CANN

集合通信用户指南

文档版本 01

发布日期 2025-02-08

华为技术有限公司

商标声明


和其他华为商标均为华为技术有限公司的商标。
本文档提及的其他所有商标或注册商标，由各自的所有人拥有。

注意

您购买的产品、服务或特性等应受华为公司商业合同和条款的约束，本文档中描述的全部或部分产品、服务或特性可能不在您的购买或使用范围之内。除非合同另有约定，华为公司对本文档内容不做任何明示或暗示的声明或保证。

由于产品版本升级或其他原因，本文档内容会不定期进行更新。除非另有约定，本文档仅作为使用指导，本文档中的所有陈述、信息和建议不构成任何明示或暗示的担保。

安全声明

产品生命周期政策

华为公司对产品生命周期的规定以“产品生命周期终止政策”为准，该政策的详细内容请参见如下网址：
https://support.huawei.com/ecolumnsweb/zh/warranty-policy

漏洞处理流程

华为公司对产品漏洞管理的规定以“漏洞处理流程”为准，该流程的详细内容请参见如下网址：
https://www.huawei.com/cn/psirt/vul-response-process
如企业客户须获取漏洞信息，请参见如下网址：
https://securitybulletin.huawei.com/enterprise/cn/security-advisory

华为初始证书权责说明

华为公司对随设备出厂的初始数字证书，发布了“华为设备初始数字证书权责说明”，该说明的详细内容请参见如下网址：
https://support.huawei.com/enterprise/zh/bulletins-service/ENEWS2000015766

华为企业业务最终用户许可协议(EULA)

本最终用户许可协议是最终用户（个人、公司或其他任何实体）与华为公司就华为软件的使用所缔结的协议。最终用户对华为软件的使用受本协议约束，该协议的详细内容请参见如下网址：
https://e.huawei.com/cn/about/eula

产品资料生命周期策略

华为公司针对随产品版本发布的售后客户资料（产品资料），发布了“产品资料生命周期策略”，该策略的详细内容请参见如下网址：
https://support.huawei.com/enterprise/zh/bulletins-website/ENEWS2000017760

目 录

1 HCCL 概述

集合通信库HCCL（Huawei Collective Communication Library）是基于昇腾AI处理器的高性能集合通信库，提供单机多卡以及多机多卡间的数据并行、模型并行集合通信方案。

HCCL支持AllReduce、Broadcast、AllGather、ReduceScatter、AlltoAll等通信原语，Ring、Mesh、Halving-Doubling（HD）等通信算法，基于HCCS、RoCE和PCIe高速链路实现集合通信。

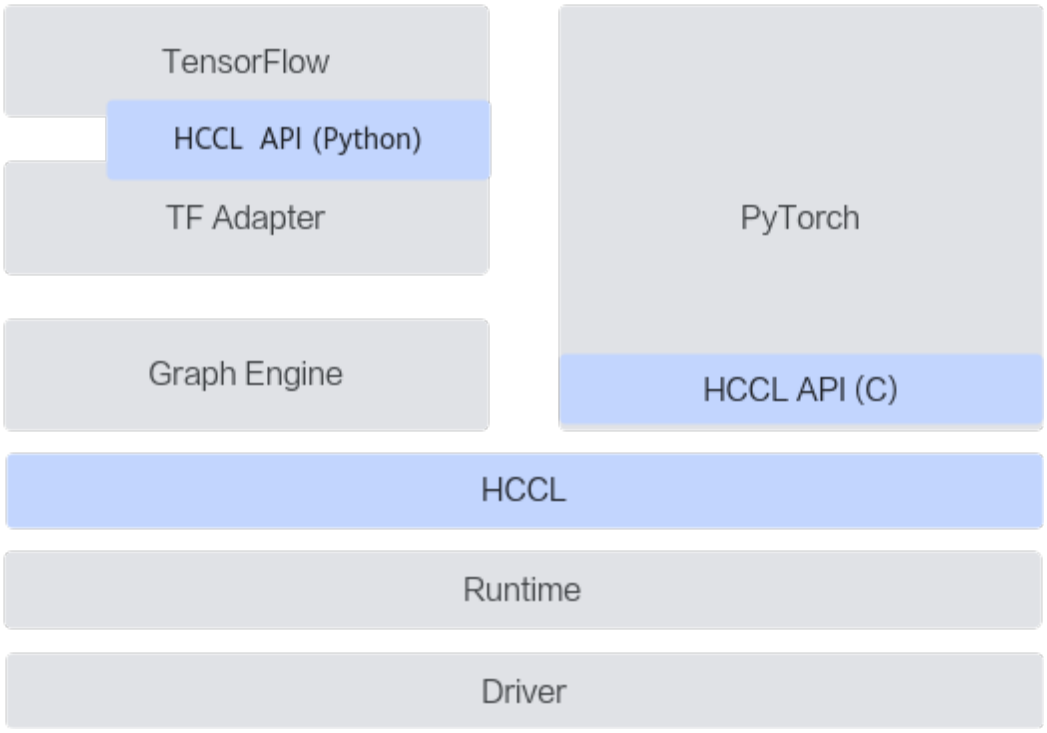
支持的产品型号

Atlas 训练系列产品

Atlas A2 训练系列产品

Atlas 300I Duo 推理卡

HCCL 在系统中的位置

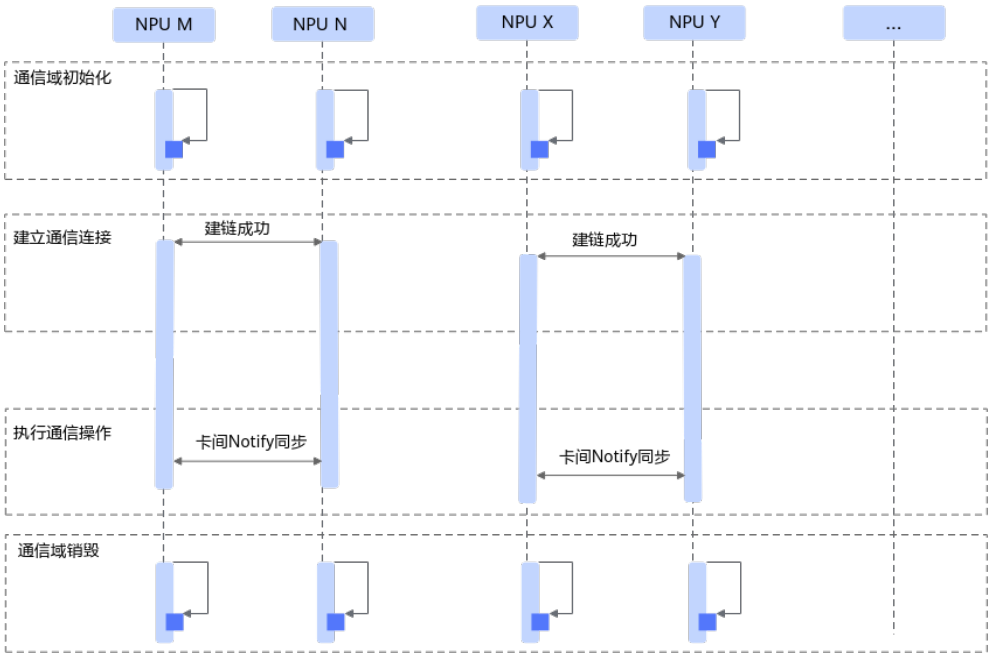


HCCL提供了Python与C两种语言的接口，其中Python语言的接口用于实现TensorFlow网络在昇腾AI处理器执行分布式优化；C语言接口用于实现单算子模式下的框架适配，实现分布式能力，例如HCCL单算子API嵌入到PyTorch后端代码中，PyTorch用户直接使用PyTorch原生集合通信API，即可实现分布式能力。

HCCL 执行流程

分布式场景中，HCCL提供了服务器间高性能集合通信功能，其执行流程如图1-1所示。

图 1-1 分布式场景集合通信流程



服务器间通信过程分为四个阶段：

1. 通信域初始化：获取必要的集合通信配置参数并初始化通信域。

   通信初始化阶段不涉及NPU设备之间的交互。

2. 建立通信连接：建立socket连接并交换通信两端的通信参数和内存信息。

   建立通信连接阶段，HCCL根据用户提供的集群信息并结合网络拓扑与其他NPU设备建链，交换用于通信的参数信息。如果在建链超时时间内未得到其他NPU设备的及时响应，会上报建链超时错误并退出业务进程。

3. 执行通信操作：通过“等待-通知”机制同步设备执行状态，传递内存数据。

   通信操作执行阶段，HCCL会将通信算法编排、内存访问等任务通过Runtime下发给昇腾设备的任务调度器，设备根据编排信息调度并执行任务。

4. 通信域销毁：销毁通信域，释放通信资源。

2 术语与相关概念

为了您有更好的阅读体验，使用本文档前请先了解如下术语、缩略语与基本概念。

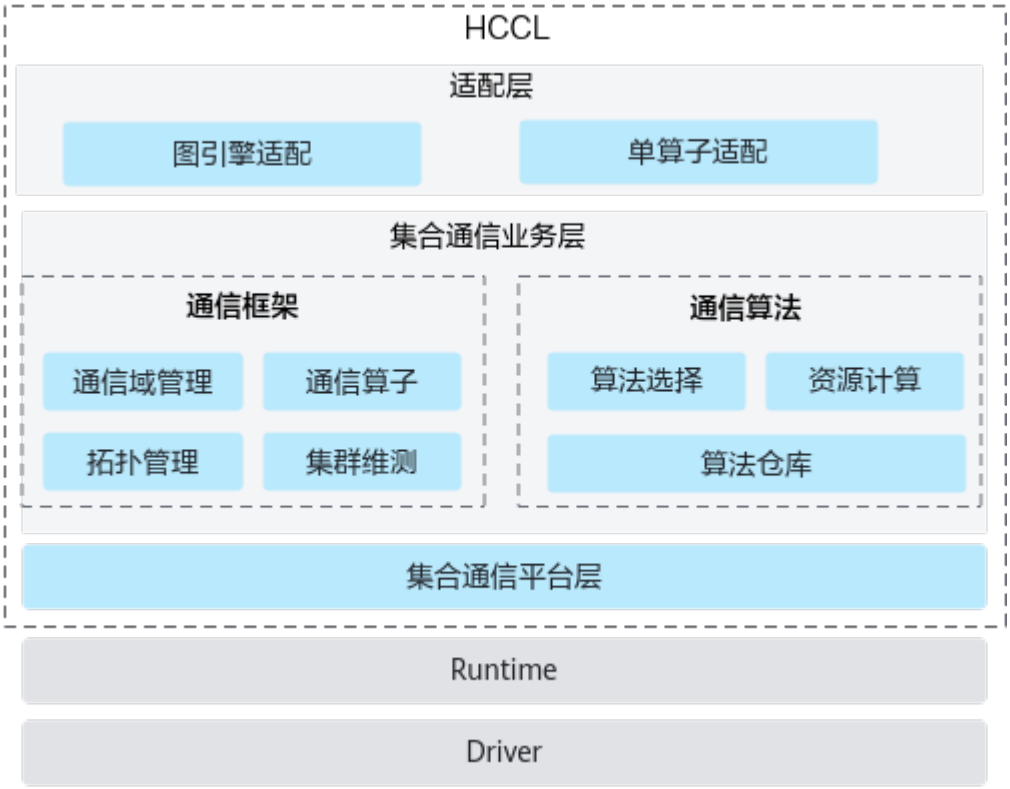
表 2-1 术语与相关概念

名称	说明
NPU	Neural Network Processing Unit，神经网络处理单元。 采用“数据驱动并行计算”的架构，特别擅长处理视频、图像类的海量多媒体业务数据，专门用于处理人工智能应用中的大量计算任务。
HCCL	Huawei Collective Communication Library，华为集合通信库。 提供单机多卡以及多机多卡间的数据并行、模型并行集合通信方案。
HCCS	Huawei Cache Coherence System，华为缓存一致性系统。 HCCS用于CPU/NPU之间的高速互联。
HCCP	Huawei Collective Communication adaptive Protocol，集合通信适配协议。 提供跨NPU设备通信能力，向上屏蔽具体通讯协议差异。
TOPO	拓扑、拓扑结构。 一个局域网内或者多个局域网之间的设备连接所构成的网络配置或者布置。
PCIe	Peripheral Component Interconnect Express，一种串行外设扩展总线标准，通常用于计算机系统中外设扩展使用。
PCIe-SW	PCIe Switch，符合PCIe总线扩展的交换设备。
SDMA	System Direct Memory Access，系统直接内存访问技术，简称DMA，允许外围设备直接访问系统内存，而不需要CPU的干预。
RDMA	Remote Direct Memory Access，远程直接内存访问技术，它将数据直接从一台机器的内存传输到另一台机器，无需双方操作系统的介入，一般指可以跨过网络的内存访问方式。
RoCE	RDMA over Converged Ethernet，承载在融合以太网上的RDMA技术，即跨越以太网的RDMA通信方式。

名称	说明
AI节点	昇腾AI节点，又称昇腾AI Server，通常是8卡或16卡的昇腾NPU设备组成的服务器形态的统称。
AI集群	多个AI节点通过交换机（Switch）互联后用于分布式训练或推理的系统。
通信域	通信域是集合通信执行的上下文，管理对应的通信实体（例如一个NPU就是一个通信实体）和通信所需的资源。
Rank	通信域中的每个通信实体称为一个Rank。

3 HCCL 软件架构

图 3-1 集合通信库软件架构图



集合通信库软件架构分为三层：

- 适配层，图引擎与单算子适配，进行通信切分寻优等操作。
- 集合通信业务层，包含通信框架与通信算法两个模块：
  - 通信框架：负责通信域管理，通信算子的业务串联，协同通信算法模块完成算法选择，协同通信平台模块完成资源申请并实现集合通信任务的下发。
  - 通信算法：作为集合通信算法的承载模块，提供特定集合通信操作的资源计算，并根据通信域信息完成通信任务编排。
- 集合通信平台层，提供NPU之上与集合通信关联的资源抽象，并提供集合通信的相关维护、测试能力。

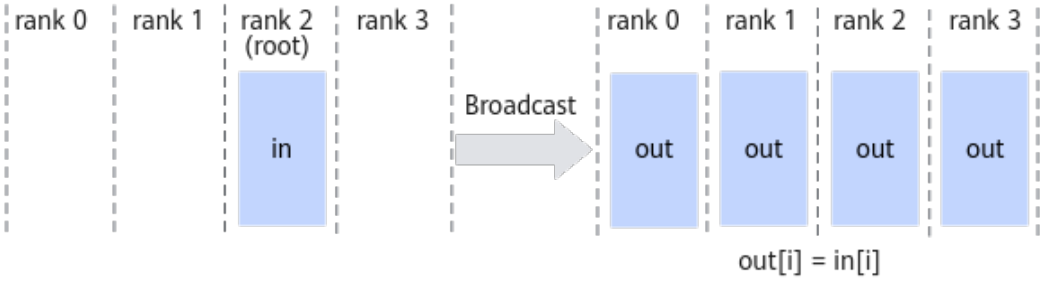
4 集合通信原语

集合通信是一个进程组的所有进程都参与的全局通信操作，其最为基础的操作有发送、接收、复制、节点间进程同步等，这些基本的操作经过组合构成了一组通信模板，也称为通信原语，这些通信原语通过相应的集合通信算子实现。

下面展开介绍HCCL提供的通信原语。

Broadcast

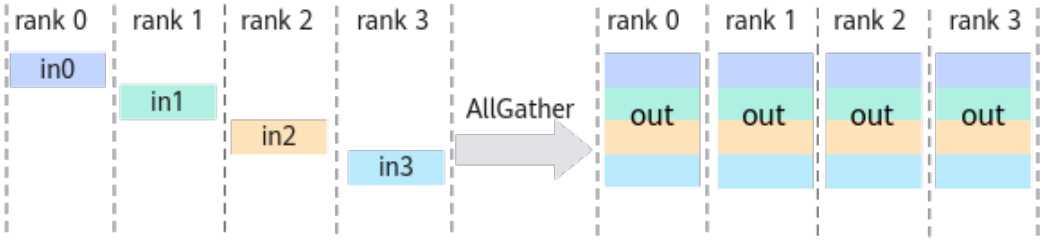
Broadcast操作是将通信域内root节点的数据广播到其他rank。



注意：全局只能有一个root节点。

AllGather

AllGather操作是将group内所有节点的输入按照rank id重新排序，然后拼接起来，再将结果发送到所有节点的输出。

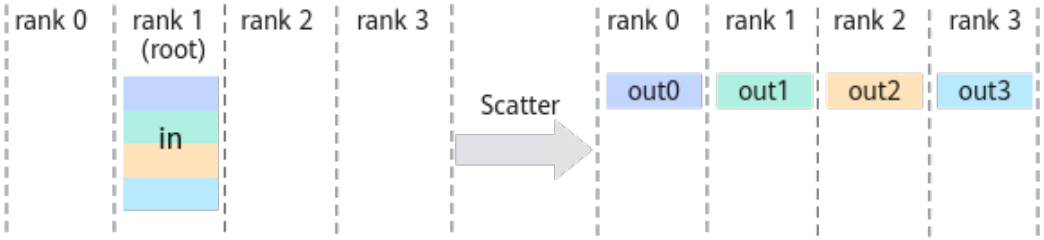


说明

针对AllGather操作，每个节点都接收按照rank id重新排序后的数据集合，即每个节点的AllGather输出都是一样的。

Scatter

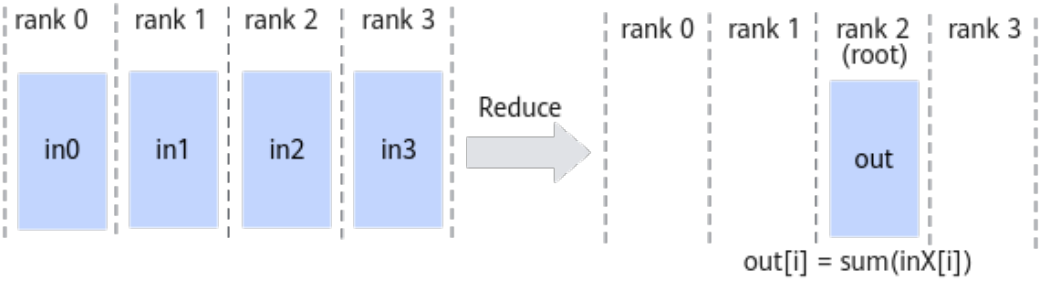
Scatter操作是将通信域内root节点的数据均分并散布至其他rank。



注意：全局只能有一个root节点。

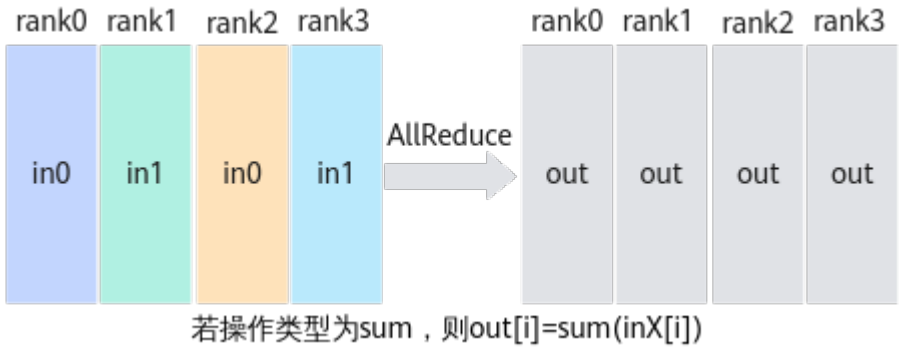
Reduce

Reduce操作是将所有rank的input相加（或prod/max/min）后，再把结果发送到root节点的输出buffer。



AllReduce

AllReduce操作是将通信域内所有节点的输入数据进行reduce操作后，再把结果发送到所有节点的输出buffer，其中reduce操作类型支持sum、prod、max、min。

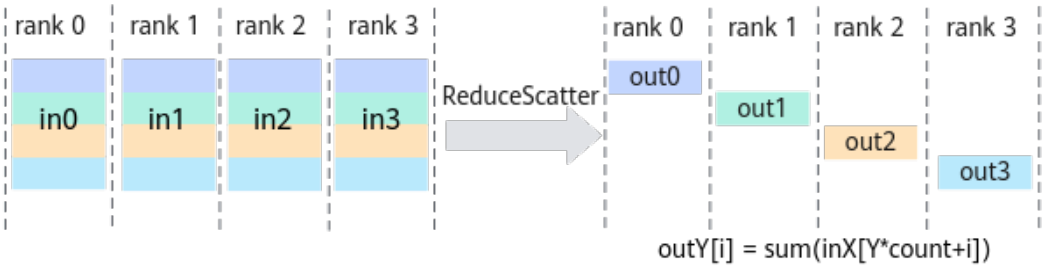


注意：每个rank只能有一个输入。

ReduceScatter

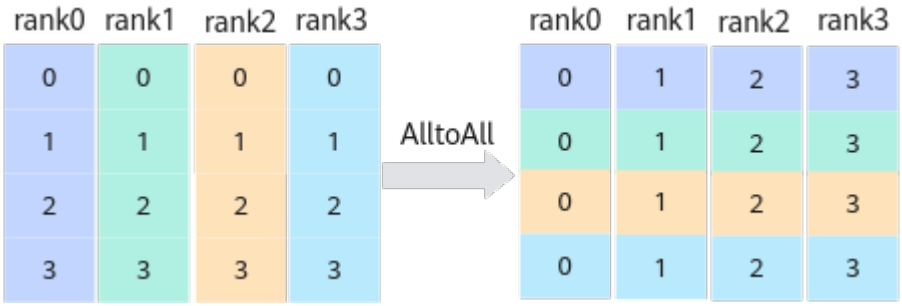
ReduceScatter操作是将所有rank的输入相加（或其他操作）后，再把结果按照rank编号均匀分散到各个rank的输出buffer，每个进程拿到其他进程1/ranksize份的数据进行归约操作。

如下图所示，有rank0、rank1、rank2、rank3四个rank，每个rank的输入数据切分成4份，每个进程分别取每个rank的1/4数据进行sum操作（或其他操作），将结果发送到输出buffer。



