_OPERATIONS=gloo HOROVOD_GLOO_IFACE=lo HOROVOD_CONTROLLER=gloo python
train.py
```

execute_function_multithreaded 具体代码如下，其中：

- `fn` 就是前面提到的程序运行环境（能力）`exec_command`。
- `fn(*arg[:-1])` 就是在 `exec_command` 之中运行 `slot_info_to_command`。

```
def execute_function_multithreaded(fn,
                                   args_list,
                                   block_until_all_done=True,
```

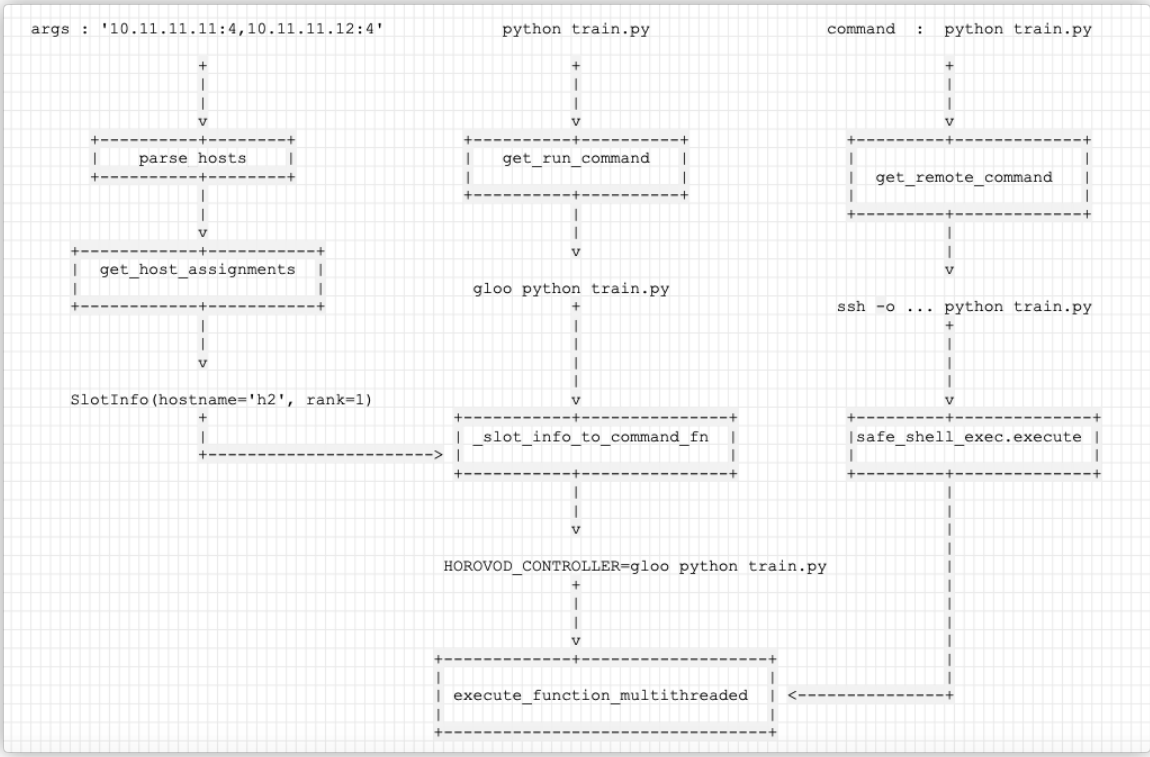
```
                                    max_concurrent_executions=1000):
    """
    Executes fn in multiple threads each with one set of the args in the
    args_list.
    :param fn: function to be executed
    :type fn:
    :param args_list:
    :type args_list: list(list)
    :param block_until_all_done: if is True, function will block until all the
    threads are done and will return the results of each thread's execution.
    :type block_until_all_done: bool
    :param max_concurrent_executions:
    :type max_concurrent_executions: int
    :return:
    If block_until_all_done is False, returns None. If block_until_all_done is
    True, function returns the dict of results.
        {
            index: execution result of fn with args_list[index]
        }
    :rtype: dict
    """
    result_queue = queue.Queue()
    worker_queue = queue.Queue()

    for i, arg in enumerate(args_list):
        arg.append(i)
        worker_queue.put(arg)

    def fn_execute():
        while True:
            try:
                arg = worker_queue.get(block=False)
            except queue.Empty:
                return
            exec_index = arg[-1]
            # fn 就是前面提到的程序运行环境（能力）exec_command
            # fn(*arg[:-1])是在 exec_command 之中运行 slot_info_to_command
            res = fn(*arg[:-1])
            result_queue.put((exec_index, res))

    threads = []
    number_of_threads = min(max_concurrent_executions, len(args_list))