AlltoAll

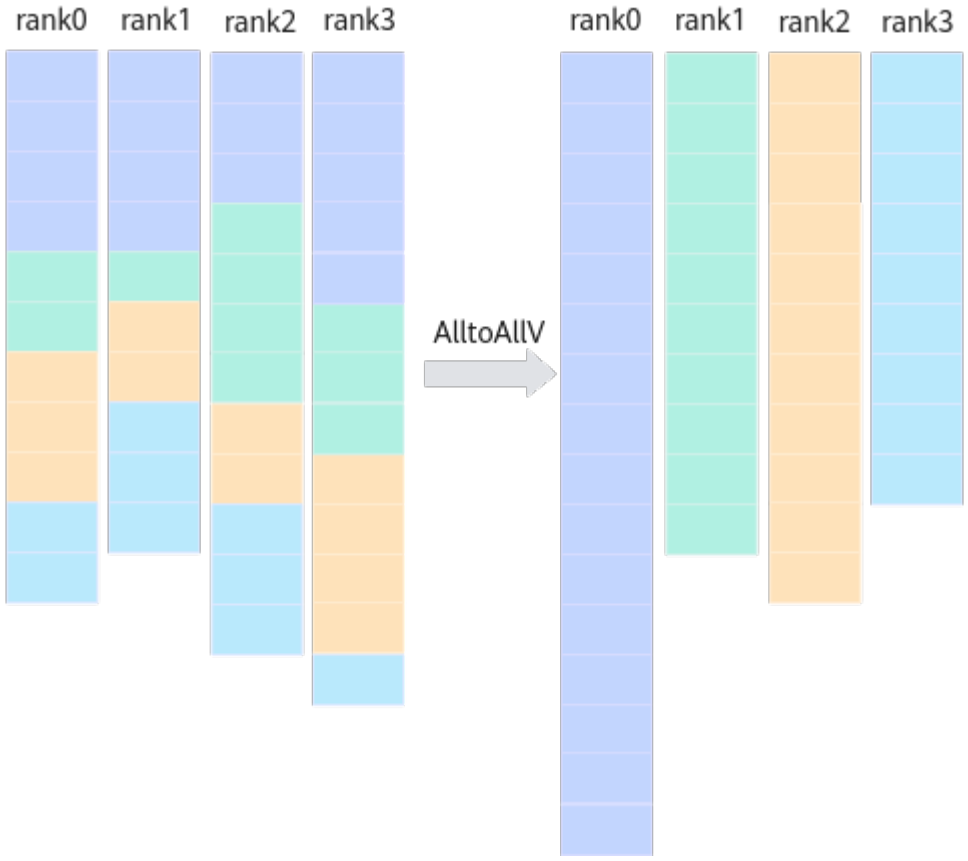
AlltoAll操作是向通信域内所有rank发送相同数据量的数据，并从所有rank接收相同数据量的数据。



AlltoAll操作将输入数据在特定的维度切分成特定的块数，并按顺序发送给其他rank，同时从其他rank接收输入，按顺序在特定的维度拼接数据。

AlltoAllV

AlltoAllV操作是向通信域内所有rank发送数据（数据量可以定制），并从所有rank接收数据。



5 分级通信原理

HCCL通常按Server内和Server间分为两级拓扑，分级执行集合通信，各集合通信算子的具体分级通信过程如下表所示：

表 5-1 集合通信算子分级通信过程

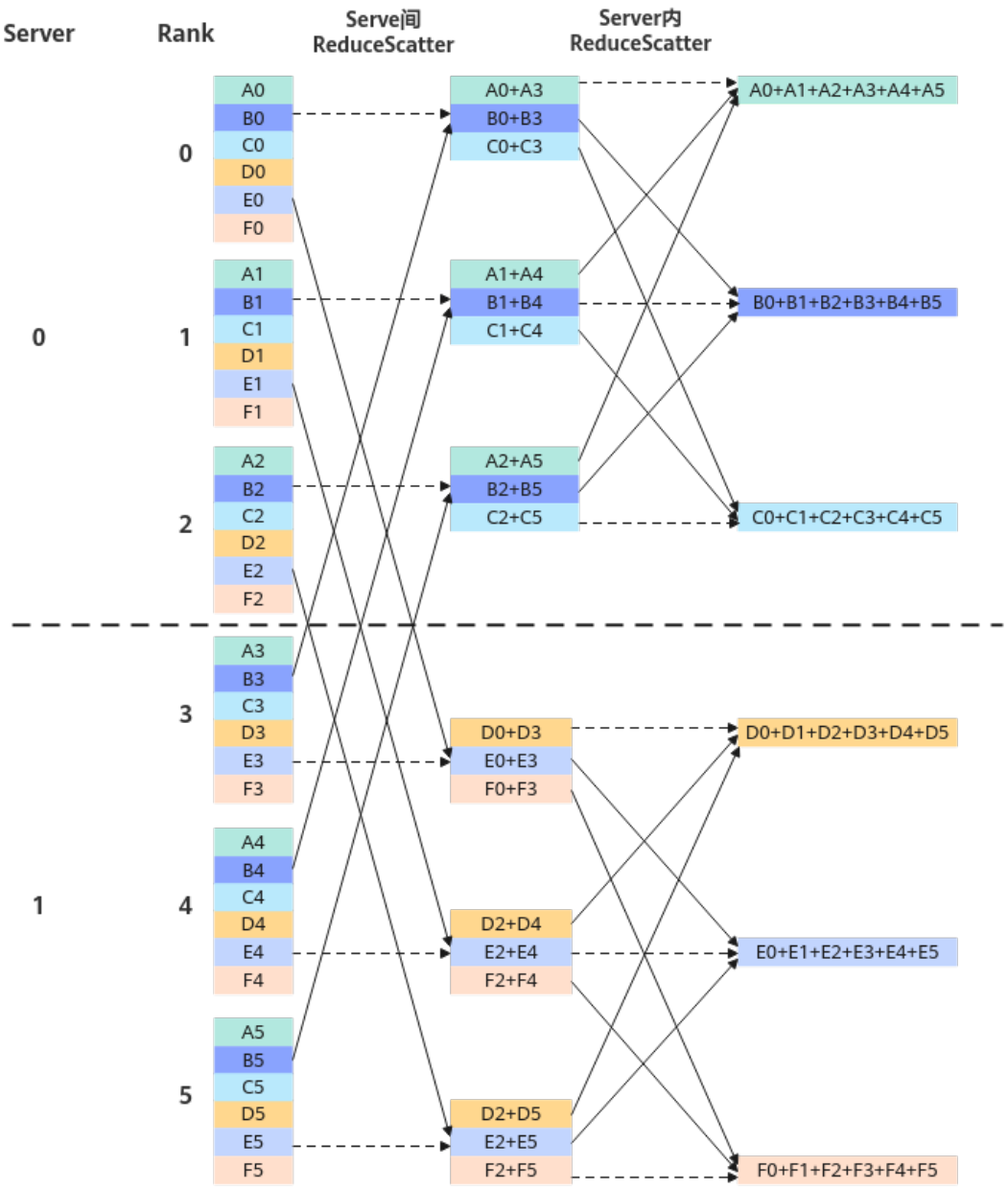
集合通信算子	阶段一	阶段二	阶段三
ReduceScatter	Server间ReduceScatter	Server内ReduceScatter	/
AllGather	Server内Allgather	Server间AllGather	/
AllReduce	Server内ReduceScatter	Server间AllReduce	Server内AllGather
Scatter	Server间Scatter	Server内Scatter	/
Broadcast	Server内Scatter	Server间Broadcast	Server内AllGather
Reduce	Server内ReduceScatter	Server间Reduce	Server内Gather
AlltoAll	Server内AlltoAll	Server间AlltoAll	/
AlltoAllV	Server内AlltoAllV	Server间AlltoAllV	/

下面以典型的通信算子ReduceScatter、AllGather、AllReduce为例介绍分级通信的流程。

ReduceScatter

如图5-1所示，ReduceScatter算子严格要求第i个rank最终得到第i份规约结果，为了保证Server间通信数据块的连续性，首先在Server间执行ReduceScatter操作，再在Server内执行ReduceScatter操作。

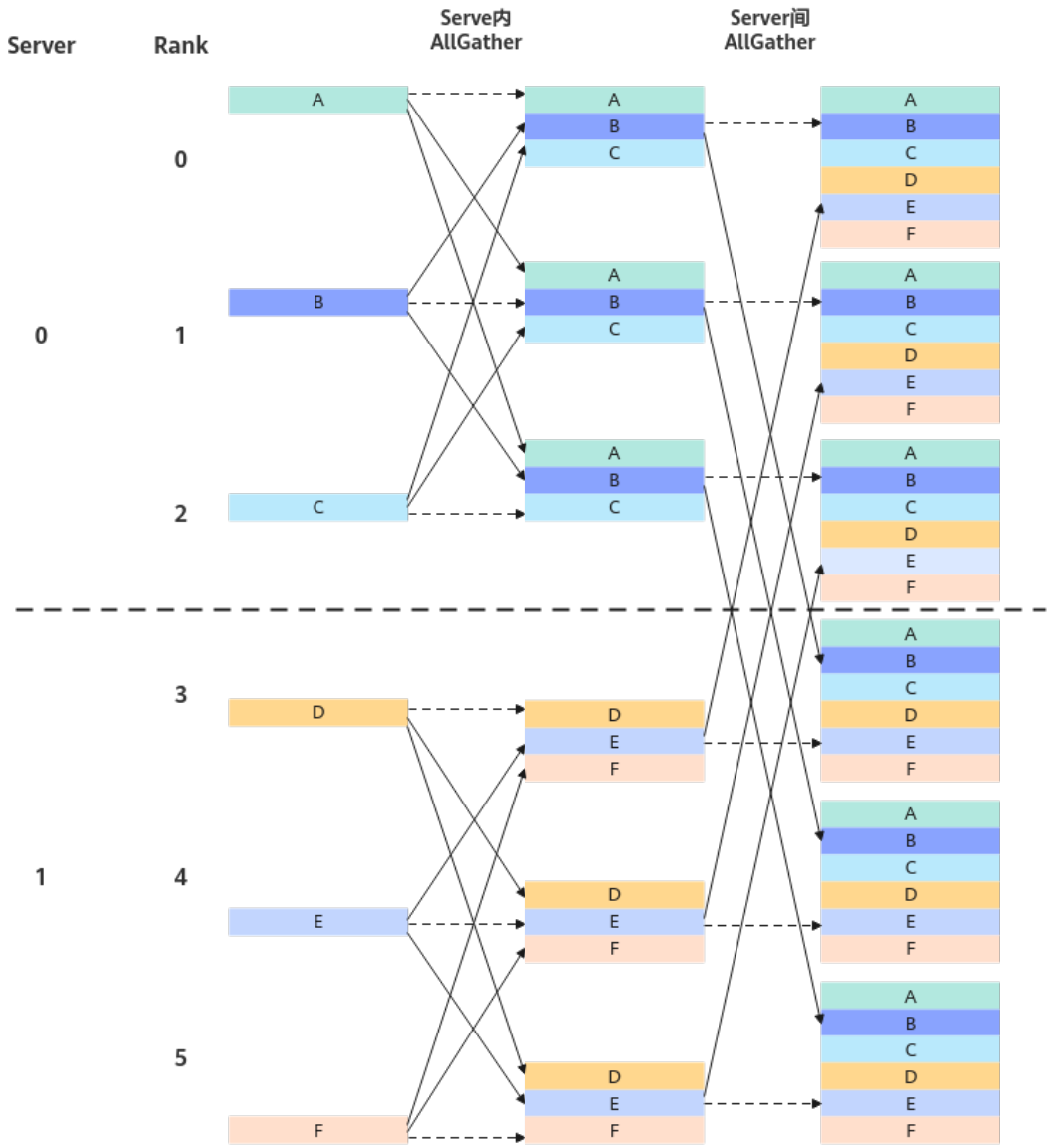
图 5-1 ReduceScatter 算子分级通信流程



AllGather

如图5-2所示，AllGather算子严格要求第i个rank的输入数据出现在结果的第i个位置上，为了保证Server间通信数据块的连续性，首先在Server内执行AllGather操作，然后在Server间执行AllGather操作。

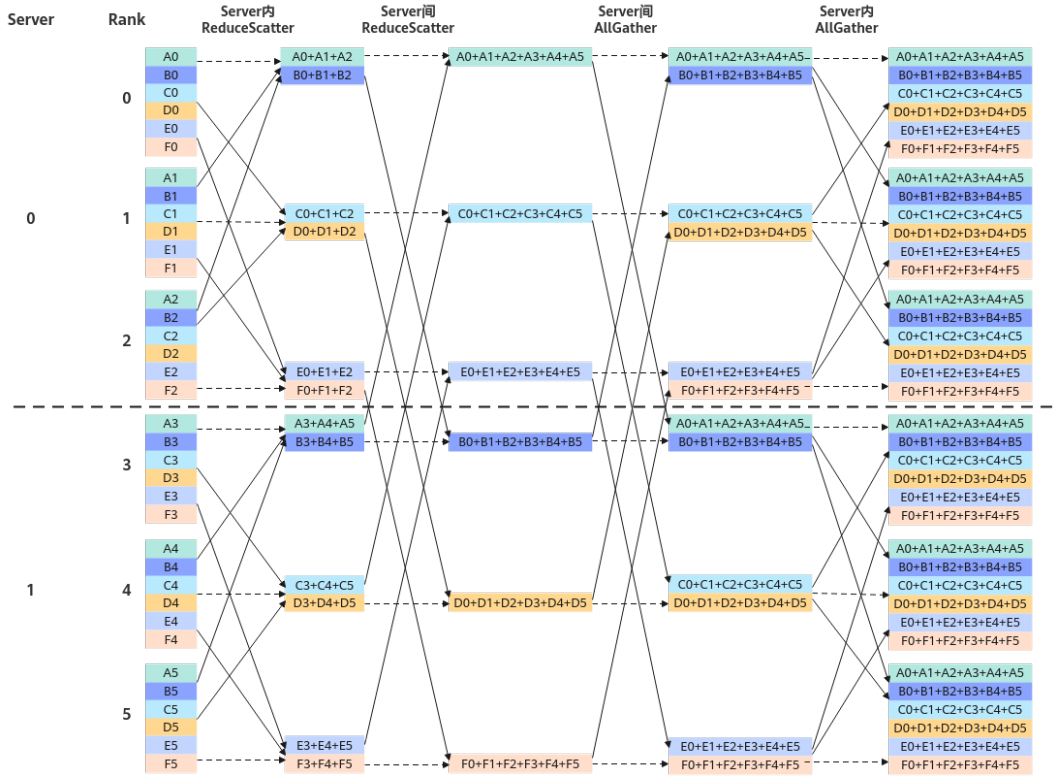
图 5-2 AllGather 算子分级通信流程



AllReduce

如图5-3所示，AllReduce算子的输出是完整的规约结果，因此虽然拆解为了ReduceScatter和AllGather两个阶段，但不需要严格遵循ReduceScatter和AllGather的语义，可以将较大数据量的通信过程放在带宽更高的Server内，即先在Server内执行ReduceScatter操作，然后在Server间执行AllReduce操作，最后在Server内执行AllGather操作。

图 5-3 AllReduce 算子分级通信流程



6 集合通信算法

针对同一个通信算子，随着网络拓扑、通信数据量、硬件资源等的不同，往往会采用不同的通信算法，从而最大化集群通信性能。HCCL提供了Mesh、Ring、Recursive Halving-Doubling（RHD）、NHR（Nonuniform Hierarchical Ring）、NB（Nonuniform Bruck）、Pipeline和Pairwise几种拓扑算法用于Server内和Server间的集合通信。

Server 内通信算法

HCCL通信域Server内支持Mesh、Ring和Star三种算法，具体使用的算法根据硬件拓扑自动选择，用户无须配置也不支持配置。

Server 间通信算法

HCCL通信域Server间支持Ring、RHD、NHR、NB、Pipeline、Pairwise几种算法的自适应选择，自适应算法会根据产品形态、数据量和Server个数进行选择，用户默认无需配置，大致算法选择逻辑如下：

- Ring算法：基于环结构的并行调度算法，适用于小规模节点数（例如<32机，且非2幂）和中大规模通信数据量（例如>=256M）的场景。
- RHD算法：递归二分和倍增算法，支持如下两种场景：
  - 通信域节点个数为2的整数次幂，任意数据量。
  - 中小通信数据量（例如<256M），任意节点数。
- NHR算法：非均衡的层次环算法，适用于中大规模节点数(例如>32机，且非2的整数次幂)和中大规模通信数据量（例如>=256M）的场景。
- NB算法：非均匀的数据块通信算法，适用于中大规模节点数(例如>32机，且非2的整数次幂)和中大规模通信数据量（例如>=256M）的场景。
- Pipeline算法：流水并行算法，适用于多机多卡且数据量较大（例如>=1M * RankSize）的场景。
- Pairwise算法：比较算法，仅用于AllToAll、AlltoAllV与AlltoAllVC算子，适用于数据量较小（例如<=1M * RankSize）的场景。

说明

- cann-hccl仓当前开放的算法有Mesh、Ring、RHD、PairWise、Star，开发者可访问**Gitee-cann-hccl**详细了解对应实现。
- 开发者若想指定集合通信Server间跨机通信算法，可通过环境变量HCCL_ALGO进行设置。

7 集群信息配置

除了通过ranktable文件配置资源信息的方式外，开发者还可以通过本节所述的环境变量组合的方式配置资源信息。

7.1 简介

针对TensorFlow网络，HCCL提供了两种集群信息配置方式用于集合通信初始化：ranktable配置文件的方式和环境变量的方式，开发者可以任选其中一种方式，但两种方式不可以混用。

针对PyTorch网络，集群信息由Ascend Extension for PyTorch内部自动获取，无需开发者手工配置，本章节跳过。

7.2 ranktable 文件配置资源信息

开发者可以通过ranktable文件配置参与集合通信的NPU资源信息，ranktable文件为json格式，开发者可以在此文件中配置全量NPU资源信息，后续进程启动时可使用其中指定的几个NPU资源。

配置文件说明（Atlas A2 训练系列产品）

针对Atlas A2 训练系列产品，ranktable文件配置说明如下：

```
{
"status":"completed",  // ranktable可用标识，completed为可用
"version":"1.0",       // ranktable模板版本信息,当前必须为"1.0"
"server_count":"1",  //参与训练的AI Server数目，此例中，只有一个AI Server
"server_list":
[
  {
      "device":[   // AI Server中的Device列表
                  {
                   "device_id":"0",  // 处理器HDC通道号
                   "device_ip":"192.168.1.8",    // 处理器真实网卡IP
                   "rank_id":"0"     // rank的标识，rankID从0开始
                   },
                   {
```

```
                    "device_id":"1",
                    "device_ip":"192.168.1.9",
                    "rank_id":"1"
                    }
                ],
        "server_id":"node_0"    //AI Server标识，String类型，请确保全局唯一
    }
]
}
```

表 7-1 ranktable 文件说明

配置项	配置说明	可选/必选
status	ranktable可用标识。 ● completed：表示ranktable可用，可执行训练。 ● initializing：表示ranktable不可用，不可执行训练。	必选
version	ranktable模板版本信息。配置为1.0。	必选
server_count	本次参与训练的AI Server个数。	必选
server_list	本次参与训练的AI Server列表。	必选
server_id	AI Server标识，字符串类型，长度小于64，请确保全局唯一。 配置示例：node_0	必选
device_id	昇腾AI处理器的ID，即Device在AI Server上的序列号。对应HDC通道号，请配置为整数。 取值范围：[0，实际Device数量-1]	必选
device_ip	昇腾AI处理器集成网卡IP，全局唯一，要求为常规IPv4或IPv6格式。 **需要注意：** ● 多机场景下，device_ip必须配置。 ● 单机场景下，device_ip可不配置。 可以在当前AI Server执行指令cat /etc/hccn.conf获取网卡IP，例如： address_0=xx.xx.xx.xx netmask_0=xx.xx.xx.xx netdetect_0=xx.xx.xx.xx address_1=xx.xx.xx.xx netmask_1=xx.xx.xx.xx netdetect_1=xx.xx.xx.xx ... 查询到的address_xx即为网卡IP，address后的序号为昇腾AI处理器或BS9SX1A AI处理器SoC物理ID，即device_id，后面的ip地址即为需要用户填入的该device对应的网卡IP。	可选

配置项	配置说明	可选/必选
rank_id	Rank唯一标识，请配置为整数，从0开始配置，且全局唯一，取值范围：[0, 总Device数量-1]。 **为方便管理，建议rank_id按照Device物理连接顺序进行排序，即将物理连接上较近的device编排在一起。** 例如，若device_ip按照物理连接从小到大设置，则rank_id也建议按照从小到大的顺序设置。	必选

配置文件说明（Atlas 300I Duo 推理卡）

针对Atlas 300I Duo 推理卡，ranktable文件配置说明如下：

```
{
"status":"completed",  // ranktable可用标识，completed为可用
"version":"1.0",       // ranktable模板版本信息,当前必须为"1.0"
"server_count":"1",  //参与训练的AI Server数目，此例中，只有一个AI Server
"server_list":
[
  {
      "device":[   // AI Server中的Device列表
                  {
                   "device_id":"0",  // 处理器HDC通道号
                   "device_ip":"192.168.1.8",   // 处理器真实网卡IP
                   "rank_id":"0"     // rank的标识，rankID从0开始
                   },
                   {
                    "device_id":"1",
                    "device_ip":"192.168.1.9",
                    "rank_id":"1"
                    }
             ],
       "server_id":"node_0"   //AI Server标识，String类型，请确保全局唯一
   }
]
}
```

表 7-2 ranktable 文件说明

配置项	配置说明	可选/必选
status	ranktable可用标识。 - completed：表示ranktable可用，可执行训练。 - initializing：表示ranktable不可用，不可执行训练。	必选
version	ranktable模板版本信息，当前仅支持配置为1.0。	必选
server_count	本次参与训练的AI Server个数。	必选
server_list	本次参与训练的AI Server列表。	必选

配置项	配置说明	可选/必选
server_id	AI Server标识，字符串类型，长度小于64，请确保全局唯一。 配置示例：node_0	必选
device_id	昇腾AI处理器的ID，即Device在AI Server上的序列号，对应HDC通道号，请配置为整数。 取值范围：[0，实际Device数量-1] **须知** “device_id”配置项的优先级高于环境变量“ASCEND_DEVICE_ID”。	必选
device_ip	昇腾AI处理器集成网卡IP，全局唯一，要求为常规IPv4或IPv6格式。 可以在当前AI Server执行指令“cat /etc/hccn.conf”获取网卡IP，例如： address_0=xx.xx.xx.xx netmask_0=xx.xx.xx.xx netdetect_0=xx.xx.xx.xx address_1=xx.xx.xx.xx netmask_1=xx.xx.xx.xx netdetect_1=xx.xx.xx.xx ... 查询到的address_xx即为网卡IP，address后的序号为昇腾AI处理器物理ID，即device_id，后面的ip地址即为需要用户填入的该device对应的网卡IP。	必选
rank_id	Rank唯一标识，请配置为整数，从0开始配置，且全局唯一，取值范围：[0, 总Device数量-1] **为方便管理，建议rank_id按照Device物理连接顺序进行排序，即将物理连接上较近的device编排在一起。** 例如，若device_ip按照物理连接从小到大设置，则rank_id也建议按照从小到大的顺序设置。	必选

配置文件说明（Atlas 训练系列产品）

针对Atlas 训练系列产品，在ranktable文件中配置参与训练的昇腾AI处理器信息支持两种配置模板，全新场景推荐使用模板一，模板二用于兼容部分已有场景。

- 模板一（推荐使用）

```
{
"status":"completed",  // ranktable可用标识，completed为可用
"version":"1.0",       // ranktable模板版本信息,当前必须为"1.0"
"server_count":"1",//参与训练的AI Server数目，此例中，只有一个AI Server
"server_list":
[
  {
      "device":[   // AI Server中的Device列表
                  {
                   "device_id":"0",  // 处理器HDC通道号
                   "device_ip":"192.168.1.8",    // 处理器真实网卡IP
                   "rank_id":"0"     // rank的标识，rankID从0开始
```



```
                },
                {
                 "device_id":"1",
                 "device_ip":"192.168.1.9",
                 "rank_id":"1"
                 }
            ],
        "server_id":"node_0"    //AI Server标识，String类型，请确保全局唯一
    }
]
}
```

表 7-3 ranktable 文件说明

配置项	配置说明	可选/必选
status	ranktable可用标识。 ● completed：表示ranktable可用，可执行训练。 ● initializing：表示ranktable不可用，不可执行训练。	必选
version	ranktable模板版本信息，当前仅支持配置为1.0。	必选
server_count	本次参与训练的AI Server个数。	必选
server_list	本次参与训练的AI Server列表。	必选
server_id	AI Server标识，字符串类型，长度小于64，请确保全局唯一。 配置示例：node_0	必选
device_id	昇腾AI处理器的ID，即Device在AI Server上的序列号，对应HDC通道号，请配置为整数。 取值范围：[0，实际Device数量-1] **须知** “device_id”配置项的优先级高于环境变量“ASCEND_DEVICE_ID”。	必选
device_ip	昇腾AI处理器集成网卡IP，全局唯一，要求为常规IPv4或IPv6格式。 可以在当前AI Server执行指令“cat /etc/hccn.conf”获取网卡IP，例如： address_0=xx.xx.xx.xx netmask_0=xx.xx.xx.xx netdetect_0=xx.xx.xx.xx address_1=xx.xx.xx.xx netmask_1=xx.xx.xx.xx netdetect_1=xx.xx.xx.xx ... 查询到的address_xx即为网卡IP，address后的序号为昇腾AI处理器物理ID，即device_id，后面的ip地址即为需要用户填入的该device对应的网卡IP。	必选

配置项	配置说明	可选/必选
rank_id	Rank唯一标识，请配置为整数，从0开始配置，且全局唯一，取值范围：[0, 总Device数量-1] **为方便管理，建议rank_id按照Device物理连接顺序进行排序，即将物理连接上较近的device编排在一起。** 例如，若device_ip按照物理连接从小到大设置，则rank_id也建议按照从小到大的顺序设置。	必选

- 模板二（兼容部分已有场景）

```
{
"status":"completed",  // ranktable可用标识，completed为可用
"group_count":"1",     // group数量，建议为1
"group_list":          // group列表
 [
  {
   "group_name":"hccl_world_group",//group名称，建议hccl_world_group
   "instance_count":"2",        // instance实例个数，容器场景下可理解为容器个数
   "device_count":"2",         // group中的所有device数目
   "instance_list":[           // instance实例信息列表
      {
        "pod_name":"tf-bae41",    //instance实例名称，一般为容器名称
        "server_id":"node_0",  //AI Server标识，String类型，请确保全局唯一
        "devices":[               //instance实例的device列表
         {
           "device_id":"0",          // 昇腾AI处理器HDC通道号
           "device_ip":"192.168.1.8"  // 昇腾AI处理器真实网卡IP
         }
        ]
      },
      {
         "pod_name":"tf-tbdf1",
         "server_id":"node_1",
         "devices":[
            {
               "device_id":"1",
               "device_ip":"192.168.1.9"
            }
         ]
       }
    ]
  }
 ]
}
```

表 7-4 ranktable 文件说明

配置项	配置说明	可选/必选
status	ranktable可用标识。 - completed：表示ranktable可用，可执行训练。 - initializing：表示ranktable不可用，不可执行训练。	必选

配置项	配置说明	可选/必选
group_count	用户申请的group数量，建议配置为1。	必选
group_list	Group列表。	必选
group_name	Group名称，当group_count为1时，建议配置为hccl_world_group或者空。因为当前版本无论定义为任何值，都会创建名称为hccl_world_group的group。 如果通过该配置文件创建了多个group，则系统会自动将多个group合并为一个名称为“hccl_world_group”的group资源。	可选
instance_count	和instance_list中pod_name个数保持一致，例如：容器场景下为容器实际数量。	必选
device_count	group中设备数量。	必选
instance_list	instance实例信息列表。	必选
pod_name	用户自定义配置，保持instance_list内全局唯一。	必选
server_id	AI Server标识，字符串类型，长度小于64，请确保全局唯一。 配置示例：node_0	必选
devices	devices信息列表。	必选
device_id	昇腾AI处理器物理ID，即Device在Server上的序列号，对应HDC通道号，请配置为整数。 取值范围：[0，实际Device数量-1] **须知** “device_id”配置项的优先级高于环境变量“ASCEND_DEVICE_ID”。	必选
device_ip	昇腾AI处理器集成网卡IP，全局唯一，要求为常规IPv4或IPv6格式。 可以在当前Server执行指令cat /etc/hccn.conf获取网卡IP。	必选

7.3 环境变量配置资源信息

除了通过ranktable文件配置资源信息的方式外，开发者还可以通过本节所述的环境变量组合的方式配置资源信息。

通过环境变量配置资源信息的方式仅支持如下产品型号：

Atlas 训练系列产品

Atlas A2 训练系列产品

配置说明

需要在执行训练的每个AI Server节点上分别配置如下环境变量，进行资源信息的配置，示例如下：

```
export CM_CHIEF_IP = 192.168.1.1
export CM_CHIEF_PORT = 6000
export CM_CHIEF_DEVICE = 0
export CM_WORKER_SIZE = 8
export CM_WORKER_IP = 192.168.0.1
export HCCL_SOCKET_FAMILY=AF_INET
```

- CM_CHIEF_IP、CM_CHIEF_PORT、CM_CHIEF_DEVICE用于配置Master节点的Host监听IP、监听端口与主Device ID。其中Master节点为集群管理主节点，负责集群内设备信息的管理、资源的分配和调度等。
  - 监听IP可以通过ifconfig命令进行查询，要求为常规IPv4或IPv6格式。
  - 指定的监听端口号需要确保在训练进程拉起时，无其他业务占用。
- CM_WORKER_SIZE：用于配置参与集群训练的Device数量。
- CM_WORKER_IP：用于配置当前节点与Master进行通信时所用的网卡IP，可通过ifconfig命令查询，要求为常规IPv4或IPv6格式。

  需要确保指定的网卡IP能够与Master节点正常通信。
- HCCL_SOCKET_FAMILY：**此环境变量可选**，用于控制Device侧通信网卡使用的IP协议版本。AF_INET代表使用IPv4协议，AF_INET6代表使用IPv6协议，**缺省时，优先使用IPv4协议**。

  说明

  如果环境变量“HCCL_SOCKET_FAMILY”指定的IP协议与实际获取到的网卡信息不匹配，则以实际环境上的网卡信息为准。

  例如，环境变量“HCCL_SOCKET_FAMILY”指定为“AF_INET6”，但Device侧只存在IPv4协议的网卡，则实际会使用IPv4协议的网卡。

配置示例

假设执行分布式训练的Server节点数量为2，Device数量为16为例，每个Server节点有8个Device。拉起训练进程前，在对应的shell窗口中配置如下环境变量，进行资源信息的配置。

- 节点0，此节点为Master节点，负责集群信息管理、资源分配与调度。
  ```
  export CM_CHIEF_IP = 192.168.1.1
  export CM_CHIEF_PORT = 6000
  export CM_CHIEF_DEVICE = 0
  export CM_WORKER_SIZE = 16
  export CM_WORKER_IP = 192.168.1.1
  ```
- 节点1
  ```
  export CM_CHIEF_IP = 192.168.1.1
  export CM_CHIEF_PORT = 6000
  export CM_CHIEF_DEVICE = 0
  export CM_WORKER_SIZE = 16
  export CM_WORKER_IP = 192.168.2.1
  ```


8 通信功能开发

8.1 简介

HCCL提供了C与Python两种语言的开发接口，用于实现分布式能力。

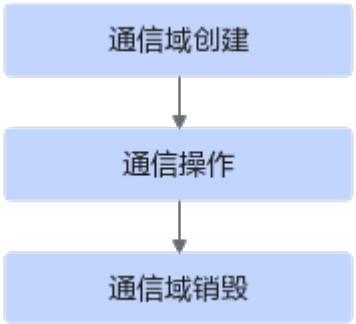
- C语言接口用于实现单算子模式下的框架适配，实现分布式能力。

  针对PyTorch框架网络，HCCL单算子API已嵌入到Ascend Extension for PyTorch后端代码中，PyTorch用户直接使用PyTorch原生集合通信API，即可实现分布式能力。
- Python语言接口用于实现TensorFlow网络在昇腾AI处理器执行分布式优化。

本章节针对如何调用HCCL的C语言接口进行集合通信功能的开发进行介绍。

开发者调用HCCL C接口实现集合通信功能的主要开发流程如下所示。

图 8-1 集合通信操作流程



1. 首先进行集群信息配置，创建通信域句柄，并初始化HCCL通信域。
2. 实现通信操作，HCCL通信操作包含两大类：点对点通信与集合通信。
   - 点对点通信，指在多NPU环境下两个NPU之间直接传输数据的过程，常用于pipeline并行场景下对激活值的数据收发。HCCL提供了不同粒度的点对点通信，包括单rank到单rank的单收单发接口，以及多个rank之间的批量收发接口。
   - 集合通信，指多个NPU共同参与进行数据传输操作，例如AllReduce，AllGather，Broadcast等，常用于大规模集群中不同NPU之间的梯度同步和参数更新等操作。集合通信操作可让所有计算节点并行、高效、有序执行数据交换，提升数据传输效率。
3. 集合通信操作完成后，需要释放相关资源，销毁通信域。

8.2 通信域管理

通信域是集合通信算子执行的上下文，管理对应的通信对象（例如一个NPU就是一个通信对象）和通信所需的资源。通信域中的每个通信对象称为一个rank，每个rank都会分配一个介于0~n-1（n为npu的数量）的唯一标识。

通信域创建根据用户场景的不同主要有以下几种方式：

- 多机集合通信场景，如果有完整的描述集群信息的ranktable文件，可通过HcclCommInitClusterInfo接口创建通信域，或者通过HcclCommInitClusterInfoConfig接口创建具有特定配置的通信域。
- 多机集合通信场景，如果无完整的ranktable，可通过HcclGetRootInfo接口与HcclCommInitRootInfo/HcclCommInitRootInfoConfig接口配合使用，基于root节点广播方式创建通信域。
- 单机集合通信场景，可通过HcclCommInitAll接口在单机内批量创建通信域。
- 基于已有的通信域，可通过HcclCreateSubCommConfig接口切分具有特定配置的子通信域。

须知

- 针对同一个rank，通信算子需要one by one保序调用，不支持并发下发。
- 针对整个通信域，通信算子的调用在不同rank间要保证同一下发顺序。

基于 ranktable 创建通信域

多机集合通信、基于集群信息配置ranktable文件创建通信域的场景，每张卡需要使用一个单独的进程参考如下流程创建通信域：

1. 构造ranktable文件（ranktable文件的配置可参见**7.2 ranktable文件配置资源信息**），如果已存在ranktable文件，也可以通过环境变量RANK_TABLE_FILE获取。
2. 每张卡使用HcclCommInitClusterInfo接口创建通信域，或者使用HcclCommInitClusterInfoConfig接口创建具有特定配置的通信域。

一个简单的代码示例片段如下：

```
// 获取ranktable路径
char* rankTableFile = getenv("RANK_TABLE_FILE");
// 定义通信域句柄
HcclComm hcclComm;
```



```
// 初始化HCCL通信域
HcclCommInitClusterInfo(rankTableFile, devId, &hcclComm);

/* 集合通信操作 */

// 销毁HCCL通信域
HcclCommDestroy(hcclComm);
```

完整的代码示例链接：

- 点对点通信（HcclSend/HcclRecv）：**8.5.2.1 HcclCommInitClusterInfo初始化方式**和**8.5.2.2 HcclCommInitClusterInfoConfig初始化方式**。
- 集合通信（HcclAllReduce）：**8.5.3.1 HcclCommInitClusterInfo初始化方式**和**8.5.3.2 HcclCommInitClusterInfoConfig初始化方式**。

基于 root 节点广播方式创建通信域

多机集合通信场景，若无完整的集群信息配置ranktable文件，HCCL提供了基于root节点广播的方式创建通信域，详细流程如下：

1. 配置环境变量HCCL_IF_IP，指定root通信网卡的IP地址。

   需要配置为root节点Host网卡的IP地址，可以是IPv4或IPv6格式，仅支持配置一个IP地址，配置示例如下：

   ```
   export HCCL_IF_IP=10.10.10.1
   ```

2. 在root节点调用HcclGetRootInfo接口，生成root节点rank标识信息“rootInfo”，包括device ip、device id等信息。
3. 将root节点的rank信息广播至集群中的所有rank。
4. 在所有节点调用HcclCommInitRootInfo或者HcclCommInitRootInfoConfig接口（创建具有特定配置的通信域），基于接收到的“rootInfo”，以及本rank的rank id等信息，进行HCCL初始化。

需要注意：每个卡需要使用一个单独的进程进行以上操作。

使用HcclCommInitRootInfo接口创建通信域的简单代码示例片段如下：

```
// 获取当前进程在所属进程组的编号
MPI_Comm_rank(MPI_COMM_WORLD, &procRank);
int devId = procRank;
// 在root节点获取其rank信息
HcclRootInfo rootInfo;
int32_t rootRank = 0;
if(devId == rootRank) {
    HcclGetRootInfo(&rootInfo);
}
// 将root_info广播到通信域内的其他rank
MPI_Bcast(&rootInfo, HCCL_ROOT_INFO_BYTES, MPI_CHAR, rootRank, MPI_COMM_WORLD);
MPI_Barrier(MPI_COMM_WORLD);
// 定义通信域句柄
HcclComm hcclComm;
// 初始化HCCL通信域
HcclCommInitRootInfo(devCount, &rootInfo, devId, &hcclComm);

/* 集合通信操作 */

// 销毁HCCL通信域
HcclCommDestroy(hcclComm);
```

完整的代码示例链接：

- 点对点通信（HcclSend/HcclRecv）：**8.5.2.3 HcclCommInitRootInfo初始化方式**和**8.5.2.4 HcclCommInitRootInfoConfig初始化方式**。

- 集合通信（HcclAllReduce）：**8.5.3.3 HcclCommInitRootInfo初始化方式**和**8.5.3.4 HcclCommInitRootInfoConfig初始化方式**。

单机内批量创建通信域

单机通信场景中，开发者通过一个进程统一创建多张卡的通信域，其中一张卡对应一个线程，创建流程如下：

1. 构造通信域中的Device列表，例如：{0, 1, 2, 3, 4, 5, 6, 7}，其中列表中的Device ID是逻辑ID。
2. 在进程中调用HcclCommInitAll接口创建通信域。

```
uint32_t ndev = 8;
// 构造Device的逻辑ID列表
int32_t devices[8] = {0, 1, 2, 3, 4, 5, 6, 7};
// 定义通信域句柄
HcclComm comms[ndev];
// 初始化HCCL通信域
HcclCommInitAll(ndev, devices, comms);

// 启动线程执行集合通信操作
std::vector<std::unique_ptr<std::thread> > threads(ndev);
struct ThreadContext args[ndev];
for (uint32_t i = 0; i < ndev; i++) {
    args[i].device = i;
    args[i].comm = comms[i];
   /*  集合通信操作   */
}

// 销毁HCCL通信域
for (uint32_t i = 0; i < ndev; i++) {
    HcclCommDestroy(comms[i]);
}
```

需要注意，多线程调用集合通信操作API时（例如HcclAllReduce时），需要确保不同线程中调用集合通信操作API的前后时间差不超过环境变量HCCL_CONNECT_TIMEOUT的时间，避免建链超时。

完整的代码示例可参见点对点通信（HcclSend/HcclRecv）样例**8.5.2.5 HcclCommInitAll初始化**和集合通信（HcclAllReduce）样例**8.5.3.5 HcclCommInitAll初始化方式**。

基于已有通信域切分子通信域

HCCL提供了HcclCreateSubCommConfig接口，实现基于已有通信域切分具有特性配置子通信域的功能。该子通信域创建方式无需进行socket建链与rank信息交换，可应用于业务故障下的快速通信域创建。

使用HcclCreateSubCommConfig接口切分子通信域的代码示例可参见点对点通信（HcclSend/HcclRecv）样例**8.5.2.6 HcclCreateSubCommConfig方式创建子通信域**和集合通信（HcclAllReduce）样例**8.5.3.6 HcclCreateSubCommConfig初始化方式**。

需要注意，该接口仅支持从全局通信域切分子通信域，不支持通信域的嵌套切分。

销毁通信域

集合通信操作完成后，需要调用运行时管理接口释放通信所用的内存、Stream、Device资源，并调用HcclCommDestroy接口销毁指定的通信域，完成的示例代码可参见**8.5 样例代码**。

8.3 集合通信

集合通信是指多个NPU共同参与进行数据传输，从而形成一次集体操作的通信模式，常用于大规模集群中不同NPU之间的梯度同步和参数更新等场景。

HCCL支持AllReduce、Broadcast、AllGather、Scatter、ReduceScatter、Reduce、AlltoAll和AlltoAllV等通信原语，关于这些通信原语的详细含义可参见**4 集合通信原语**。针对这些通信原语，HCCL提供了对应的API供开发者调用，用于快速实现集合通信能力。

例如，HCCL针对集合通信操作AllReduce提供了HcclAllReduce接口，原型定义如下：

```
HcclResult HcclAllReduce(void *sendBuf, void *recvBuf, uint64_t count, HcclDataType dataType,
HcclReduceOp op, HcclComm comm, aclrtStream stream)
```

HcclAllReduce用于将通信域内所有节点的输入进行Reduce操作，再把结果发送到所有节点的输出，其中op参数用于指定Reduce的操作类型，当前版本支持的操作类型有sum、prod、max、min。HcclAllReduce允许每个节点只有一个输入。

如下代码片段所示，将通信域内所有输入内存中的数据，按照float32的数据格式执行加法操作（示例中每个rank中只有一个数据参与），然后把相加结果发送到所有节点的输出内存。

```
void* host_buf = nullptr;
void* send_buff = nullptr;
void* recv_buff = nullptr;
uint64_t count = 1;
int malloc_kSize = count * sizeof(float);
aclrtStream stream;
aclrtCreateStream(&stream);

//申请集合通信操作的内存
aclrtMalloc((void**)&sendbuff, mal1oc_kSize, ACL_MEM_MALLOC_HUGE_FIRST);
aclrtMalloc((void**)&recvbuff, malloc_kSize, ACL_MEM_MALLOC_HUGE_FIRST);

//初始化输入内存
aclrtMallocHost((void**)&host_buf, malloc_kSize);
aclrtMemcpy((void*)send_buff, malloc_kSize, (void*)host_buf, malloc_kSize,
ACL_MEMCPY_HOST_TO_DEVICE);

//执行集合通信操作
HcclAllReduce((void *)send_buff, (void*)recv_buff, count, HCCL_DATA_TYPE_FP32, HCCL_REDUCE_SUM,
hcclComm, stream);
```

HcclAllReduce接口调用的完整示例可参见**8.5.3 集合通信样例（HcclAllReduce）**，HCCL提供的更多集合通信接口可参见集合通信算子。

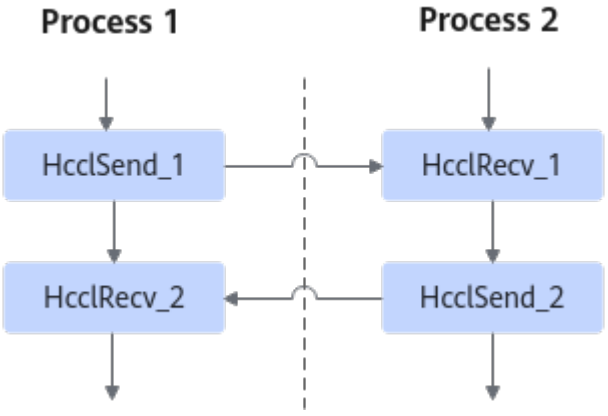
8.4 点对点通信

点对点通信是指多NPU环境下两个NPU之间直接传输数据的通信模式，常用于pipeline并行场景下对激活值的数据收发。

HCCL提供了不同粒度的点对点通信，包括单rank到单rank的单收单发接口，以及多个rank之间的批量收发接口。

HcclSend/HcclRecv

HcclSend/HcclRecv用于单收单发场景，需严格保序下发并配对使用，收发两端需完成同步后才能进行数据收发，数据收发完成后才能执行后续算子任务。



一个简单的代码示例片段如下：

```
if(rankId == 0){
    uint32_t destRank = 1;
    uint32_t srcRank = 1;
    HcclSend(sendBuf, count, dataType, destRank, hcclComm, stream);
    HcclRecv(recvBuf, count, dataType, srcRank, hcclComm, stream);
}
if(rankId == 1){
    uint32_t srcRank = 0;
    uint32_t destRank = 0;
    HcclRecv(recvBuf, count, dataType, srcRank, hcclComm, stream);
    HcclSend(sendBuf, count, dataType, destRank, hcclComm, stream);
}
```