    # 在多线程中执行 fn_execute
    for _ in range(number_of_threads):
        thread = in_thread(target=fn_execute, daemon=not block_until_all_done)
        threads.append(thread)

    # Returns the results only if block_until_all_done is set.
    # 如果有设置，则 block 等待
    results = None
    if block_until_all_done:
        # Because join() cannot be interrupted by signal, a single join()
        # needs to be separated into join()s with timeout in a while loop.
        have_alive_child = True
        while have_alive_child:
            have_alive_child = False
            for t in threads:
```

```
                t.join(0.1)
                if t.is_alive():
                    have_alive_child = True
        results = {}
        while not result_queue.empty():
            item = result_queue.get()
            results[item[0]] = item[1]

    return results
```

python train.py 就会进入到我们的训练代码。

大致逻辑如下图，可以看到，结合了各种信息之后，构建了一个可以执行的结果，然后多host执行：

- 图左面，是从 参数中获取 host 等信息，然后解析出 slot 信息；
- 图右边，是从 python train.py 这个待运行的命令，基于各种配置来生成可以执行命令环境。如果是远程，就得生成 相关远程可运行命令环境（包括切换目录，远程执行等等）；
- 图中间，是从 python train.py 这个待运行的命令，经过添加 env 信息，gloo 信息。然后结合 左面的 slot 信息 和 右面 的可以执行命令环境 之后，得到了可以在多线程上运行，从而在 多slot 运行的命令。



4.6 C++举例

我们给出一个底层代码，大家就进一步了解 Gloo 可以起到什么作用。

这个就是 Horovod 之中，rank 0 最终给其他 rank 发送构建好的 Tensor。

```
void GlooController::SendFinalTensors(ResponseList& response_list) {
  // Notify all nodes which tensors we'd like to reduce at this step.
  std::string encoded_response;
  ResponseList::SerializeToString(response_list, encoded_response);

  // Boardcast the response length
  int encoded_response_length = (int)encoded_response.length() + 1;
  {
```

```
    gloo::BroadcastOptions opts(gloo_context_.ctx);
    opts.setOutput(&encoded_response_length, 1);
    opts.setRoot(RANK_ZERO);
    gloo::broadcast(opts); // 广播给其他rank
  }

  // Boardcast the response
  {
    gloo::BroadcastOptions opts(gloo_context_.ctx);
    opts.setOutput((uint8_t*)(encoded_response.c_str()),
                   encoded_response_length);
    opts.setRoot(RANK_ZERO);
    gloo::broadcast(opts); // 广播给其他rank
  }
}
```

0x05 Mpi 实现

5.1 openmpi 库

horovod 这里主要依赖 openmpi。

- MPI：英文全称是Message Passing Interface，MPI是一个跨语言的通讯协议，用于编写并行计算机。支持点对点和广播。MPI是一个信息传递应用程序接口，包括协议和和语义说明，他们指明其如何在各种实现中发挥其特性。MPI的目标是高性能，大规模性，和可移植性。
- openMPI：英文全称是open Message Passing Interface。openMPI是MPI的一种实现，一种库项目。

MPI在Hovorod的角色比较特殊：

- 一方面Horovod内集成了基于MPI的AllReduce，类似于NCCL，都是用作梯度规约；
- 另一方面，MPI可以用来在所有机器上启动多个进程(Hovorod里用Rank表示)，实现并行计算；

5.2 mpi_run 函数

此部分代码位于：horovod/runner/mpi_run.py。

首先摘录其**关键代码**如下，可以看出来其核心是运行 mpirun 命令。

```
# 我是下面大段代码中的关键代码!
mpirun_command = (
        'mpirun {basic_args} '
        '-np {num_proc}{ppn_arg}{hosts_arg} '
        '{binding_args} '
        '{mpi_args} '
        '{mpi_ssh_args} '
        '{tcp_intf_arg} '
        '{nccl_socket_intf_arg} '
        '{output_filename_arg} '
        '{env} {extra_mpi_args} {command}'
        .format(basic_args=basic_args,
                num_proc=settings.num_proc,
                ppn_arg=ppn_arg,
                hosts_arg=hosts_arg,
                binding_args=binding_args,
                mpi_args=' '.join(mpi_impl_flags),
                tcp_intf_arg=tcp_intf_arg,
```

```
                nccl_socket_intf_arg=nccl_socket_intf_arg,
                mpi_ssh_args=mpi_ssh_args,
                output_filename_arg=' '.join(output),
                env=env_list,
                extra_mpi_args=settings.extra_mpi_args if
settings.extra_mpi_args else '',
                command=' '.join(quote(par) for par in command))
    )

    # Execute the mpirun command.
    if settings.run_func_mode:
        exit_code = safe_shell_exec.execute(mpirun_command, env=env,
stdout=stdout, stderr=stderr)
    else:
        os.execve('/bin/sh', ['/bin/sh', '-c', mpirun_command], env)
```

就是依据各种配置以及参数来构建 mpirun 命令的所有参数，比如 ssh 的参数，mpi 参数，nccl 参数等等。

最后得到的 mpirun 命令举例如下：

```
mpirun --allow-run-as-root --np 2 -bind-to none -map-by slot \
    -x NCCL_DEBUG=INFO -x LD_LIBRARY_PATH -x PATH \
    -mca pml ob1 -mca btl ^openib \
    python train.py
```

具体代码如下，具体是：

```
# 上面代码是我之中的片段
def mpi_run(settings, nics, env, command, stdout=None, stderr=None):
    """
    Runs mpi_run.

    Args:
        settings: Settings for running MPI.
                  Note: settings.num_proc and settings.hosts must not be None.
        nics: Interfaces to include by MPI.
        env: Environment dictionary to use for running command.
        command: Command and arguments to run as a list of string.
        stdout: Stdout of the mpi process.
                Only used when settings.run_func_mode is True.
        stderr: Stderr of the mpi process.
                Only used when settings.run_func_mode is True.
    """
    # 得到各种配置
    mpi_impl_flags, impl_binding_args, mpi =
_get_mpi_implementation_flags(settings.tcp_flag, env=env)
    impi = _IMPI_IMPL == mpi

    # 处理ssh参数
    ssh_args = []
    if settings.ssh_port:
        ssh_args += [f'-p {settings.ssh_port}']
    if settings.ssh_identity_file:
        ssh_args += [f'-i {settings.ssh_identity_file}']
```

```
    mpi_ssh_args = ''
    if ssh_args:
        joined_ssh_args = ' '.join(ssh_args)
        mpi_ssh_args = f'-bootstrap=ssh -bootstrap-exec-args \"
{joined_ssh_args}\"' if impi else f'-mca plm_rsh_args \"{joined_ssh_args}\"'

    # 处理网络配置，网卡信息等
    tcp_intf_arg = '-mca btl_tcp_if_include {nics}'.format(
        nics=','.join(nics)) if nics and not impi else ''
    nccl_socket_intf_arg = '-{opt} NCCL_SOCKET_IFNAME={nics}'.format(
        opt='genv' if impi else 'x',
        nics=','.join(nics)) if nics else ''

    # 处理host信息
    # On large cluster runs (e.g. Summit), we need extra settings to work around
OpenMPI issues
    host_names, host_to_slots = hosts.parse_hosts_and_slots(settings.hosts)
    if not impi and host_names and len(host_names) >= _LARGE_CLUSTER_THRESHOLD:
        mpi_impl_flags.append('-mca plm_rsh_no_tree_spawn true')
        mpi_impl_flags.append('-mca plm_rsh_num_concurrent
{}'.format(len(host_names)))

    # if user does not specify any hosts, mpirun by default uses local host.
    # There is no need to specify localhost.
    hosts_arg = '-{opt} {hosts}'.format(opt='hosts' if impi else 'H',
                hosts=','.join(host_names) if host_names and impi else
settings.hosts)

    # 处理ppn配置
    ppn_arg = ' '
    if host_to_slots and impi:
        ppn = host_to_slots[host_names[0]]
        for h_name in host_names[1:]:
        ppn_arg = ' -ppn {} '.format(ppn)