完整的代码示例可参见**8.5.2 点对点通信样例（HcclSend/HcclRecv）**。

HcclBatchSendRecv

HcclBatchSendRecv可用于通信域内多个rank之间的数据收发，该接口有两个特征：

- 接口内部会对批量数据收发顺序进行重排，所以不严格要求单次接口调用中批量收发的任务顺序，但约束批量收发在一次接口调用下全量收发任务的严格匹配。
- 收发过程独立调度执行，收发不相互阻塞，从而实现双工链路并发。

该接口使用时需要注意：单次接口调用下，两个rank之间单向数据流仅支持传递一块内存数据，避免收发过程中混淆多块内存数据的收发地址。

一个简单的代码示例片段如下：

```
HcclSendRecvItem sendRecvInfo[itemNum];
HcclSendRecvType currType;
for (size_t i = 0; i < op_type.size(); ++i) {
    if (op_type[i] == "isend") {
        currType = HcclSendRecvType::HCCL_SEND;
    } else if (op_type[i] == "irecv") {
        currType = HcclSendRecvType::HCCL_RECV;
    }
    sendRecvInfo[i] = HcclSendRecvItem{currType,
                                       tensor_ptr_list[i],
                                       count_list[i],
                                       type_list[i],
                                       remote_rank_list[i]
                                       };
}
HcclBatchSendRecv(sendRecvInfo, itemNum, hcclComm, stream);
```


8.5 样例代码

8.5.1 安装配置 MPI

HCCL的通信域初始化依赖MPI拉起多个进程，所以进行HCCL的代码样例编写前，需要先安装配置MPI软件包。

MPI软件包的安装配置可参见《HCCL性能测试工具使用指南》中的MPI安装与配置。

8.5.2 点对点通信样例（HcclSend/HcclRecv）

8.5.2.1 HcclCommInitClusterInfo 初始化方式

该样例仅支持单机8卡的组网。

准备 ranktable 文件

该样例通过获取ranktable的方式进行初始化，所以需准备一份ranktable文件配置集群信息，供后续调用接口时使用。

配置“RANK_TABLE_FILE”环境变量，指定ranktable文件所在路径，如下所示，文件名称为“ranktable.json”。

```
export RANK_TABLE_FILE=/home/test/ranktable.json
```

以**Atlas A2 训练系列产品**为例，ranktable.json配置示例如下，不同产品形态ranktable文件的配置示例及详细参数说明可参见**7.2 ranktable文件配置资源信息**。

```
{
    "status":"completed",  // ranktable可用标识，completed为可用
    "version": "1.0",
    "server_count": "1",  // 参与训练的AI Server数目
    "server_list": [{
        "server_id": "SERVER_ID_SV1",  // AI Server标识，String类型，请确保全局唯一
        "device": [{          // AI Server中的Device列表
            "device_id": "0",
            "device_ip": "192.168.1.8",
            "rank_id": "0"
        },
        {
            "device_id": "1",
            "device_ip": "192.168.1.9",
            "rank_id": "1"
        },
        {
            "device_id": "2",
            "device_ip": "192.168.1.10",
            "rank_id": "2"
        },
        {
            "device_id": "3",
            "device_ip": "192.168.1.10",
            "rank_id": "3"
        },
        {
            "device_id": "4",
            "device_ip": "192.168.1.10",
            "rank_id": "4"
        },
        {
```



```
                "device_id": "5",
                "device_ip": "192.168.1.10",
                "rank_id": "5"
            },
            {
                "device_id": "6",
                "device_ip": "192.168.1.10",
                "rank_id": "6"
            },
            {
                "device_id": "7",
                "device_ip": "192.168.1.11",
                "rank_id": "7"
            }]
        }]
}
```

代码示例

```
#include <iostream>
#include <vector>
#include <memory>
#include <thread>
#include <chrono>
#include "hccl/hccl.h"
#include "hccl/hccl_types.h"
#include "mpi.h"

#define ACLCHECK(ret) do { \
    if(ret != ACL_SUCCESS)\
    {\
        printf("acl interface return err %s:%d, retcode: %d \n", __FILE__, __LINE__, ret);\
        return ret;\
    }\
} while(0)

#define HCCLCHECK(ret) do {  \
    if(ret != HCCL_SUCCESS) \
    {   \
        printf("hccl interface return errreturn err %s:%d, retcode: %d \n", __FILE__, __LINE__, ret); \
        return ret;\
    } \
} while(0)

struct ThreadContext {
    HcclComm comm;
    int32_t device;
};
int Sample(void *arg)
{
    ThreadContext* ctx = (ThreadContext *)arg;
    // 申请通信用device、sendBuf，recvBuf内存、stream等资源
    ACLCHECK(aclrtSetDevice(ctx->device));
    aclrtStream stream;
    ACLCHECK(aclrtCreateStream(&stream));
    void* sendBuff;
    void* recvBuff;
    void* hostBuff;
    uint64_t count = 8;
    int mallocSize = count * sizeof(float);
     //初始化输入内存
    ACLCHECK(aclrtMallocHost((void**)&hostBuff, mallocSize));
    float* tmpHostBuff = static_cast<float*>(hostBuff);
    for (uint32_t i = 0; i < count; ++i) {
        tmpHostBuff[i] = 2;
    }
    ACLCHECK(aclrtMalloc((void**)&sendBuff, mallocSize, ACL_MEM_MALLOC_HUGE_FIRST));
    ACLCHECK(aclrtMemcpy((void*)sendBuff, mallocSize, (void*)hostBuff, mallocSize,
ACL_MEMCPY_HOST_TO_DEVICE));
```



```
    ACLCHECK(aclrtMalloc((void**)&recvBuff, mallocSize, ACL_MEM_MALLOC_HUGE_FIRST));
    // 执行SendRecv操作
    if (ctx->device / 4 == 0) {
        HCCLCHECK(HcclSend(sendBuff, count, HCCL_DATA_TYPE_FP32, ctx->device + 4, ctx->comm, stream));
    } else {
        HCCLCHECK(HcclRecv(recvBuff, count, HCCL_DATA_TYPE_FP32, ctx->device - 4, ctx->comm, stream));
    }
ACLCHECK(aclrtSynchronizeStream(stream));

    if (ctx->device / 4 == 1) {
        void* resultBuff;
        ACLCHECK(aclrtMallocHost((void**)&resultBuff, mallocSize));
        ACLCHECK(aclrtMemcpy((void*)resultBuff, mallocSize, (void*)recvBuff, mallocSize,
ACL_MEMCPY_DEVICE_TO_HOST));
        float* tmpResBuff = static_cast<float*>(resultBuff);
        for (uint32_t i = 0; i < count; ++i) {
            std::cout <<  "rankId:" << ctx->device << ",i" << i << " " << tmpResBuff[i] << std::endl;
        }
        ACLCHECK(aclrtFreeHost(resultBuff));
    }
    // 释放通信用sendBuf、recvBuf内存，stream等资源
    ACLCHECK(aclrtFreeHost(hostBuff));
    ACLCHECK(aclrtFree(recvBuff));
    ACLCHECK(aclrtFree(sendBuff));
    ACLCHECK(aclrtDestroyStream(stream));
    ACLCHECK(aclrtResetDevice(ctx->device));
    return 0;
}
int main()
{
    MPI_Init(NULL, NULL);
    int procSize = 0;
    int procRank = 0;
    // 获取当前进程在所属进程组的编号
    MPI_Comm_size(MPI_COMM_WORLD, &procSize);
    MPI_Comm_rank(MPI_COMM_WORLD, &procRank);
    int devId = procRank;
    int devCount = procSize;
    // 设备资源初始化
    ACLCHECK(aclInit(NULL));
    // 获取ranktable路径
    char* rankTableFile = getenv("RANK_TABLE_FILE");
    // 指定集合通信操作使用的设备
    ACLCHECK(aclrtSetDevice(devId));
    HcclComm hcclComm;
    HcclCommInitClusterInfo(rankTableFile, devId, &hcclComm);
    struct ThreadContext args;
    args.comm = hcclComm;
    args.device = devId;
    Sample((void *)&args);
    HCCLCHECK(HcclCommDestroy(hcclComm));
    // 设备资源去初始化
    ACLCHECK(aclFinalize());
    MPI_Finalize();
    return 0;
}
```

8.5.2.2 HcclCommInitClusterInfoConfig 初始化方式

该样例仅支持单机8卡的组网。

准备 ranktable 文件

该样例通过获取ranktable的方式进行初始化，所以需准备一份ranktable文件配置集群信息，供后续调用接口时使用。

配置“RANK_TABLE_FILE”环境变量，指定ranktable文件所在路径，如下所示，文件名称为“ranktable.json”。

```
export RANK_TABLE_FILE=/home/test/ranktable.json
```

以**Atlas A2 训练系列产品**为例，ranktable.json配置示例如下，不同产品形态ranktable文件的配置示例及详细参数说明可参见**7.2 ranktable文件配置资源信息**。

```
{
      "status":"completed",  // ranktable可用标识，completed为可用
      "version": "1.0",
      "server_count": "1",  // 参与训练的AI Server数目
      "server_list": [{
            "server_id": "SERVER_ID_SV1",  // AI Server标识，String类型，请确保全局唯一
            "device": [{          // AI Server中的Device列表
                  "device_id": "0",
                  "device_ip": "192.168.1.8",
                  "rank_id": "0"
            },
            {
                  "device_id": "1",
                  "device_ip": "192.168.1.9",
                  "rank_id": "1"
            },
            {
                  "device_id": "2",
                  "device_ip": "192.168.1.10",
                  "rank_id": "2"
            },
            {
                  "device_id": "3",
                  "device_ip": "192.168.1.10",
                  "rank_id": "3"
            },
            {
                  "device_id": "4",
                  "device_ip": "192.168.1.10",
                  "rank_id": "4"
            },
            {
                  "device_id": "5",
                  "device_ip": "192.168.1.10",
                  "rank_id": "5"
            },
            {
                  "device_id": "6",
                  "device_ip": "192.168.1.10",
                  "rank_id": "6"
            },
            {
                  "device_id": "7",
                  "device_ip": "192.168.1.11",
                  "rank_id": "7"
            }]
      }]
}
```

代码示例

```
#include <iostream>
#include <vector>
#include <memory>
#include <thread>
#include <chrono>
#include <cstring>
#include "hccl/hccl.h"
#include "hccl/hccl_types.h"
#include "mpi.h"
```



```
#define ACLCHECK(ret) do { \
    if(ret != ACL_SUCCESS)\
    {\
        printf("acl interface return err %s:%d, retcode: %d \n", __FILE__, __LINE__, ret);\
        return ret;\
    }\
} while(0)

#define HCCLCHECK(ret) do {  \
    if(ret != HCCL_SUCCESS) \
    {   \
        printf("hccl interface return errreturn err %s:%d, retcode: %d \n", __FILE__, __LINE__, ret); \
        return ret;\
    } \
} while(0)

struct ThreadContext {
    HcclComm comm;
    int32_t device;
};
int Sample(void *arg)
{
    ThreadContext* ctx = (ThreadContext *)arg;
    // 申请通信用device、sendBuf，recvBuf内存、stream等资源
    ACLCHECK(aclrtSetDevice(ctx->device));
    aclrtStream stream;
    ACLCHECK(aclrtCreateStream(&stream));
    void* sendBuff;
    void* recvBuff;
    void* hostBuff;
    uint64_t count = 8;
    int mallocSize = count * sizeof(float);
     //初始化输入内存
    ACLCHECK(aclrtMallocHost((void**)&hostBuff, mallocSize));
    float* tmpHostBuff = static_cast<float*>(hostBuff);
    for (uint32_t i = 0; i < count; ++i) {
        tmpHostBuff[i] = 2;
    }
    ACLCHECK(aclrtMalloc((void**)&sendBuff, mallocSize, ACL_MEM_MALLOC_HUGE_FIRST));
    ACLCHECK(aclrtMemcpy((void*)sendBuff, mallocSize, (void*)hostBuff, mallocSize,
ACL_MEMCPY_HOST_TO_DEVICE));
    ACLCHECK(aclrtMalloc((void**)&recvBuff, mallocSize, ACL_MEM_MALLOC_HUGE_FIRST));
    // 执行SendRecv操作
    if (ctx->device / 4 == 0) {
        HCCLCHECK(HcclSend(sendBuff, count, HCCL_DATA_TYPE_FP32, ctx->device + 4, ctx->comm, stream));
    } else {
        HCCLCHECK(HcclRecv(recvBuff, count, HCCL_DATA_TYPE_FP32, ctx->device - 4, ctx->comm, stream));
    }
ACLCHECK(aclrtSynchronizeStream(stream));

    if (ctx->device / 4 == 1) {
        void* resultBuff;
        ACLCHECK(aclrtMallocHost((void**)&resultBuff, mallocSize));
        ACLCHECK(aclrtMemcpy((void*)resultBuff, mallocSize, (void*)recvBuff, mallocSize,
ACL_MEMCPY_DEVICE_TO_HOST));
        float* tmpResBuff = static_cast<float*>(resultBuff);
        for (uint32_t i = 0; i < count; ++i) {
            std::cout <<  "rankId:" << ctx->device << ",i" << i << " " << tmpResBuff[i] << std::endl;
        }
        ACLCHECK(aclrtFreeHost(resultBuff));
    }
    // 释放通信用sendBuf、recvBuf内存，stream等资源
    ACLCHECK(aclrtFreeHost(hostBuff));
    ACLCHECK(aclrtFree(recvBuff));
    ACLCHECK(aclrtFree(sendBuff));
    ACLCHECK(aclrtDestroyStream(stream));
    ACLCHECK(aclrtResetDevice(ctx->device));
    return 0;
}
```



```
int main()
{
    MPI_Init(NULL, NULL);
    int procSize = 0;
    int procRank = 0;
    // 获取当前进程在所属进程组的编号
    MPI_Comm_size(MPI_COMM_WORLD, &procSize);
    MPI_Comm_rank(MPI_COMM_WORLD, &procRank);
    int devId = procRank;
    int devCount = procSize;
    // 设备资源初始化
    ACLCHECK(aclInit(NULL));
    // 获取ranktable路径
    char* rankTableFile = getenv("RANK_TABLE_FILE");
    // 指定集合通信操作使用的设备
    ACLCHECK(aclrtSetDevice(devId));

    // 创建并初始化通信域配置项
    HcclCommConfig config;
    HcclCommConfigInit(&config);
    // 根据需要修改通信域配置
    config.hcclBufferSize = 50;
    strcpy(config.hcclCommName, "comm_1");

    HcclComm hcclComm;
    HcclCommInitClusterInfoConfig(rankTableFile, devId, &config, &hcclComm);
    struct ThreadContext args;
    args.comm = hcclComm;
    args.device = devId;
    Sample((void *)&args);
    HCCLCHECK(HcclCommDestroy(hcclComm));
    // 设备资源去初始化
    ACLCHECK(aclFinalize());
    MPI_Finalize();
    return 0;
}
```

8.5.2.3 HcclCommInitRootInfo 初始化方式

该样例仅支持单机8卡的组网。

```
#include <iostream>
#include <vector>
#include <memory>
#include <thread>
#include <chrono>
#include "hccl/hccl.h"
#include "hccl/hccl_types.h"
#include "mpi.h"

#define ACLCHECK(ret) do {\
    if(ret != ACL_SUCCESS)\
    {\
        printf("acl interface return err %s:%d, retcode: %d \n", __FILE__, __LINE__, ret);\
        return ret;\
    }\
} while(0)\

#define HCCLCHECK(ret) do {\
    if(ret != HCCL_SUCCESS)\
    {\
        printf("hccl interface return errreturn err %s:%d, retcode: %d \n", __FILE__, __LINE__, ret);\
        return ret;\
    }\
} while(0)

struct ThreadContext {
    HcclComm comm;
    int32_t device;
```



```
};
int Sample(void *arg)
{
    ThreadContext* ctx = (ThreadContext *)arg;
    // 申请通信用device、sendBuf，recvBuf内存、stream等资源
    ACLCHECK(aclrtSetDevice(ctx->device));
    aclrtStream stream;
    ACLCHECK(aclrtCreateStream(&stream));
    void* sendBuff;
    void* recvBuff;
    void* hostBuff;
    uint64_t count = 8;
    int mallocSize = count * sizeof(float);
     //初始化输入内存
    ACLCHECK(aclrtMallocHost((void**)&hostBuff, mallocSize));
    float* tmpHostBuff = static_cast<float*>(hostBuff);
    for (uint32_t i = 0; i < count; ++i) {
        tmpHostBuff[i] = 2;
    }
    ACLCHECK(aclrtMalloc((void**)&sendBuff, mallocSize, ACL_MEM_MALLOC_HUGE_FIRST));
    ACLCHECK(aclrtMemcpy((void*)sendBuff, mallocSize, (void*)hostBuff, mallocSize,
ACL_MEMCPY_HOST_TO_DEVICE));
    ACLCHECK(aclrtMalloc((void**)&recvBuff, mallocSize, ACL_MEM_MALLOC_HUGE_FIRST));
    //执行SendRecv操作
    if (ctx->device / 4 == 0) {
        HCCLCHECK(HcclSend(sendBuff, count, HCCL_DATA_TYPE_FP32, ctx->device + 4, ctx->comm, stream));
    } else {
        HCCLCHECK(HcclRecv(recvBuff, count, HCCL_DATA_TYPE_FP32, ctx->device - 4, ctx->comm, stream));
    }

    ACLCHECK(aclrtSynchronizeStream(stream));
    if (ctx->device / 4 == 1) {
        void* resultBuff;
        ACLCHECK(aclrtMallocHost((void**)&resultBuff, mallocSize));
        ACLCHECK(aclrtMemcpy((void*)resultBuff, mallocSize, (void*)recvBuff, mallocSize,
ACL_MEMCPY_DEVICE_TO_HOST));
        float* tmpResBuff = static_cast<float*>(resultBuff);
        for (uint32_t i = 0; i < count; ++i) {
            std::cout <<  "rankId:" << ctx->device << ",i" << i << " " << tmpResBuff[i] << std::endl;
        }
        ACLCHECK(aclrtFreeHost(resultBuff));
    }
    // 释放通信用sendBuf、recvBuf内存，stream等资源
    ACLCHECK(aclrtFreeHost(hostBuff));
    ACLCHECK(aclrtFree(recvBuff));
    ACLCHECK(aclrtFree(sendBuff));
    ACLCHECK(aclrtDestroyStream(stream));
    ACLCHECK(aclrtResetDevice(ctx->device));
    HCCLCHECK(HcclCommDestroy(ctx->comm));
    return 0;
}
int main() {
    MPI_Init(NULL, NULL);
    int procSize = 0;
    int procRank = 0;
    // 获取当前进程在所属进程组的编号
    MPI_Comm_size(MPI_COMM_WORLD, &procSize);
    MPI_Comm_rank(MPI_COMM_WORLD, &procRank);
    int devId = procRank;
    int devCount = procSize;
    //设备资源初始化
    ACLCHECK(aclInit(NULL));
    // 指定集合通信操作使用的设备
    ACLCHECK(aclrtSetDevice(devId));
    // 在 rootRank 获取 rootInfo
    HcclRootInfo rootInfo;
    int32_t rootRank = 0;
    if(devId == rootRank) {
        HCCLCHECK(HcclGetRootInfo(&rootInfo));
```



```
    }
    // 将root_info广播到通信域内的其他rank
    MPI_Bcast(&rootInfo, HCCL_ROOT_INFO_BYTES, MPI_CHAR, rootRank, MPI_COMM_WORLD);
    MPI_Barrier(MPI_COMM_WORLD);
    // 初始化集合通信域
    HcclComm hcclComm;
    HCCLCHECK(HcclCommInitRootInfo(devCount, &rootInfo, devId, &hcclComm));
    struct ThreadContext args;
    args.comm = hcclComm;
    args.device = devId;
    Sample((void *)&args);
    // 设备资源去初始化
    ACLCHECK(aclFinalize());
    MPI_Finalize();
    return 0;
}
```