    # 处理超时配置
    if settings.prefix_output_with_timestamp and not impi:
        mpi_impl_flags.append('--timestamp-output')

    binding_args = settings.binding_args if settings.binding_args and not impi
else ' '.join(impl_binding_args)

    # 配置需要root身份运行
    basic_args = '-l' if impi else '--allow-run-as-root --tag-output'

    output = []
    if settings.output_filename:
        output.append('-outfile-pattern' if impi else '--output-filename')
        output.append(settings.output_filename)

    # 构建环境信息列表
    env_list = '' if impi else ' '.join(
                    '-x %s' % key for key in sorted(env.keys()) if
env_util.is_exportable(key))

    # 构建最终的 MPI 命令
    # Pass all the env variables to the mpirun command.
    mpirun_command = (
```

```
        'mpirun {basic_args} '
        '-np {num_proc}{ppn_arg}{hosts_arg} '
        '{binding_args} '
        '{mpi_args} '
        '{mpi_ssh_args} '
        '{tcp_intf_arg} '
        '{nccl_socket_intf_arg} '
        '{output_filename_arg} '
        '{env} {extra_mpi_args} {command}'  # expect a lot of environment
variables
        .format(basic_args=basic_args,
                num_proc=settings.num_proc,
                ppn_arg=ppn_arg,
                hosts_arg=hosts_arg,
                binding_args=binding_args,
                mpi_args=' '.join(mpi_impl_flags),
                tcp_intf_arg=tcp_intf_arg,
                nccl_socket_intf_arg=nccl_socket_intf_arg,
                mpi_ssh_args=mpi_ssh_args,
                output_filename_arg=' '.join(output),
                env=env_list,
                extra_mpi_args=settings.extra_mpi_args if
settings.extra_mpi_args else '',
                command=' '.join(quote(par) for par in command))
    )

    # we need the driver's PATH and PYTHONPATH in env to run mpirun,
    # env for mpirun is different to env encoded in mpirun_command
    for var in ['PATH', 'PYTHONPATH']:
        if var not in env and var in os.environ:
            # copy env so we do not leak env modifications
            env = copy.copy(env)
            # copy var over from os.environ
            env[var] = os.environ[var]

    # Execute the mpirun command.
    if settings.run_func_mode:
        exit_code = safe_shell_exec.execute(mpirun_command, env=env,
stdout=stdout, stderr=stderr)
    else:
        os.execve('/bin/sh', ['/bin/sh', '-c', mpirun_command], env)
```

5.3 mpirun命令

因为 mpi_run 使用的是 mpirun 命令来运行，所以我们介绍一下。

mpirun是MPI程序的启动脚本，它简化了并行进程的启动过程，尽可能屏蔽了底层的实现细节，从而为用户提供了一个通用的MPI并行机制。

在用mpirun命令执行并行程序时，参数-np指明了需要并行运行的进程个数。mpirun首先在本地结点上启动一个进程，然后根据/usr/local/share/machines.LINUX文件中所列出的主机，为每个主机启动一个进程。若进程数比可用的并行节点数多，则多余的进程将重新按照上述规则进行。按这个机制分配好进程后，一般会给每个节点分一个固定的标号，类似于身份证了，后续在消息传递中会用到。

这里需要说明的是，实际运行的

orterun(Open MPI SPMD / MPMD启动器; mpirun / mpiexec只是它的符号链接)

命令举例如下：

```
mpirun -np 4 \
    -bind-to none -map-by slot \
    -x NCCL_DEBUG=INFO -x LD_LIBRARY_PATH -x PATH \
    -mca pml ob1 -mca btl ^openib \
    python train.py
```

0x06 总结

对比 gloo 和 mpi 的实现，我们还是能看出来区别。

6.1 gloo

gloo 只是一个库，需要 horovod 来完成命令分发功能。

gloo 需要 horovod 自己实现本地运行和远端运行方式，即 get_remote_command 函数 实现 `'ssh -o PasswordAuthentication=no -o StrictHostKeyChecking=no'`。

gloo 需要实现 RendezvousServer，底层会利用 RendezvousServer 进行通讯。

6.2 mpi

mpi 则功能强大很多，只要把命令配置成被 mpirun 包装，openmpi 就可以自行完成命令分发执行。说到底，horovod 是一个 mpirun 程序，即使运行了 tensor flow，也是一个mpi程序，可以互相交互。