8.5.2.4 HcclCommInitRootInfoConfig 初始化方式

该样例仅支持单机8卡的组网。

```
#include <iostream>
#include <vector>
#include <memory>
#include <thread>
#include <chrono>
#include "hccl/hccl.h"
#include "hccl/hccl_types.h"
#include "mpi.h"

#define ACLCHECK(ret) do {\
    if(ret != ACL_SUCCESS)\
    {\
        printf("acl interface return err %s:%d, retcode: %d \n", __FILE__, __LINE__, ret);\
        return ret;\
    }\
} while(0)\

#define HCCLCHECK(ret) do {\
    if(ret != HCCL_SUCCESS)\
    {\
        printf("hccl interface return errreturn err %s:%d, retcode: %d \n", __FILE__, __LINE__, ret);\
        return ret;\
    }\
} while(0)

struct ThreadContext {
    HcclComm comm;
    int32_t device;
};
int Sample(void *arg)
{
    ThreadContext* ctx = (ThreadContext *)arg;
    // 申请通信用device、sendBuf，recvBuf内存、stream等资源
    ACLCHECK(aclrtSetDevice(ctx->device));
    aclrtStream stream;
    ACLCHECK(aclrtCreateStream(&stream));
    void* sendBuff;
    void* recvBuff;
    void* hostBuff;
    uint64_t count = 8;
    int mallocSize = count * sizeof(float);
     //初始化输入内存
    ACLCHECK(aclrtMallocHost((void**)&hostBuff, mallocSize));
    float* tmpHostBuff = static_cast<float*>(hostBuff);
    for (uint32_t i = 0; i < count; ++i) {
        tmpHostBuff[i] = 2;
    }
    ACLCHECK(aclrtMalloc((void**)&sendBuff, mallocSize, ACL_MEM_MALLOC_HUGE_FIRST));
```



```
    ACLCHECK(aclrtMemcpy((void*)sendBuff, mallocSize, (void*)hostBuff, mallocSize,
ACL_MEMCPY_HOST_TO_DEVICE));
    ACLCHECK(aclrtMalloc((void**)&recvBuff, mallocSize, ACL_MEM_MALLOC_HUGE_FIRST));
    //执行SendRecv操作
    if (ctx->device / 4 == 0) {
        HCCLCHECK(HcclSend(sendBuff, count, HCCL_DATA_TYPE_FP32, ctx->device + 4, ctx->comm, stream));
    } else {
        HCCLCHECK(HcclRecv(recvBuff, count, HCCL_DATA_TYPE_FP32, ctx->device - 4, ctx->comm, stream));
    }

    ACLCHECK(aclrtSynchronizeStream(stream));
    if (ctx->device / 4 == 1) {
        void* resultBuff;
        ACLCHECK(aclrtMallocHost((void**)&resultBuff, mallocSize));
        ACLCHECK(aclrtMemcpy((void*)resultBuff, mallocSize, (void*)recvBuff, mallocSize,
ACL_MEMCPY_DEVICE_TO_HOST));
        float* tmpResBuff = static_cast<float*>(resultBuff);
        for (uint32_t i = 0; i < count; ++i) {
            std::cout <<  "rankId:" << ctx->device << ",i" << i << " " << tmpResBuff[i] << std::endl;
        }
        ACLCHECK(aclrtFreeHost(resultBuff));
    }
    // 释放通信用sendBuf、recvBuf内存，stream等资源
    ACLCHECK(aclrtFreeHost(hostBuff));
    ACLCHECK(aclrtFree(recvBuff));
    ACLCHECK(aclrtFree(sendBuff));
    ACLCHECK(aclrtDestroyStream(stream));
    ACLCHECK(aclrtResetDevice(ctx->device));
    HCCLCHECK(HcclCommDestroy(ctx->comm));
    return 0;
}
int main() {
    MPI_Init(NULL, NULL);
    int procSize = 0;
    int procRank = 0;
    // 获取当前进程在所属进程组的编号
    MPI_Comm_size(MPI_COMM_WORLD, &procSize);
    MPI_Comm_rank(MPI_COMM_WORLD, &procRank);
    int devId = procRank;
    int devCount = procSize;
    //设备资源初始化
    ACLCHECK(aclInit(NULL));
    // 指定集合通信操作使用的设备
    ACLCHECK(aclrtSetDevice(devId));
    // 在 rootRank 获取 rootInfo
    HcclRootInfo rootInfo;
    int32_t rootRank = 0;
    if(devId == rootRank) {
        HCCLCHECK(HcclGetRootInfo(&rootInfo));
    }
    // 将root_info广播到通信域内的其他rank
    MPI_Bcast(&rootInfo, HCCL_ROOT_INFO_BYTES, MPI_CHAR, rootRank, MPI_COMM_WORLD);
    MPI_Barrier(MPI_COMM_WORLD);

    // 创建并初始化通信域配置项
    HcclCommConfig config;
    HcclCommConfigInit(&config);

    // 根据需要修改通信域配置
    config.hcclBufferSize = 50;
    strcpy(config.hcclCommName, "comm_1");

    // 初始化集合通信域
    HcclComm hcclComm;
    HCCLCHECK(HcclCommInitRootInfoConfig(devCount, &rootInfo, devId, &config, &hcclComm));
    struct ThreadContext args;
    args.comm = hcclComm;
    args.device = devId;
    Sample((void *)&args);
```



```
    // 设备资源去初始化
    ACLCHECK(aclFinalize());
    MPI_Finalize();
    return 0;
}
```

8.5.2.5 HcclCommInitAll 初始化

该样例仅支持单机8卡的组网，且仅支持单进程方式拉起。

```
#include <iostream>
#include <vector>
#include <memory>
#include <thread>
#include <chrono>
#include "hccl/hccl.h"
#include "hccl/hccl_types.h"
#include "mpi.h"

#define ACLCHECK(ret) do { \
    if(ret != ACL_SUCCESS)\
    {\
        printf("acl interface return err %s:%d, retcode: %d \n", __FILE__, __LINE__, ret);\
        return ret;\
    }\
} while(0)

#define HCCLCHECK(ret) do {  \
    if(ret != HCCL_SUCCESS) \
    {   \
        printf("hccl interface return errreturn err %s:%d, retcode: %d \n", __FILE__, __LINE__, ret); \
        return ret;\
    } \
} while(0)

struct ThreadContext {
    HcclComm comm;
    int32_t device;
};
int Sample(void *arg)
{
    ThreadContext* ctx = (ThreadContext *)arg;
    // 申请通信用device、sendBuf，recvBuf内存、stream等资源
    ACLCHECK(aclrtSetDevice(ctx->device));
    aclrtStream stream;
    ACLCHECK(aclrtCreateStream(&stream));
    void* sendBuff;
    void* recvBuff;
    void* hostBuff;
    uint64_t count = 8;
    int mallocSize = count * sizeof(float);
     //初始化输入内存
    ACLCHECK(aclrtMallocHost((void**)&hostBuff, mallocSize));
    float* tmpHostBuff = static_cast<float*>(hostBuff);
    for (uint32_t i = 0; i < count; ++i) {
        tmpHostBuff[i] = 2;
    }
    ACLCHECK(aclrtMalloc((void**)&sendBuff, mallocSize, ACL_MEM_MALLOC_HUGE_FIRST));
    ACLCHECK(aclrtMemcpy((void*)sendBuff, mallocSize, (void*)hostBuff, mallocSize,
ACL_MEMCPY_HOST_TO_DEVICE));
    ACLCHECK(aclrtMalloc((void**)&recvBuff, mallocSize, ACL_MEM_MALLOC_HUGE_FIRST));
    // 执行SendRecv操作
    if (ctx->device / 4 == 0) {
    HCCLCHECK(HcclSend(sendBuff, count, HCCL_DATA_TYPE_FP32, ctx->device + 4, ctx->comm, stream));
    } else {
        HCCLCHECK(HcclRecv(recvBuff, count, HCCL_DATA_TYPE_FP32, ctx->device - 4, ctx->comm, stream));
    }

    ACLCHECK(aclrtSynchronizeStream(stream));
```



```
    if (ctx->device / 4 == 1) {
        void* resultBuff;
        ACLCHECK(aclrtMallocHost((void**)&resultBuff, mallocSize));
        ACLCHECK(aclrtMemcpy((void*)resultBuff, mallocSize, (void*)recvBuff, mallocSize,
ACL_MEMCPY_DEVICE_TO_HOST));
        float* tmpResBuff = static_cast<float*>(resultBuff);
        for (uint32_t i = 0; i < count; ++i) {
            std::cout <<  "rankId:" << ctx->device << ",i" << i << " " << tmpResBuff[i] << std::endl;
        }
        ACLCHECK(aclrtFreeHost(resultBuff));
    }

    // 释放通信用sendBuf、recvBuf内存，stream等资源
    ACLCHECK(aclrtFreeHost(hostBuff));
    ACLCHECK(aclrtFree(recvBuff));
    ACLCHECK(aclrtFree(sendBuff));
    ACLCHECK(aclrtDestroyStream(stream));
    ACLCHECK(aclrtResetDevice(ctx->device));
    HCCLCHECK(HcclCommDestroy(ctx->comm));
    return 0;
}
int main() {
    // 设备资源初始化
    ACLCHECK(aclInit(NULL));
    uint32_t ndev = 8;
    int32_t devices[8] = {0, 1, 2, 3, 4, 5, 6, 7};
    HcclComm comms[ndev];
    for (int32_t i = 0; i < ndev; i++) {
        ACLCHECK(aclrtSetDevice(devices[i]));
    }
    // 初始化通信域
    HCCLCHECK(HcclCommInitAll(ndev, devices, comms));

    // 启动线程执行集合通信操作
    std::vector<std::unique_ptr<std::thread> > threads(ndev);
    struct ThreadContext args[ndev];
    for (uint32_t i = 0; i < ndev; i++) {
        args[i].device = i;
        args[i].comm = comms[i];
        threads[i].reset(new (std::nothrow) std::thread(&Sample, (void *)&args[i]));
    }

    for (uint32_t i = 0; i < ndev; i++) {
        threads[i]->join();
    }

    // 设备资源去初始化
    ACLCHECK(aclFinalize());
    return 0;
}
```

8.5.2.6 HcclCreateSubCommConfig 方式创建子通信域

该样例仅支持从单机N卡的组网中切分出1个4卡子通信域，N需要大于等于4且小于等于8。实际执行节点仅限于属于子通信域的4张卡，属于组网但不属于子通信域的节点不可执行该用例。

全局通信域可以基于ranktable文件或者root info协商方式创建，该样例以基于ranktable文件创建全局通信域为例。

准备 ranktable 文件

该样例中全局通信域通过获取ranktable的方式进行初始化，所以需准备一份ranktable文件配置集群信息，供后续调用接口时使用。

配置“RANK_TABLE_FILE”环境变量，指定ranktable文件所在路径，如下所示，文件名称为“ranktable.json”。

```
export RANK_TABLE_FILE=/home/test/ranktable.json
```

以**Atlas A2 训练系列产品**为例，ranktable.json配置示例如下，不同产品形态ranktable文件的配置示例及详细参数说明可参见**7.2 ranktable文件配置资源信息**。

```
{
      "status":"completed",  // ranktable可用标识，completed为可用
      "version": "1.0",
      "server_count": "1",  // 参与训练的AI Server数目
      "server_list": [{
            "server_id": "SERVER_ID_SV1",  // AI Server标识，String类型，请确保全局唯一
            "device": [{          // AI Server中的Device列表
                  "device_id": "0",
                  "device_ip": "192.168.1.8",
                  "rank_id": "0"
            },
            {
                  "device_id": "1",
                  "device_ip": "192.168.1.9",
                  "rank_id": "1"
            },
            {
                  "device_id": "2",
                  "device_ip": "192.168.1.10",
                  "rank_id": "2"
            },
            {
                  "device_id": "3",
                  "device_ip": "192.168.1.10",
                  "rank_id": "3"
            },
            {
                  "device_id": "4",
                  "device_ip": "192.168.1.10",
                  "rank_id": "4"
            },
            {
                  "device_id": "5",
                  "device_ip": "192.168.1.10",
                  "rank_id": "5"
            },
            {
                  "device_id": "6",
                  "device_ip": "192.168.1.10",
                  "rank_id": "6"
            },
            {
                  "device_id": "7",
                  "device_ip": "192.168.1.11",
                  "rank_id": "7"
            }]
      }]
}
```

代码示例

```
#include <iostream>
#include <vector>
#include <memory>
#include <thread>
#include <chrono>
#include <cstring>
#include "hccl/hccl.h"
#include "hccl/hccl_types.h"
#include "mpi.h"
```



```
#define ACLCHECK(ret) do { \
    if(ret != ACL_SUCCESS)\
    {\
        printf("acl interface return err %s:%d, retcode: %d \n", __FILE__, __LINE__, ret);\
        return ret;\
    }\
} while(0)

#define HCCLCHECK(ret) do {  \
    if(ret != HCCL_SUCCESS) \
    {   \
        printf("hccl interface return errreturn err %s:%d, retcode: %d \n", __FILE__, __LINE__, ret); \
        return ret;\
    } \
} while(0)

struct ThreadContext {
    HcclComm comm;
    int32_t device;
};
int Sample(void *arg)
{
    ThreadContext* ctx = (ThreadContext *)arg;
    // 申请通信用device、sendBuf，recvBuf内存、stream等资源
    ACLCHECK(aclrtSetDevice(ctx->device));
    aclrtStream stream;
    ACLCHECK(aclrtCreateStream(&stream));
    void* sendBuff;
    void* recvBuff;
    void* hostBuff;
    uint64_t count = 4;
    int mallocSize = count * sizeof(float);
     //初始化输入内存
    ACLCHECK(aclrtMallocHost((void**)&hostBuff, mallocSize));
    float* tmpHostBuff = static_cast<float*>(hostBuff);
    for (uint32_t i = 0; i < count; ++i) {
        tmpHostBuff[i] = 2;
    }
    ACLCHECK(aclrtMalloc((void**)&sendBuff, mallocSize, ACL_MEM_MALLOC_HUGE_FIRST));
    ACLCHECK(aclrtMemcpy((void*)sendBuff, mallocSize, (void*)hostBuff, mallocSize,
ACL_MEMCPY_HOST_TO_DEVICE));
    ACLCHECK(aclrtMalloc((void**)&recvBuff, mallocSize, ACL_MEM_MALLOC_HUGE_FIRST));
    // 执行SendRecv操作
    if (ctx->device / 2 == 0) {
        HCCLCHECK(HcclSend(sendBuff, count, HCCL_DATA_TYPE_FP32, ctx->device + 2, ctx->comm, stream));
    } else {
        HCCLCHECK(HcclRecv(recvBuff, count, HCCL_DATA_TYPE_FP32, ctx->device - 2, ctx->comm, stream));
    }
    ACLCHECK(aclrtSynchronizeStream(stream));

    if (ctx->device / 2 == 1) {
        void* resultBuff;
        ACLCHECK(aclrtMallocHost((void**)&resultBuff, mallocSize));
        ACLCHECK(aclrtMemcpy((void*)resultBuff, mallocSize, (void*)recvBuff, mallocSize,
ACL_MEMCPY_DEVICE_TO_HOST));
        float* tmpResBuff = static_cast<float*>(resultBuff);
        for (uint32_t i = 0; i < count; ++i) {
            std::cout <<  "rankId:" << ctx->device << ",i" << i << " " << tmpResBuff[i] << std::endl;
        }
        ACLCHECK(aclrtFreeHost(resultBuff));
    }
    // 释放通信用sendBuf、recvBuf内存，stream等资源
    ACLCHECK(aclrtFreeHost(hostBuff));
    ACLCHECK(aclrtFree(recvBuff));
    ACLCHECK(aclrtFree(sendBuff));
    ACLCHECK(aclrtDestroyStream(stream));
    ACLCHECK(aclrtResetDevice(ctx->device));
    return 0;
}
```



```
int main()
{
    MPI_Init(NULL, NULL);
    int procSize = 0;
    int procRank = 0;
    // 获取当前进程在所属进程组的编号
    MPI_Comm_size(MPI_COMM_WORLD, &procSize);
    MPI_Comm_rank(MPI_COMM_WORLD, &procRank);
    int devId = procRank;
    int devCount = procSize;
    // 设备资源初始化
    ACLCHECK(aclInit(NULL));
    // 获取ranktable路径
    char* rankTableFile = getenv("RANK_TABLE_FILE");
    // 指定集合通信操作使用的设备
    ACLCHECK(aclrtSetDevice(devId));

    // 创建并初始化通信域配置项
    HcclCommConfig config;
    HcclCommConfigInit(&config);
    // 根据需要修改通信域配置
    config.hcclBufferSize = 50;
    strcpy(config.hcclCommName, "comm_1");

    HcclComm globalHcclComm;
    HcclCommInitClusterInfoConfig(rankTableFile, devId, &config, &globalHcclComm);
    HcclComm hcclComm;
    strcpy(config.hcclCommName, "comm_2");
    uint32_t rankIds[4] = {0, 1, 2, 3};
    HCCLCHECK(HcclCreateSubCommConfig(&globalHcclComm, 4, rankIds, 1, devId, &config, &hcclComm));
    struct ThreadContext args;
    args.comm = hcclComm;
    args.device = devId;
    Sample((void *)&args);
    HCCLCHECK(HcclCommDestroy(hcclComm));
    // 设备资源去初始化
    ACLCHECK(aclFinalize());
    MPI_Finalize();
    return 0;
}
```

8.5.3 集合通信样例（HcclAllReduce）

8.5.3.1 HcclCommInitClusterInfo 初始化方式

该样例支持单机N卡的组网，N需要小于等于8。

准备 ranktable 文件

该样例通过获取ranktable的方式进行初始化，所以需准备一份ranktable文件配置集群信息，供后续调用接口时使用。

配置“RANK_TABLE_FILE”环境变量，指定ranktable文件所在路径，如下所示，文件名称为“ranktable.json”。

```
export RANK_TABLE_FILE=/home/test/ranktable.json
```

以**Atlas A2 训练系列产品**为例，ranktable.json配置示例如下，不同产品形态ranktable文件的配置示例及详细参数说明可参见**7.2 ranktable文件配置资源信息**。

```
{
      "status":"completed",  // ranktable可用标识，completed为可用
      "version": "1.0",
      "server_count": "1",  // 参与训练的AI Server数目
      "server_list": [{
```



```
            "server_id": "SERVER_ID_SV1",   // AI Server标识，String类型，请确保全局唯一
            "device": [{          // AI Server中的Device列表
                "device_id": "0",
                "device_ip": "192.168.1.8",
                "rank_id": "0"
            },
            {
                "device_id": "1",
                "device_ip": "192.168.1.9",
                "rank_id": "1"
            },
            {
                "device_id": "2",
                "device_ip": "192.168.1.10",
                "rank_id": "2"
            },
            {
                "device_id": "3",
                "device_ip": "192.168.1.10",
                "rank_id": "3"
            },
            {
                "device_id": "4",
                "device_ip": "192.168.1.10",
                "rank_id": "4"
            },
            {
                "device_id": "5",
                "device_ip": "192.168.1.10",
                "rank_id": "5"
            },
            {
                "device_id": "6",
                "device_ip": "192.168.1.10",
                "rank_id": "6"
            },
            {
                "device_id": "7",
                "device_ip": "192.168.1.11",
                "rank_id": "7"
            }]
        }]
}
```

代码示例

```
#include <iostream>
#include <vector>
#include <memory>
#include <thread>
#include <chrono>
#include "hccl/hccl.h"
#include "hccl/hccl_types.h"
#include "mpi.h"

#define ACLCHECK(ret) do { \
    if(ret != ACL_SUCCESS)\
    {\
        printf("acl interface return err %s:%d, retcode: %d \n", __FILE__, __LINE__, ret);\
        return ret;\
    }\
} while(0)

#define HCCLCHECK(ret) do {  \
    if(ret != HCCL_SUCCESS) \
    {   \
        printf("hccl interface return errreturn err %s:%d, retcode: %d \n", __FILE__, __LINE__, ret); \
        return ret;\
    } \
```



```
} while(0)

struct ThreadContext {
    HcclComm comm;
    int32_t device;
};
int Sample(void *arg)
{
    ThreadContext* ctx = (ThreadContext *)arg;
    void* host_buf = nullptr;
    void* send_buff = nullptr;
    void* recv_buff = nullptr;
    uint64_t count = 1;
    int malloc_kSize = count * sizeof(float);
    aclrtEvent start_event, end_event;
    aclrtStream stream;
    ACLCHECK(aclrtCreateStream(&stream));
    ACLCHECK(aclrtCreateEvent(&start_event));
    ACLCHECK(aclrtCreateEvent(&end_event));

    //申请集合通信操作的内存
    ACLCHECK(aclrtMalloc((void**)&send_buff, malloc_kSize, ACL_MEM_MALLOC_HUGE_FIRST));
    ACLCHECK(aclrtMalloc((void**)&recv_buff, malloc_kSize, ACL_MEM_MALLOC_HUGE_FIRST));

    //初始化输入内存
    ACLCHECK(aclrtMallocHost((void**)&host_buf, malloc_kSize));
    ACLCHECK(aclrtMemcpy((void*)send_buff, malloc_kSize, (void*)host_buf, malloc_kSize,
ACL_MEMCPY_HOST_TO_DEVICE));

    //执行集合通信操作
    HCCLCHECK(HcclAllReduce((void *)send_buff, (void*)recv_buff, count, HCCL_DATA_TYPE_FP32,
HCCL_REDUCE_SUM, ctx->comm, stream));

    //等待stream中集合通信任务执行完成
    ACLCHECK(aclrtSynchronizeStream(stream));

    if (ctx->device < 8) {
        void* resultBuff;
        ACLCHECK(aclrtMallocHost((void**)&resultBuff, malloc_kSize));
        ACLCHECK(aclrtMemcpy((void*)resultBuff, malloc_kSize, (void*)recv_buff, malloc_kSize,
ACL_MEMCPY_DEVICE_TO_HOST));
        float* tmpResBuff = static_cast<float*>(resultBuff);
        for (uint32_t i = 0; i < count; ++i) {
            std::cout <<  "rankId:" << ctx->device << ",i" << i << " " << tmpResBuff[i] << std::endl;
        }
        ACLCHECK(aclrtFreeHost(resultBuff));
    }
    ACLCHECK(aclrtFree(send_buff));
    ACLCHECK(aclrtFree(recv_buff));
    ACLCHECK(aclrtFreeHost(host_buf));
    //销毁任务流
    ACLCHECK(aclrtDestroyStream(stream));
    ACLCHECK(aclrtDestroyEvent(start_event));
    ACLCHECK(aclrtDestroyEvent(end_event));
}

int main()
{
    MPI_Init(NULL, NULL);
    int procSize = 0;
    int procRank = 0;
    // 获取当前进程在所属进程组的编号
    MPI_Comm_size(MPI_COMM_WORLD, &procSize);
    MPI_Comm_rank(MPI_COMM_WORLD, &procRank);
    int devId = procRank;
    int devCount = procSize;
    // 设备资源初始化
    ACLCHECK(aclInit(NULL));
    // 获取ranktable路径
```



```
    char* rankTableFile = getenv("RANK_TABLE_FILE");
    // 指定集合通信操作使用的设备
    ACLCHECK(aclrtSetDevice(devId));
    HcclComm hcclComm;
    HcclCommInitClusterInfo(rankTableFile, devId, &hcclComm);
    struct ThreadContext args;
    args.comm = hcclComm;
    args.device = devId;
    Sample((void *)&args);
    HCCLCHECK(HcclCommDestroy(hcclComm));
    // 设备资源去初始化
    ACLCHECK(aclFinalize());
    MPI_Finalize();
    return 0;
}
```

8.5.3.2 HcclCommInitClusterInfoConfig 初始化方式

该样例支持单机N卡的组网，N需要小于等于8。

准备 ranktable 文件

该样例通过获取ranktable的方式进行初始化，所以需准备一份ranktable文件配置集群信息，供后续调用接口时使用。

配置“RANK_TABLE_FILE”环境变量，指定ranktable文件所在路径，如下所示，文件名称为“ranktable.json”。

```
export RANK_TABLE_FILE=/home/test/ranktable.json
```

以**Atlas A2 训练系列产品**为例，ranktable.json配置示例如下，不同产品形态ranktable文件的配置示例及详细参数说明可参见**7.2 ranktable文件配置资源信息**。

```
{
    "status":"completed",  // ranktable可用标识，completed为可用
    "version": "1.0",
    "server_count": "1",  // 参与训练的AI Server数目
    "server_list": [{
        "server_id": "SERVER_ID_SV1",  // AI Server标识，String类型，请确保全局唯一
        "device": [{          // AI Server中的Device列表
            "device_id": "0",
            "device_ip": "192.168.1.8",
            "rank_id": "0"
        },
        {
            "device_id": "1",
            "device_ip": "192.168.1.9",
            "rank_id": "1"
        },
        {
            "device_id": "2",
            "device_ip": "192.168.1.10",
            "rank_id": "2"
        },
        {
            "device_id": "3",
            "device_ip": "192.168.1.10",
            "rank_id": "3"
        },
        {
            "device_id": "4",
            "device_ip": "192.168.1.10",
            "rank_id": "4"
        },
        {
            "device_id": "5",
            "device_ip": "192.168.1.10",
```



```
                "rank_id": "5"
            },
            {
                "device_id": "6",
                "device_ip": "192.168.1.10",
                "rank_id": "6"
            },
            {
                "device_id": "7",
                "device_ip": "192.168.1.11",
                "rank_id": "7"
            }]
        }]
}
```

代码示例

```
#include <iostream>
#include <vector>
#include <memory>
#include <thread>
#include <chrono>
#include <cstring>
#include "hccl/hccl.h"
#include "hccl/hccl_types.h"
#include "mpi.h"

#define ACLCHECK(ret) do { \
    if(ret != ACL_SUCCESS)\
    {\
        printf("acl interface return err %s:%d, retcode: %d \n", __FILE__, __LINE__, ret);\
        return ret;\
    }\
} while(0)

#define HCCLCHECK(ret) do {  \
    if(ret != HCCL_SUCCESS) \
    {   \
        printf("hccl interface return errreturn err %s:%d, retcode: %d \n", __FILE__, __LINE__, ret); \
        return ret;\
    } \
} while(0)

struct ThreadContext {
    HcclComm comm;
    int32_t device;
};
int Sample(void *arg)
{
    ThreadContext* ctx = (ThreadContext *)arg;
    void* host_buf = nullptr;
    void* send_buff = nullptr;
    void* recv_buff = nullptr;
    uint64_t count = 1;
    int malloc_kSize = count * sizeof(float);
    aclrtEvent start_event, end_event;
    aclrtStream stream;
    ACLCHECK(aclrtCreateStream(&stream));
    ACLCHECK(aclrtCreateEvent(&start_event));
    ACLCHECK(aclrtCreateEvent(&end_event));

    //申请集合通信操作的内存
    ACLCHECK(aclrtMalloc((void**)&send_buff, malloc_kSize, ACL_MEM_MALLOC_HUGE_FIRST));
    ACLCHECK(aclrtMalloc((void**)&recv_buff, malloc_kSize, ACL_MEM_MALLOC_HUGE_FIRST));

    //初始化输入内存
    ACLCHECK(aclrtMallocHost((void**)&host_buf, malloc_kSize));
    ACLCHECK(aclrtMemcpy((void*)send_buff, malloc_kSize, (void*)host_buf, malloc_kSize,
ACL_MEMCPY_HOST_TO_DEVICE));
```



```
    //执行集合通信操作
    HCCLCHECK(HcclAllReduce((void *)send_buff, (void*)recv_buff, count, HCCL_DATA_TYPE_FP32,
HCCL_REDUCE_SUM, ctx->comm, stream));
    //等待stream中集合通信任务执行完成
    ACLCHECK(aclrtSynchronizeStream(stream));

    if (ctx->device < 8) {
        void* resultBuff;
        ACLCHECK(aclrtMallocHost((void**)&resultBuff, malloc_kSize));
        ACLCHECK(aclrtMemcpy((void*)resultBuff, malloc_kSize, (void*)recv_buff, malloc_kSize,
ACL_MEMCPY_DEVICE_TO_HOST));
        float* tmpResBuff = static_cast<float*>(resultBuff);
        for (uint32_t i = 0; i < count; ++i) {
            std::cout <<  "rankId:" << ctx->device << ",i" << i << " " << tmpResBuff[i] << std::endl;
        }
        ACLCHECK(aclrtFreeHost(resultBuff));
    }

    ACLCHECK(aclrtFree(send_buff));
    ACLCHECK(aclrtFree(recv_buff));
    ACLCHECK(aclrtFreeHost(host_buf));
    //销毁任务流
    ACLCHECK(aclrtDestroyStream(stream));
    ACLCHECK(aclrtDestroyEvent(start_event));
    ACLCHECK(aclrtDestroyEvent(end_event));
    return 0;
}

int main()
{
    MPI_Init(NULL, NULL);
    int procSize = 0;
    int procRank = 0;
    // 获取当前进程在所属进程组的编号
    MPI_Comm_size(MPI_COMM_WORLD, &procSize);
    MPI_Comm_rank(MPI_COMM_WORLD, &procRank);
    int devId = procRank;
    int devCount = procSize;
    // 设备资源初始化
    ACLCHECK(aclInit(NULL));
    // 获取ranktable路径
    char* rankTableFile = getenv("RANK_TABLE_FILE");
    // 指定集合通信操作使用的设备
    ACLCHECK(aclrtSetDevice(devId));

    // 创建并初始化通信域配置项
    HcclCommConfig config;
    HcclCommConfigInit(&config);
    // 根据需要修改通信域配置
    config.hcclBufferSize = 50;
    strcpy(config.hcclCommName, "comm_1");

    HcclComm hcclComm;
    HcclCommInitClusterInfoConfig(rankTableFile, devId, &config, &hcclComm);
    struct ThreadContext args;
    args.comm = hcclComm;
    args.device = devId;
    Sample((void *)&args);
    HCCLCHECK(HcclCommDestroy(hcclComm));
    // 设备资源去初始化
    ACLCHECK(aclFinalize());
    MPI_Finalize();
    return 0;
}
```

8.5.3.3 HcclCommInitRootInfo 初始化方式

该样例支持单机N卡的组网，N需要小于等于8。


```
#include <iostream>
#include <vector>
#include <memory>
#include <thread>
#include <chrono>
#include "hccl/hccl.h"
#include "hccl/hccl_types.h"
#include "mpi.h"

#define ACLCHECK(ret) do {\
    if(ret != ACL_SUCCESS)\
    {\
        printf("acl interface return err %s:%d, retcode: %d \n", __FILE__, __LINE__, ret);\
        return ret;\
    }\
} while(0)\

#define HCCLCHECK(ret) do {\
    if(ret != HCCL_SUCCESS)\
    {\
        printf("hccl interface return errreturn err %s:%d, retcode: %d \n", __FILE__, __LINE__, ret);\
        return ret;\
    }\
} while(0)

struct ThreadContext {
    HcclComm comm;
    int32_t device;
};
int Sample(void *arg)
{
    ThreadContext* ctx = (ThreadContext *)arg;
    void* host_buf = nullptr;
    void* send_buff = nullptr;
    void* recv_buff = nullptr;
    uint64_t count = 1;
    int malloc_kSize = count * sizeof(float);
    aclrtEvent start_event, end_event;
    aclrtStream stream;
    ACLCHECK(aclrtCreateStream(&stream));
    ACLCHECK(aclrtCreateEvent(&start_event));
    ACLCHECK(aclrtCreateEvent(&end_event));

    //申请集合通信操作的内存
    ACLCHECK(aclrtMalloc((void**)&send_buff, malloc_kSize, ACL_MEM_MALLOC_HUGE_FIRST));
    ACLCHECK(aclrtMalloc((void**)&recv_buff, malloc_kSize, ACL_MEM_MALLOC_HUGE_FIRST));

    //初始化输入内存
    ACLCHECK(aclrtMallocHost((void**)&host_buf, malloc_kSize));
    ACLCHECK(aclrtMemcpy((void*)send_buff, malloc_kSize, (void*)host_buf, malloc_kSize,
ACL_MEMCPY_HOST_TO_DEVICE));

    //执行集合通信操作
    HCCLCHECK(HcclAllReduce((void *)send_buff, (void*)recv_buff, count, HCCL_DATA_TYPE_FP32,
HCCL_REDUCE_SUM, ctx->comm, stream));

    //等待stream中集合通信任务执行完成
    ACLCHECK(aclrtSynchronizeStream(stream));

    if (ctx->device < 8) {
        void* resultBuff;
        ACLCHECK(aclrtMallocHost((void**)&resultBuff, malloc_kSize));
        ACLCHECK(aclrtMemcpy((void*)resultBuff, malloc_kSize, (void*)recv_buff, malloc_kSize,
ACL_MEMCPY_DEVICE_TO_HOST));
        float* tmpResBuff = static_cast<float*>(resultBuff);
        for (uint32_t i = 0; i < count; ++i) {
            std::cout <<  "rankId:" << ctx->device << ",i" << i << " " << tmpResBuff[i] << std::endl;
        }
        ACLCHECK(aclrtFreeHost(resultBuff));
```



```
    }
    ACLCHECK(aclrtFree(send_buff));
    ACLCHECK(aclrtFree(recv_buff));
    ACLCHECK(aclrtFreeHost(host_buf));
    //销毁任务流
    ACLCHECK(aclrtDestroyStream(stream));
    ACLCHECK(aclrtDestroyEvent(start_event));
    ACLCHECK(aclrtDestroyEvent(end_event));
    return 0;
}
int main(int argc, char*argv[])
{
    MPI_Init(&argc, &argv);
    int procSize = 0;
    int procRank = 0;
    // 获取当前进程在所属进程组的编号
    MPI_Comm_size(MPI_COMM_WORLD, &procSize);
    MPI_Comm_rank(MPI_COMM_WORLD, &procRank);
    int devId = procRank;
    int devCount = procSize;
    //设备资源初始化
    ACLCHECK(aclInit(NULL));
    // 指定集合通信操作使用的设备
    ACLCHECK(aclrtSetDevice(devId));
    // 在 rootRank 获取 rootInfo
    HcclRootInfo rootInfo;
    int32_t rootRank = 0;
    if(devId == rootRank) {
        HCCLCHECK(HcclGetRootInfo(&rootInfo));
    }
    // 将root_info广播到通信域内的其他rank
    MPI_Bcast(&rootInfo, HCCL_ROOT_INFO_BYTES, MPI_CHAR, rootRank, MPI_COMM_WORLD);
    MPI_Barrier(MPI_COMM_WORLD);
    // 初始化集合通信域
    HcclComm hcclComm;
    HCCLCHECK(HcclCommInitRootInfo(devCount, &rootInfo, devId, &hcclComm));
    // 创建任务stream
    struct ThreadContext args;
    args.comm = hcclComm;
    args.device = devId;
    Sample((void *)&args);
    //销毁集合通信域
    HCCLCHECK(HcclCommDestroy(hcclComm));
    //重置设备
    ACLCHECK(aclrtResetDevice(devId));
    //设备去初始化
    ACLCHECK(aclFinalize());
    return 0;
}
```

8.5.3.4 HcclCommInitRootInfoConfig 初始化方式

该样例支持单机N卡的组网，N需要小于等于8。

```
#include <iostream>
#include <vector>
#include <memory>
#include <thread>
#include <chrono>
#include "hccl/hccl.h"
#include "hccl/hccl_types.h"
#include "mpi.h"

#define ACLCHECK(ret) do {\
    if(ret != ACL_SUCCESS)\
    {\
        printf("acl interface return err %s:%d, retcode: %d \n", __FILE__, __LINE__, ret);\
        return ret;\
    }\
```



```
} while(0)\

#define HCCLCHECK(ret) do {\
    if(ret != HCCL_SUCCESS)\
    {\
        printf("hccl interface return errreturn err %s:%d, retcode: %d \n", __FILE__, __LINE__, ret);\
        return ret;\
    }\
} while(0)

struct ThreadContext {
    HcclComm comm;
    int32_t device;
};
int Sample(void *arg)
{
    ThreadContext* ctx = (ThreadContext *)arg;
    void* host_buf = nullptr;
    void* send_buff = nullptr;
    void* recv_buff = nullptr;
    uint64_t count = 1;
    int malloc_kSize = count * sizeof(float);
    aclrtEvent start_event, end_event;
    aclrtStream stream;
    ACLCHECK(aclrtCreateStream(&stream));
    ACLCHECK(aclrtCreateEvent(&start_event));
    ACLCHECK(aclrtCreateEvent(&end_event));

    //申请集合通信操作的内存
    ACLCHECK(aclrtMalloc((void**)&send_buff, malloc_kSize, ACL_MEM_MALLOC_HUGE_FIRST));
    ACLCHECK(aclrtMalloc((void**)&recv_buff, malloc_kSize, ACL_MEM_MALLOC_HUGE_FIRST));

    //初始化输入内存
    ACLCHECK(aclrtMallocHost((void**)&host_buf, malloc_kSize));
    ACLCHECK(aclrtMemcpy((void*)send_buff, malloc_kSize, (void*)host_buf, malloc_kSize,
ACL_MEMCPY_HOST_TO_DEVICE));

    //执行集合通信操作
    HCCLCHECK(HcclAllReduce((void *)send_buff, (void*)recv_buff, count, HCCL_DATA_TYPE_FP32,
HCCL_REDUCE_SUM, ctx->comm, stream));

    //等待stream中集合通信任务执行完成
    ACLCHECK(aclrtSynchronizeStream(stream));

    if (ctx->device < 8) {
        void* resultBuff;
        ACLCHECK(aclrtMallocHost((void**)&resultBuff, malloc_kSize));
        ACLCHECK(aclrtMemcpy((void*)resultBuff, malloc_kSize, (void*)recv_buff, malloc_kSize,
ACL_MEMCPY_DEVICE_TO_HOST));
        float* tmpResBuff = static_cast<float*>(resultBuff);
        for (uint32_t i = 0; i < count; ++i) {
            std::cout <<  "rankId:" << ctx->device << ",i" << i << " " << tmpResBuff[i] << std::endl;
        }
        ACLCHECK(aclrtFreeHost(resultBuff));
    }
    ACLCHECK(aclrtFree(send_buff));
    ACLCHECK(aclrtFree(recv_buff));
    ACLCHECK(aclrtFreeHost(host_buf));
    //销毁任务流
    ACLCHECK(aclrtDestroyStream(stream));
    ACLCHECK(aclrtDestroyEvent(start_event));
    ACLCHECK(aclrtDestroyEvent(end_event));
    return 0;
}
int main(int argc, char*argv[])
{
    MPI_Init(&argc, &argv);
    int procSize = 0;
    int procRank = 0;
```



```
    // 获取当前进程在所属进程组的编号
    MPI_Comm_size(MPI_COMM_WORLD, &procSize);
    MPI_Comm_rank(MPI_COMM_WORLD, &procRank);
    int devId = procRank;
    int devCount = procSize;
    //设备资源初始化
    ACLCHECK(aclInit(NULL));
    // 指定集合通信操作使用的设备
    ACLCHECK(aclrtSetDevice(devId));
    // 在 rootRank 获取 rootInfo
    HcclRootInfo rootInfo;
    int32_t rootRank = 0;
    if(devId == rootRank) {
        HCCLCHECK(HcclGetRootInfo(&rootInfo));
    }
    // 将root_info广播到通信域内的其他rank
    MPI_Bcast(&rootInfo, HCCL_ROOT_INFO_BYTES, MPI_CHAR, rootRank, MPI_COMM_WORLD);
    MPI_Barrier(MPI_COMM_WORLD);
    // 创建并初始化通信域配置项
    HcclCommConfig config;
    HcclCommConfigInit(&config);

    // 根据需要修改通信域配置
    config.hcclBufferSize = 1024;
    config.hcclDeterministic = 1;
    strcpy(config.hcclCommName, "comm_1");

    // 初始化集合通信域
    HcclComm hcclComm;
    HCCLCHECK(HcclCommInitRootInfoConfig(devCount, &rootInfo, devId, &config, &hcclComm));

    // 创建任务stream
    struct ThreadContext args;
    args.comm = hcclComm;
    args.device = devId;
    Sample((void *)&args);
    //销毁集合通信域
    HCCLCHECK(HcclCommDestroy(hcclComm));
    //重置设备
    ACLCHECK(aclrtResetDevice(devId));
    //设备去初始化
    ACLCHECK(aclFinalize());
    return 0;
}
```

8.5.3.5 HcclCommInitAll 初始化方式

该样例仅支持单机8卡的组网，且仅支持单进程方式拉起。

```
#include <iostream>
#include <vector>
#include <memory>
#include <thread>
#include <chrono>
#include "hccl/hccl.h"
#include "hccl/hccl_types.h"

#define ACLCHECK(ret) do { \
    if(ret != ACL_SUCCESS)\
    {\
        printf("acl interface return err %s:%d, retcode: %d \n", __FILE__, __LINE__, ret);\
        return ret;\
    }\
} while(0)

#define HCCLCHECK(ret) do {  \
    if(ret != HCCL_SUCCESS) \
    {   \
        printf("hccl interface return errreturn err %s:%d, retcode: %d \n", __FILE__, __LINE__, ret); \
```

```
        return ret;\
    } \
} while(0)

struct ThreadContext {
    HcclComm comm;
    int32_t device;
};

int Sample(void *arg)
{
    ThreadContext* ctx = (ThreadContext *)arg;
    //std::cout << "----------------------" << "begin rankId:" << ctx->device << "----------------------------" <<
std::endl;
    // 申请通信用device、sendBuf，recvBuf内存、stream等资源
    ACLCHECK(aclrtSetDevice(ctx->device));
    aclrtStream stream;
    ACLCHECK(aclrtCreateStream(&stream));
    void* sendBuff;
    void* recvBuff;
    void* hostBuff;

    uint64_t count = 8;
    int mallocSize = count * sizeof(float);
    //初始化输入内存
    ACLCHECK(aclrtMallocHost((void**)&hostBuff, mallocSize));
    float* tmpHostBuff = static_cast<float*>(hostBuff);
    for (uint32_t i = 0; i < count; ++i) {
        tmpHostBuff[i] = 2;
    }
    ACLCHECK(aclrtMalloc((void**)&sendBuff, mallocSize, ACL_MEM_MALLOC_HUGE_FIRST));
    ACLCHECK(aclrtMemcpy((void*)sendBuff, mallocSize, (void*)hostBuff, mallocSize,
ACL_MEMCPY_HOST_TO_DEVICE));
    ACLCHECK(aclrtMalloc((void**)&recvBuff, mallocSize, ACL_MEM_MALLOC_HUGE_FIRST));

    // 执行AllReduce操作
    HCCLCHECK(HcclAllReduce((void *)sendBuff, (void*)recvBuff, count, HCCL_DATA_TYPE_FP32,
HCCL_REDUCE_SUM, ctx->comm, stream));
    ACLCHECK(aclrtSynchronizeStream(stream));

    if (ctx->device < 8) {
        void* resultBuff;
        ACLCHECK(aclrtMallocHost((void**)&resultBuff, mallocSize));
        ACLCHECK(aclrtMemcpy((void*)resultBuff, mallocSize, (void*)recvBuff, mallocSize,
ACL_MEMCPY_DEVICE_TO_HOST));
        float* tmpResBuff = static_cast<float*>(resultBuff);
        for (uint32_t i = 0; i < count; ++i) {
            std::cout <<  "rankId:" << ctx->device << ",i" << i << " " << tmpResBuff[i] << std::endl;
        }
        ACLCHECK(aclrtFreeHost(resultBuff));
    }
    // 释放通信用sendBuf、recvBuf内存，stream等资源
    ACLCHECK(aclrtFree(recvBuff));
    ACLCHECK(aclrtFree(sendBuff));
    ACLCHECK(aclrtFreeHost(hostBuff));
    ACLCHECK(aclrtDestroyStream(stream));
    ACLCHECK(aclrtResetDevice(ctx->device));
    //std::cout << "----------------------" << "end rankId:" << ctx->device << "----------------------------" <<
std::endl;
    return 0;
}

int main() {
    // 设备资源初始化
    ACLCHECK(aclInit(NULL));
    uint32_t ndev = 8;
    int32_t devices[8] = {0, 1, 2, 3, 4, 5, 6, 7};
    HcclComm comms[ndev];
    for (int32_t i = 0; i < ndev; i++) {
```



```
    ACLCHECK(aclrtSetDevice(devices[i]));
  }
  // 初始化通信域
  HCCLCHECK(HcclCommInitAll(ndev, devices, comms));
  // 启动线程执行集合通信操作
  std::vector<std::unique_ptr<std::thread> > threads(ndev);
  struct ThreadContext args[ndev];
  for (uint32_t i = 0; i < ndev; i++) {
    args[i].device = i;
    args[i].comm = comms[i];
    threads[i].reset(new (std::nothrow) std::thread(&Sample, (void *)&args[i]));
    std::chrono::seconds duration(6);
    std::this_thread::sleep_for(duration);
  }
  for (uint32_t i = 0; i < ndev; i++) {

    threads[i]->join();
  }
  // 释放通信域等相关资源
  for (uint32_t i = 0; i < ndev; i++) {
     HCCLCHECK(HcclCommDestroy(comms[i]));
  }
  std::cout << "end end end" << std::endl;
  // 设备资源去初始化
  ACLCHECK(aclFinalize());
  return 0;
}
```

8.5.3.6 HcclCreateSubCommConfig 初始化方式

该样例仅支持从单机8卡的组网中切分出2个4卡子通信域。

全局通信域可以是通过ranktable文件或者root info协商方式创建的，该样例以ranktable文件创建全局通信域为例。

准备 ranktable 文件

该样例通过获取ranktable的方式进行初始化，所以需准备一份ranktable文件配置集群信息，供后续调用接口时使用。

配置“RANK_TABLE_FILE”环境变量，指定ranktable文件所在路径，如下所示，文件名称为“ranktable.json”。

```
export RANK_TABLE_FILE=/home/test/ranktable.json
```

以**Atlas A2 训练系列产品**为例，ranktable.json配置示例如下，不同产品形态ranktable文件的配置示例及详细参数说明可参见**7.2 ranktable文件配置资源信息**。

```
{
    "status":"completed",  // ranktable可用标识，completed为可用
    "version": "1.0",
    "server_count": "1",  // 参与训练的AI Server数目
    "server_list": [{
        "server_id": "SERVER_ID_SV1",  // AI Server标识，String类型，请确保全局唯一
        "device": [{          // AI Server中的Device列表
            "device_id": "0",
            "device_ip": "192.168.1.8",
            "rank_id": "0"
        },
        {
            "device_id": "1",
            "device_ip": "192.168.1.9",
            "rank_id": "1"
        },
        {
            "device_id": "2",
            "device_ip": "192.168.1.10",
```



```
                "rank_id": "2"
            },
            {
                "device_id": "3",
                "device_ip": "192.168.1.10",
                "rank_id": "3"
            },
            {
                "device_id": "4",
                "device_ip": "192.168.1.10",
                "rank_id": "4"
            },
            {
                "device_id": "5",
                "device_ip": "192.168.1.10",
                "rank_id": "5"
            },
            {
                "device_id": "6",
                "device_ip": "192.168.1.10",
                "rank_id": "6"
            },
            {
                "device_id": "7",
                "device_ip": "192.168.1.11",
                "rank_id": "7"
            }]
        }]
}
```

代码示例

```
#include <iostream>
#include <vector>
#include <memory>
#include <thread>
#include <chrono>
#include <cstring>
#include "hccl/hccl.h"
#include "hccl/hccl_types.h"
#include "mpi.h"

#define ACLCHECK(ret) do { \
    if(ret != ACL_SUCCESS)\
    {\
        printf("acl interface return err %s:%d, retcode: %d \n", __FILE__, __LINE__, ret);\
        return ret;\
    }\
} while(0)

#define HCCLCHECK(ret) do {  \
    if(ret != HCCL_SUCCESS) \
    {   \
        printf("hccl interface return errreturn err %s:%d, retcode: %d \n", __FILE__, __LINE__, ret); \
        return ret;\
    } \
} while(0)

struct ThreadContext {
    HcclComm commFront;
    HcclComm commBack;
    int32_t device;
};
int Sample(void *arg)
{
    ThreadContext* ctx = (ThreadContext *)arg;
    void* host_buf = nullptr;
    void* send_buff = nullptr;
    void* recv_buff = nullptr;
```



```
    uint64_t count = 1;
    int malloc_kSize = count * sizeof(float);
    aclrtEvent start_event, end_event;
    aclrtStream stream;
    ACLCHECK(aclrtCreateStream(&stream));
    ACLCHECK(aclrtCreateEvent(&start_event));
    ACLCHECK(aclrtCreateEvent(&end_event));

    //申请集合通信操作的内存
    ACLCHECK(aclrtMalloc((void**)&send_buff, malloc_kSize, ACL_MEM_MALLOC_HUGE_FIRST));
    ACLCHECK(aclrtMalloc((void**)&recv_buff, malloc_kSize, ACL_MEM_MALLOC_HUGE_FIRST));

    //初始化输入内存
    ACLCHECK(aclrtMallocHost((void**)&host_buf, malloc_kSize));
    ACLCHECK(aclrtMemcpy((void*)send_buff, malloc_kSize, (void*)host_buf, malloc_kSize,
ACL_MEMCPY_HOST_TO_DEVICE));

    //执行集合通信操作
    if (ctx->device < 4) {
        HCCLCHECK(HcclAllReduce((void *)send_buff, (void*)recv_buff, count, HCCL_DATA_TYPE_FP32,
HCCL_REDUCE_SUM, ctx->commFront, stream));
    } else {
        HCCLCHECK(HcclAllReduce((void *)send_buff, (void*)recv_buff, count, HCCL_DATA_TYPE_FP32,
HCCL_REDUCE_SUM, ctx->commBack, stream));
    }

    //等待stream中集合通信任务执行完成
    ACLCHECK(aclrtSynchronizeStream(stream));

    if (ctx->device < 8) {
        void* resultBuff;
        ACLCHECK(aclrtMallocHost((void**)&resultBuff, malloc_kSize));
        ACLCHECK(aclrtMemcpy((void*)resultBuff, malloc_kSize, (void*)recv_buff, malloc_kSize,
ACL_MEMCPY_DEVICE_TO_HOST));
        float* tmpResBuff = static_cast<float*>(resultBuff);
        for (uint32_t i = 0; i < count; ++i) {
            std::cout <<  "rankId:" << ctx->device << ",i" << i << " " << tmpResBuff[i] << std::endl;
        }
        ACLCHECK(aclrtFreeHost(resultBuff));
    }
    ACLCHECK(aclrtFree(send_buff));
    ACLCHECK(aclrtFree(recv_buff));
    ACLCHECK(aclrtFreeHost(host_buf));
    //销毁任务流
    ACLCHECK(aclrtDestroyStream(stream));
    ACLCHECK(aclrtDestroyEvent(start_event));
    ACLCHECK(aclrtDestroyEvent(end_event));
    return 0;
}

int main()
{
    MPI_Init(NULL, NULL);
    int procSize = 0;
    int procRank = 0;
    // 获取当前进程在所属进程组的编号
    MPI_Comm_size(MPI_COMM_WORLD, &procSize);
    MPI_Comm_rank(MPI_COMM_WORLD, &procRank);
    int devId = procRank;
    int devCount = procSize;
    // 设备资源初始化
    ACLCHECK(aclInit(NULL));
    // 获取ranktable路径
    char* rankTableFile = getenv("RANK_TABLE_FILE");
    // 指定集合通信操作使用的设备
    ACLCHECK(aclrtSetDevice(devId));

    // 创建并初始化通信域配置项
    HcclCommConfig config;
```



```
    HcclCommConfigInit(&config);
    // 根据需要修改通信域配置
    config.hcclBufferSize = 50;
    strcpy(config.hcclCommName, "comm_1");

    HcclComm globalHcclComm;
    HcclCommInitClusterInfoConfig(rankTableFile, devId, &config, &globalHcclComm);
    struct ThreadContext args;
    if (devId < 4) {
        HcclComm hcclCommFront;
        strcpy(config.hcclCommName, "comm_2");
        uint32_t rankIdsFront[4] = {0, 1, 2, 3};
        HCCLCHECK(HcclCreateSubCommConfig(&globalHcclComm, 4, rankIdsFront, 1, devId, &config,
&hcclCommFront));
        args.commFront = hcclCommFront;
        args.device = devId;
        Sample((void *)&args);
        HCCLCHECK(HcclCommDestroy(hcclCommFront));
    } else {
        HcclComm hcclCommBack;
        strcpy(config.hcclCommName, "comm_3");
        uint32_t rankIdsBack[4] = {4, 5, 6, 7};
        HCCLCHECK(HcclCreateSubCommConfig(&globalHcclComm, 4, rankIdsBack, 2, devId - 4, &config,
&hcclCommBack));
        args.commBack = hcclCommBack;
        args.device = devId;
        Sample((void *)&args);
        HCCLCHECK(HcclCommDestroy(hcclCommBack));
    }
    // 设备资源去初始化
    ACLCHECK(aclFinalize());
    MPI_Finalize();
    return 0;
}
```

8.5.4 样例编译运行

介绍目录结构

以HcclCominitRootInfo初始化方式为例，样例代码目录结构如下所示：

```
CommInitRootInfo
├── bin
│   └── CommInitTest
├── Makefile
└── src
    └── main.cc
```

- Makefile文件内容可参见**配置Makefile文件**。
- main.cc的样例代码可参见**8.5.2 点对点通信样例（HcclSend/HcclRecv）**与**8.5.3 集合通信样例（HcclAllReduce）**。
- bin目录下的CommInitTest为编译后生成的可执行文件。

配置环境变量

配置样例编译时依赖的环境变量：

```
source /usr/local/Ascend/ascend-toolkit/set_env.sh
export INSTALL_DIR=/usr/local/Ascend/ascend-toolkit/latest
export PATH=/usr/local/mpich-3.2.1/bin:$PATH
export LD_LIBRARY_PATH=/usr/local/mpich-3.2.1/lib:${INSTALL_DIR}:$LD_LIBRARY_PATH
```

- “INSTALL_DIR”是CANN软件安装后文件存储路径，其中“/usr/local/Ascend”为root用户的默认安装路径，如果使用普通用户安装，或指定路径安装，请自行替换。

- “/usr/local/mpich-3.2.1/lib”为安装后MPI的库文件所在路径，请根据实际情况替换。

配置 Makefile 文件

Makefile文件示例如下：

```
#
#loading path
#-----------------------------------------------------------------------------------------------------------------------
----------------------
CXXFLAGS := -std=c++11\
        -Werror\
        -fstack-protector-strong\
        -fPIE -pie\
        -O2\
        -s\
        -Wl,-z,relro\
        -Wl,-z,now\
        -Wl,-z,noexecstack\
        -Wl,--copy-dt-needed-entries
Common_DIR = ./src
Common_SRC = $(wildcard ${Common_DIR}/*.cc)
HCCL_INC_DIR = ${ASCEND_DIR}/include
HCCL_LIB_DIR = ${ASCEND_DIR}/lib64
ACL_INC_DIR = ${ASCEND_DIR}/include
ACL_LIB_DIR = ${ASCEND_DIR}/lib64
MPI_INC_DIR = ${MPI_HOME}/include
MPI_LIB_DIR = ${MPI_HOME}/lib
LIST = CommInitTest
#
#library flags
#-----------------------------------------------------------------------------------------------------------------------
----------------------
LIBS = -L$(HCCL_LIB_DIR) -lhccl\
               -L$(ACL_LIB_DIR) -lascendcl\
               -L$(MPI_LIB_DIR) -lmpi
INCLUDEDIRS = -I$(HCCL_INC_DIR)\
                           -I$(ACL_INC_DIR)\
                           -I$(MPI_INC_DIR)\
                           -I$(Common_DIR)
#
#make
#-----------------------------------------------------------------------------------------------------------------------
----------------------
all:
        @mkdir -p bin
        g++ $(CXXFLAGS) $(Common_SRC) $(INCLUDEDIRS)  -o CommInitTest $(LIBS)
        @printf "\nCommInitTest compile completed\n"
        mv $(LIST) ./bin
.PHONY: clean
clean:
        rm -rf ./bin/*Test
```

编译样例

执行如下命令编译样例：

```
make MPI_HOME=/usr/local/mpich-3.2.1/ ASCEND_DIR=/usr/local/Ascend/ascend-toolkit/latest
```

编译成功后，会在执行编译命令的bin/目录下生成CommInitTest可执行文件。

执行样例

编译完成后，通过以下指令启动：


```
mpirun -n <npu_num> ./bin/CommInitTest
```

其中“-n”是需要启动的NPU总数。

须知

HcclCommInitAll初始化方式为单进程多线程的拉起方式，所以该初始化方式的样例执行命令为mpirun -1 ./bin/CommInitTest。

9 常用工具与配置

HCCL 业务配置

HCCL提供了系列环境变量用于进行集合通信业务侧配置，详细可参见集合通信相关配置。

HCCL 性能测试

分布式训练场景下，开发者可以通过HCCL Test工具测试集合通信的功能以及性能，此工具支持测试的集合通信算子包含all_gather、all_reduce、alltoallv、alltoall、broadcast、reduce_scatter、reduce、scatter，当前集群场景的集合通信常用的是all_reduce和alltoallv算子，因此测试集合通信性能当前也建议基于all_reduce和alltoallv算子。

HCCL Test工具的详细使用可参见《 HCCL性能测试工具使用指南 》。

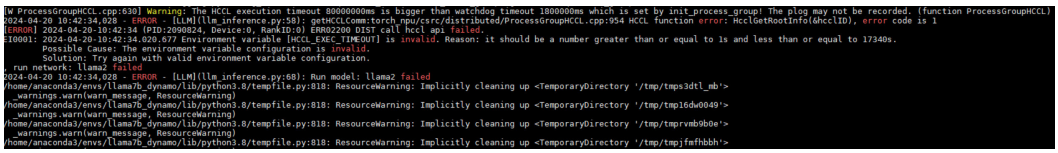
10 FAQ

10.1 HCCL 常见问题总结

10.1.1 环境变量配置错误（EI0001）

问题现象

执行日志报错：**EI0001 "Environment variable [***] is invalid"**，其中“***”是报错的环境变量名称，报错信息示例如下图所示。



原因分析

环境变量配置有问题，通常是参数超过可配置范围或在识别范围以外，报错信息关键字及含义如**表10-1**所示。

表 10-1 EI0001 报错信息关键字汇总

报错信息关键字	含义
RankIpFamily rank[ ] device ip family[ ] is not same with others[ ].	IPv4和IPv6混用。
HCCL_CONNECT_TIMEOUT it should be a number greater than or equal to 120s and less than or equal to 7200s	HCCL_CONNECT_TIMEOUT的取值不在允许范围内。

报错信息关键字	含义
HCCL_INTRA_PCIE_ENABLE or HCCL_INTRA_ROCE_ENABLE HCCL_INTRA_PCIE_ENABLE and HCCL_INTRA_ROCE_ENABLE cannot be both configured to 1	HCCL_INTRA_PCIE_ENABLE和HCCL_INTRA_ROCE_ENABLE两配置互斥，不能同时配置为1。
HCCL_WHITELIST_DISABLE It must be 0 or 1.	HCCL_WHITELIST_DISABLE的取值不在允许范围内。
HCCL_WHITELIST_FILE Please check env config	HCCL_WHITELIST_FILE配置有问题，通常为HCCL通信白名单配置文件内容异常，或者文件不存在。
HCCL_IF_IP it should be ip[ ]	HCCL_IF_IP配置的IP格式不正确。
HCCL_SOCKET_IFNAME Please check env config	HCCL_SOCKET_IFNAME配置格式不正确，请确认“，”位置。
HCCL_SOCKET_FAMILY it should be AF_INET or AF_INET6	HCCL_SOCKET_FAMILY配置参数不正确，需要是AF_INET或者AF_INET6。
HCCL_IF_BASE_PORT Value range[0,65520]	HCCL_IF_BASE_PORT的取值不在允许范围内。
HCCL_ALGO expect: levelX:algo1;levelY:algo2	HCCL_ALGO配置错误，通常为格式不符合要求、长度异常或内容不符合预期（重复配置、字段不正确）。
HCCL_RDMA_TC Value range[0, 255], Must be a multiple of 4	HCCL_RDMA_TC配置错误，通常为数值超范围、非数字、长度过长等。
HCCL_RDMA_SL Value range[0, 7]	HCCL_RDMA_SL的取值不在允许范围内。
HCCL_RDMA_TIMEOUT Value range[5, 24]	HCCL_RDMA_TIMEOUT的取值不在允许范围内。
HCCL_RDMA_RETRY_CNT Value range[1, 7]	HCCL_RDMA_RETRY_CNT的取值不在允许范围内。
HCCL_BUFFSIZE Value should be equal to or greater than 1(MB).	HCCL_BUFFSIZE的取值不在允许范围内。
HCCL_DETERMINISTIC Value should be true or false.	HCCL_DETERMINISTIC的取值不在允许范围内。
HCCL_ENTRY_LOG_ENABLE It must be 0 or 1.	HCCL_ENTRY_LOG_ENABLE的取值不在允许范围内。
HCCL_INTER_HCCS_DISABLE Value should be true or false.	HCCL_INTER_HCCS_DISABLE的取值不在允许范围内。
HCCL_OP_EXPANSION_MODE it should be "AI_CPU"	HCCL_OP_EXPANSION_MODE配置只能填AI_CPU

报错信息关键字	含义
HCCL_EXEC_TIMEOUT it should be a number greater than or equal to 0s and less than or equal to	HCCL_EXEC_TIMEOUT配置不在取值范围内
CM_CHIEF_IP it should be an available ip.	CM_CHIEF_IP配置的IP不可用。
CM_CHIEF_PORT it should be a unsigned number less than the max port num	CM_CHIEF_PORT的取值不在允许范围内。
CM_CHIEF_DEVICE it should be a unsigned number less than the max device num	CM_CHIEF_DEVICE的取值不在允许范围内。
CM_WORKER_IP it should be an available ip.	CM_WORKER_IP配置的IP不可用。
HCCL_WHITELIST_FILE HCCL_WHITELIST_DISABLE is [0] but HCCL_WHITELIST_FILE is not set	HCCL_WHITELIST_DISABLE配置为0，HCCL_WHITELIST_FILE却没有设置。
HCCL_WHITELIST_FILE hccl whitelist load config file[ ] failed.	HCCL_WHITELIST_FILE参数指定的文件打开失败，请确认路径是否正确。

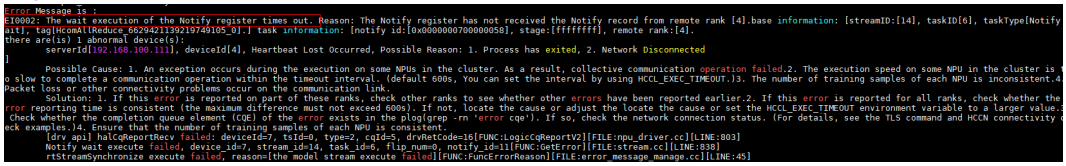
解决方法

确认报错提示的“环境变量”配置是否正确，并参见**表10-1**的报错信息进行修改。

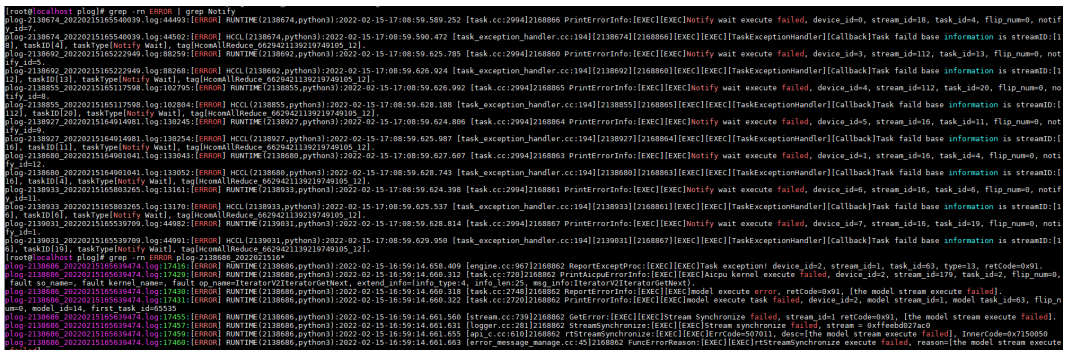
10.1.2 执行通信操作超时（EI0002）

问题现象

常见于算子执行阶段，屏显日志报错信息“EI0002: The wait execution of the Notify register times out.”



plog日志中查询Notify，有如下类似信息：



原因分析

HCCL算子的task会在指定集群的每个Device上执行，并通过notify进行状态同步，若任何一个rank或者通信链路在执行前/中发生异常，则会导致集群同步失败，剩余卡会出现notify wait timeout。常见的原因主要有：

1. 部分卡被某些耗时较长的任务阻塞，在超过设置的执行同步等待时间（可通过环境变量HCCL_EXEC_TIMEOUT配置，默认值为1800s）后才继续往下执行。
2. 部分rank未执行到notify同步阶段。
3. 网络模型等原因导致某些rank间的task执行序列不一致。
4. 节点间通信链路不稳定。
5. 训练场景下，用户设置的HCCL_EXEC_TIMEOUT小于HCCL_CONNECT_TIMEOUT时，建链失败，先报notify wait超时。

解决方法

收集所有卡的plog日志后，按以下步骤排查：

步骤1 检查所有卡的报错情况，若有卡未报notify超时错误，请通过训练或者plog日志检查此卡是否存在业务进程报错退出、卡死或core宕机的情况。

步骤2 若所有卡均上报notify超时错误，则检查错误日志中最早报错和最晚报错时间差是否超过算子执行超时阈值，若超过阈值请定位报错时间最晚的rank执行阻塞原因（如save checkpoint）或通过export HCCL_EXEC_TIMEOUT=3600（示例，请根据需要配置）调整超时阈值。

步骤3 检查集群中是否存在Device网口通信链路不稳定的情况，搜索所有卡的Device侧日志，若存在error cqe的打印，时间位于业务区间内，这种情况可能是发生网络拥塞，需要排查交换机配置是否合理、训练过程中是否有网口link down等情况。

搜索命令样例：grep –rn “err cqe” | grep HCCL

```
19987:[ERROR] HCCL(85111,python3):2023-02-18-09:44:55.431.692 [heartbeat.cc:547][85111][94635][Heartbeat]cqe err status[12], time:[2023-02-18 09:44:55.369944], ip:[192.0.1.7]
```

----结束

10.1.3 集合通信操作参数无效（EI0003）

问题现象

执行日志报错：EI0003 "value [***] for parameter [***] is invalid"，如下所示。

```
custom_group :None
test_type :all
dtype :float32
data :1024
iter :1
timeout： None
profiling :false
staged :false
pid :1036690
[2024-04-24 06:30:56.081654: F ge_plugin.cc:338] [GePlugin] Initialize ge failed, ret : failed
Error Messageis:
EI0003: 2024-04-24-06:30:52.388.386 In [HcomInitByFile], value [0] for parameter [rankTablePath] is invalid. Reason: The collective communication operator has an invalid argument. Reason[0]
        Solution: Try again with a valid argument.
        TraceBack (most recent call last):
        PluginManager InvokeAll failed.[FUNC:Initialize] [FILE:ops_kernel_manager.cc][LINE :89]
        OpsManager initialize failed.[FUNC:InnerInitialize] [FILE:gelib.cc] [LINE:241]
```



```
    GELib::InnerInitialize failed.[FUNC :Initialize][FILE:gelib.cc][LINE:169]
    GEInitialize failed. [FUNC:GEInitialize] [FILE:ge_api. cc][LINE :307]
```

原因分析

该报错常见于输入数据校验阶段，通常可能有以下几种原因：

1. 接口入参为指针，但传入指针为空。
2. 接口传入的参数超过预期枚举值范围。
3. 接口传入的字符串无效，或长度不符合预期。
4. 接口传入的路径无效，或者路径文件不符合预期。

具体报错字段及报错原因可以参考日志提示信息“value[ ]”和 “parameter[ ]”进行确认。

解决方法

针对日志提示字段“value[ ]”和 “parameter[ ]”进行排查，排查原因见上述原因分析。

10.1.4 无效的 RankTable 配置（EI00004）

问题现象

执行日志报错：EI0004 “The ranktable or rank is invalid”，如下所示。

```
custom_group :None
dtype :float32
data :1
iter :1
profiling :false
pid :1040540
[2024-04-24 06:31:38.087571: F ge_plugin. cc:338] [GePlugin] Initialize ge failed, ret :failed
Error Message is:
EI0004: 2024-04-24-06:31:33.915.634 The ranktable or rank is invalid, Reason:[The ip in ranktable is not a valid ip address]. Please check the configured ranktable. [The ranktable path configured in the training can be found in the plogs.]
    Solution: Try again with a valid cluster configuration in the ranktable file. Ensure that the configuration matches the operating environment.
    TraceBack (most recent call last):
    PluginManager InvokeAll failed. [FUNC:Initialize][FILE:ops_kernel_manager. cc][LINE :89]
    OpsManager initialize failed. [FUNC:InnerInitialize][FILE:gelib. cc] [LINE :241]
    GELib::InnerInitialize failed. [FUNC:Initialize] [FILE:gelib.cc][LINE:169]
    GEInitialize failed.[FUNC:GEInitialize][FILE:ge_api. cc][LINE :307]
```

原因分析

该报错常见于rank id或ranktable的数据校验阶段，通常有以下几种原因：

1. rank id的值不符合预期（过大的非法值、非数值、超过ranktable中的rank个数），或者同Server内已有rank id重复。
2. ranktable的格式或参数错误：版本不对，路径不对，格式不对，起始rank不为0，Server内的rank个数不正确，server_count、server_list、super_pod_id、server_id、server_index、ip等参数为空或配置不正确。

解决方法

针对执行日志提示字段中的“Reason”，可以基本看出问题方向，基于此排查配置下发参数的rank id或者ranktable，即可确认问题点。

10.1.5 卡间集合通信参数不一致（EI0005）

问题现象

执行日志报错：EI0005“The arguments for collective communication are inconsistent between ranks”，如下所示。

```
custom_group :None
test_type : sum
dtype :float32
data :1024
fusion :0
fusion_id :1
fusion_num :2
iter :1
profiling :false
para_err_type :1
pid :1052803
ranksize is 8, rankid is 4.
time start: 2024-04-24 06:32:20.702705
ERROR : GeOp3_OGEOP::::DoRunAsync Failed
Error Message is:
EI0005: 2024-04-24-06:32:27.781.599 The arguments for collective communication are inconsistent between ranks:tag[HcomAllReduce_6629421139219749105_0]，parameter[count]，local[16512]，remote [8320]
        Solution: Check whether the training script and ranktable of each NPU are consistent.
        TraceBack (most recent call last):
        Transport init error. Reason: [Create] [DestLink]Create Dest error! creakLink para:rank[5]-localUserrank[4]-localIpAddr[192.1.1. 243], dst_rank[6]-remoteUserrank[7]-remote_ip_addr[192.1.1.243]
        Transport init error. Reason: [Create] [DestLink]Create Dest error! creakLink para:rank[5]-localUserrank[4]-localIpAddr[192.1.1. 243], dst_rank[4]-remoteUserrank[5]-remote_ip_addr[192.1.1.243]
        call hccl op:HcomAllReduce(HcomAllReduce) load task fail[FUNC:Distribute][FILE:hccl_task_info.cc][LINE:329]
        [[{[node Ge0p3_0]}]]
train fail
pid is 1052803
```

原因分析

该报错是本端和对端的校验参数不一致导致，问题一般出现在建链阶段。建链阶段本端接收到对端发过来的校验帧，然后和本端数据进行对比校验，确认数据是否一致。

通常原因是上层框架或用户调用传参有问题（取决于HCCL上层为框架还是用户直调），需要结合报错提示的“tag”确定出错算子，“parameter”确定出错算子参数。

解决方法

- 若是用户直调出错集合通信算子，请根据错误信息中的"parameter"字段，确认算子接口入参的正确性。
- 若是HCCL上层框架调用场景，您可以获取日志后单击**Link**联系技术支持。

10.1.6 建链接超时（EI0006）

问题现象

常见于算子加载阶段，有以下2种情况：

- Server间的建链超时现象，出现“LINK_ERROR_INFO”报错信息，如下所示：

```
[ERROR] HCCL(1473737,python3):2022-09-29-15:31:05.319.815 [comm_base.cc:934][1473737][1482272][LOAD][LOAD]  ____________________LINK_ERROR_INFO____________________
[ERROR] HCCL(1473737,python3):2022-09-29-15:31:05.319.823 [comm_base.cc:935][1473737][1482272][LOAD][LOAD]  | comm error, device[6] num[1]                                  |
[ERROR] HCCL(1473737,python3):2022-09-29-15:31:05.319.827 [comm_base.cc:936][1473737][1482272][LOAD][LOAD]  | dest_ip[rank_id] | src_ip[rank_id]  |  Role  |  Status  |
[ERROR] HCCL(1473737,python3):2022-09-29-15:31:05.319.831 [comm_base.cc:937][1473737][1482272][LOAD][LOAD]  |------------------|------------------|--------|----------|
[ERROR] HCCL(1473737,python3):2022-09-29-15:31:05.319.848 [comm_base.cc:992][1473737][1482272][LOAD][LOAD]  | 192.2.1.20(14)   |  192.2.1.16(6)   | server | no connect |
[ERROR] HCCL(1473737,python3):2022-09-29-15:31:05.319.853 [comm_base.cc:950][1473737][1482272][LOAD][LOAD]
[ERROR] HCCL(1473737,python3):2022-09-29-15:31:05.319.856 [comm_base.cc:951][1473737][1482272][LOAD][LOAD]the connection failure between this device and target device may be due to the following reasons:
[ERROR] HCCL(1473737,python3):2022-09-29-15:31:05.319.861 [comm_base.cc:952][1473737][1482272][LOAD][LOAD]1. the connection between this device and the target device is abnormal.
```

- Server内的建链超时现象，日志报错信息如下所示：

```
[ERROR] HCCL(1869739,python3):2022-09-27-15:57:37.497.793 [exchanger_network.cc:695][1869739][1880385][LOAD][LOAD]  |device[0] userrank[0] exchanger Status: run_step[1]|
[ERROR] HCCL(1869739,python3):2022-09-27-15:57:37.497.796 [exchanger_network.cc:698][1869739][1880385][LOAD][LOAD]  |  dest_dev  |  userrank  |    Role    | connStatus |
[ERROR] HCCL(1869739,python3):2022-09-27-15:57:37.497.801 [exchanger_network.cc:697][1869739][1880385][LOAD][LOAD]  |------------|------------|------------|------------|
[ERROR] HCCL(1869739,python3):2022-09-27-15:57:37.497.806 [exchanger_network.cc:716][1869739][1880385][LOAD][LOAD]  |         0  |         0  |     NA     |     NA     |
[ERROR] HCCL(1869739,python3):2022-09-27-15:57:37.497.810 [exchanger_network.cc:716][1869739][1880385][LOAD][LOAD]  |         1  |         1  |   server   |     NO     |
[ERROR] HCCL(1869739,python3):2022-09-27-15:57:37.497.814 [exchanger_network.cc:716][1869739][1880385][LOAD][LOAD]  |         2  |         2  |   client   |     NO     |
[ERROR] HCCL(1869739,python3):2022-09-27-15:57:37.497.817 [exchanger_network.cc:716][1869739][1880385][LOAD][LOAD]  |         3  |         3  |   server   |     NO     |
[ERROR] HCCL(1869739,python3):2022-09-27-15:57:37.497.821 [exchanger_network.cc:716][1869739][1880385][LOAD][LOAD]  |         4  |         4  |   client   |     NO     |
[ERROR] HCCL(1869739,python3):2022-09-27-15:57:37.497.824 [exchanger_network.cc:716][1869739][1880385][LOAD][LOAD]  |         5  |         5  |   server   |     NO     |
[ERROR] HCCL(1869739,python3):2022-09-27-15:57:37.497.827 [exchanger_network.cc:716][1869739][1880385][LOAD][LOAD]  |         6  |         6  |   client   |     NO     |
[ERROR] HCCL(1869739,python3):2022-09-27-15:57:37.497.830 [exchanger_network.cc:716][1869739][1880385][LOAD][LOAD]  |         7  |         7  |   server   |     NO     |
[ERROR] HCCL(1869739,python3):2022-09-27-15:57:37.497.834 [exchanger_network.cc:721][1869739][1880385][LOAD][LOAD]  |------------|------------|------------|------------|
[ERROR] HCCL(1869739,python3):2022-09-27-15:57:37.497.838 [exchanger_network.cc:722][1869739][1880385][LOAD][LOAD]the connection failure between this device and the target device may be due to
following reasons:
[ERROR] HCCL(1869739,python3):2022-09-27-15:57:37.497.841 [exchanger_network.cc:723][1869739][1880385][LOAD][LOAD]1. the connection between this device and the target device is abnormal.
```

原因分析

HCCL会在指定集群的每个Device上运行，并在集群间建立socket链接，若任一个rank或者通信链路在建链前/中发生异常，则会导致集群建链失败。常见的原因包括：

1. 部分rank未执行到正确的建链阶段。
2. 部分卡被某些耗时较长的任务阻塞，在超过socket建链超时等待时间（可通过HCCL_CONNECT_TIMEOUT配置，默认值为120秒）后才执行到对应阶段。
3. 脚本等原因导致某些rank间的通信算子数量或者排序不一致。
4. 节点间通信链路不通或者不稳定。

解决方法

收集所有卡的plog日志后，按以下步骤排查：

步骤1 检查所有卡的报错情况，若有卡未报建链超时错误，请通过训练或者plog日志检查此卡是否存在业务卡死情况，若存在请定位根因。

步骤2 若所有卡均上报建链超时错误，则检查错误日志中最早和最晚时间差异是否超过超时阈值，若超过阈值请定位最晚报错的卡执行较慢的原因或通过增大建链超时时间尝试解决（例如通过配置export HCCL_CONNECT_TIMEOUT=*3600*，请根据需要调整超时阈值）。

步骤3 若时间差异未超过阈值，请检查各rank执行的集合通信算子数量是否一致，算子加载的顺序是否一致。HCCL要求一个通信域内各个rank上算子加载顺序保持一致。

如需查询tag和计算图中node name的对应关系，需开启INFO日志，在Host日志中搜索：

关键字1：GenerateOpTag:graph

关键字2：GenerateTask:graph

关键字1可以看到node的hash值，关键字2可以看到nodHash对应的NodeName，协助计算图排查。

步骤4 检查集群中是否存在Device网口通信链路不通的情况，比较常见的原因：

1. IP不在同一网段或子网掩码配置存在问题。
2. IP冲突，集群中存在IP相同的两个rank。
3. 交换机配置不在同一个vlan。
4. 链路不通。
5. 各rank的TLS(安全增强)设置不一致时也会导致建链失败。

   可通过hccn tool命令来确认TLS是否一致，如不一致请参考随产品发布的《 HCCN Tool接口参考 》进行TLS配置。

   《 HCCN Tool接口参考 》的获取方式为：单击**Link**进入企业业务网站的“**昇腾计算 文档中心**”，然后在“**中心训练硬件**”栏选择对应的硬件型号，单击进入对应的文档页面，即可在其中找到配套版本的《 HCCN Tool接口参考 》。

   查询TLS状态命令：

   ```
   hccn_tool -i 0 -tls -g
   hccn_tool -i 1 -tls -g
   hccn_tool -i 2 -tls -g
   hccn_tool -i 3 -tls -g
   hccn_tool -i 4 -tls -g
   hccn_tool -i 5 -tls -g
   hccn_tool -i 6 -tls -g
   hccn_tool -i 7 -tls -g
   ```

   TLS switch值为0表示关闭，1表示开启。如果提示no certificate found，也表示TLS功能关闭。如果各个rank的TLS情况不一致，可使用hccn_tool工具配置TLS，或者关闭所有rank的TLS功能。

   关闭TLS功能命令：

   ```
   hccn_tool -i 0 -tls -s enable 0
   hccn_tool -i 1 -tls -s enable 0
   hccn_tool -i 2 -tls -s enable 0
   hccn_tool -i 3 -tls -s enable 0
   hccn_tool -i 4 -tls -s enable 0
   hccn_tool -i 5 -tls -s enable 0
   hccn_tool -i 6 -tls -s enable 0
   hccn_tool -i 7 -tls -s enable 0
   ```

----结束

10.1.7 CANN 版本不一致，HCCL 返回错误码 EI0008

问题现象

常见于通信建链阶段，屏显日志报错：EI0008: The CANN versions are inconsistent。

```
ERROR : GeOp5_0GEOP::::DoRunAsync Failed
Error Message is :
EI0008: The CANN versions are inconsistent: tag [HcomAllReduce_6629421139219749105_0], local_version [1.83.T10.0.B206], remote_version [1.83.T10.0.B207]
        Solution: Install the same CANN version.
        TraceBack (most recent call last):
        Call ops_kernel_info_store LoadTask fail[FUNC:Distribute][FILE:hccl_task_info.cc][LINE:213]
```

原因分析

集合通信建链时会校验本端与对端Rank的CANN版本一致性，如果CANN版本不一致，HCCL会返回错误并打印错误码EI0008。

解决方法

确认不同服务器上安装的CANN软件版本是否一致，若不一致则需要安装一致的版本。

步骤1 在plog日志（INFO级别）中grep关键字CannVersion，确认各节点的CANN软件版本。

步骤2 重新安装一致的CANN软件版本。

----结束

10.2 HCCP 常见问题总结

10.2.1 EJ0001 打屏报错

问题现象

plog日志中，TDT首报错为：[DeviceMsgProcess][tid:1241254] [TsdClient] DeviceMsgProc errcode[EJ0001]

```
[ERROR] TDT(685010,all_reduce_test):2023-11-29-11:55:41.334.702 [process_mode_manager.cpp:587] [DeviceMsgProcess][tid:685010] [TsdClient] DeviceMsgProc  errcode[EJ0001]
[ERROR] TDT(685010,all_reduce_test):2023-11-29-11:55:41.334.873 [process_mode_manager.cpp:269] [WaitRsp][tid:685010] tsd client wait response fail, device response code[1]. unknown device error.
[ERROR] TDT(685010,all_reduce_test):2023-11-29-11:55:41.334.893 [process_mode_manager.cpp:123] [OpenProcess][tid:685010] Wait open response from device failed.
[ERROR] TDT(685010,all_reduce_test):2023-11-29-11:55:41.334.897 [tsd_client.cpp:31][TsdOpen] [tid:685010] TsdOpen failed, deviceId[4].
```

EJ0001错误只能说明拉起device HCCP进程失败，具体失败原因需要根据device报错进一步区分，在debug目录下，使用/usr/local/Ascend/driver/tools/msnpureport -f导出device日志，grep -rn ERROR * | grep HCCP查看HCCP首报错，Device报错有以下两种场景。

- Device报错“Create Server failed, ret(61)”

  EJ0001报错，通常看到HCCP报错：[ra_adp.c:14c_server_init(1425)：Create Server failed, ret(61)

  ```
  [ERROR] HCCP(13722,hccp_service.bin):2023-11-29-11:55:41.806.183 [ra_adp.c:1425]tid:13722,ra_hdc_server_init(1425) : Create Server failed, ret(61)
  [ERROR] HCCP(13722,hccp_service.bin):2023-11-29-11:55:41.806.200 [ra_adp.c:1546]tid:13722,hccp_init(1546) : chip_id[0] ra_hdc_server_init failed, ret[-22]
  [ERROR] HCCP(13722,hccp_service.bin):2023-11-29-11:55:41.806.265 [main.c:224]tid:13722,main(224) : hccp init error[-22]
  ```

- Device报错“certificate is not yet valid”

原因分析与解决方法（Device 报错“Create Server failed, ret(61)”）

若Device报错“Create Server failed, ret(61)”，可能有以下两种原因：

- 人为操作问题（偶现），上一次训练未完全退出，就拉起新的训练。

  原因分析：

  在Host侧执行如下hccn_tool命令：

  ```
  for i in {0..7}; do hccn_tool -i $i -process -g ; done
  ```

  查看Device侧是否存在hccp进程。

  解决办法：

  稍等一会（通常停掉训练任务脚本，会进资源销毁释放流程），确认进程完全退出后再拉起训练进程。

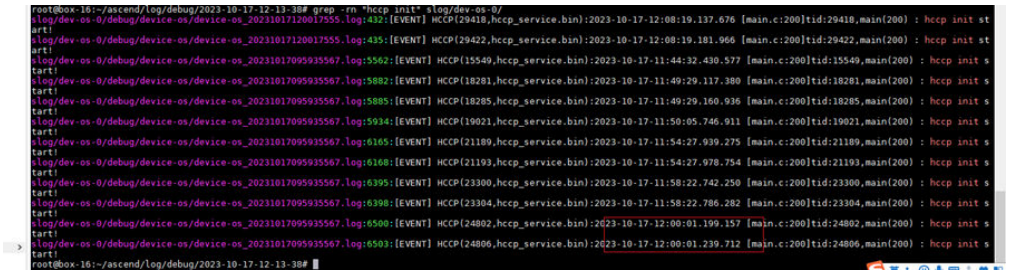
- 业务脚本问题（必现），每次拉起训练时，会在一个device上拉起2个及以上的hccp进程。

  原因分析：

  执行如下命令，导出日志，并确认是否在临近间隔时间，通常是ms级时间间隔内，有两次及以上hccp进程拉起的日志记录。

  ```
  cd /root/ascend/log/debug/
  /usr/local/Ascend/driver/tools/msnpureport -f
  grep -rn "hccp init" *
  ```

  如下所示，即可判定是由于业务脚本在一个Device上拉起2个及以上的HCCP进程导致的，需要用户排查业务脚本。

  需要注意：该日志等级为EVENT，请确保EVENT日志开关已打开。

  日志等级设置参考：

  ```
  export ASCEND_GLOBAL_LOG_LEVEL=0    # 开启DEBUG
  export ASCEND_GLOBAL_LOG_LEVEL=1    # 开启INFO
  export ASCEND_GLOBAL_LOG_LEVEL=2    # 开启WARNING
  export ASCEND_GLOBAL_LOG_LEVEL=3    # 开启ERROR，默认为ERROR级别
  ```

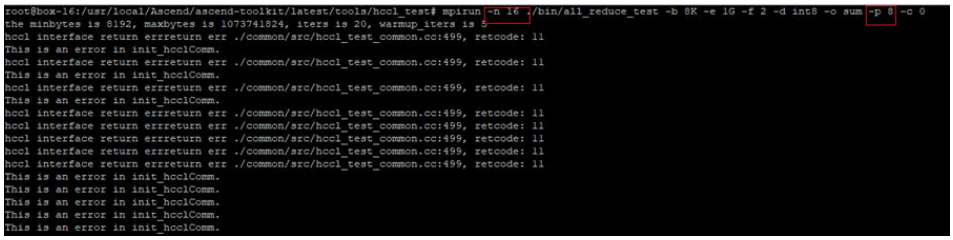
  EVENT日志打开命令：

  ```
  export ASCEND_GLOBAL_EVENT_ENABLE=1   # 1表示开，0表示关
  ```

  解决方法：

  需要自行排查业务训练脚本中问题，如下是遇到的几个案例

  a. 单机16卡跑hccl test报错：-p与-n不配套导致多次拉起。

  b. pytorch nproc_per_node这个参数应该填写这个节点上要启动的训练进程数，只有2卡可用，填写的8，所以重复拉起4次，报错了--**https://pytorch.org/docs/stable/distributed.html**。对应日志中看到重复拉起了4次：

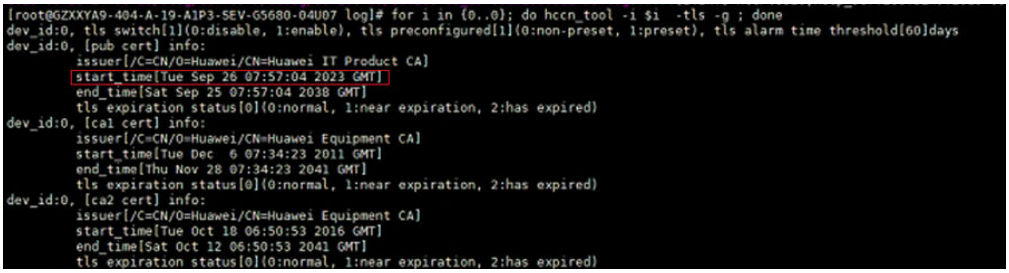
原因分析及解决方法（Device 报错“certificate is not yet valid”）

- **原因分析：**

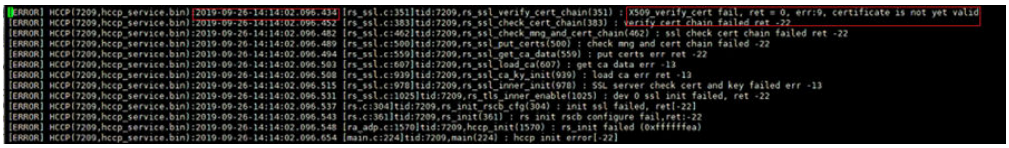
  若Device报错“certificate is not yet valid”，则原因为：

  Host时钟异常，导致系统时间早于TLS证书有效时间，TLS开关打开情况下校验证书失败，拉起训练失败。

如下所示，查询TLS证书看到证书有效起始时间是：2023/09/26



对应HCCP进程拉起时，报错证书无效，校验失败，拉起HCCP失败，看到系统时间是：2019/09/26



- **解决方法：**

  同步正常时间后，可以恢复。

10.2.2 EJ0002 打屏报错

问题现象

拉起训练进程时，报EJ0002 Environment Error，此报错通常由于环境异常，导致rdev初始化失败，拉起训练进程失败。

查看HCCP初始化**ra_rdev**阶段报错，对应HDC接口报错信息为：**ra hdc message process failed ret(-67)**

```
[ERROR] HCCP(46430,alltoallv_test):2023-09-21-03:43:49.546.469 [ra
hdc.c:1270]tid:46430,ra_hdc_rdev_init(1270) : [init][ra_hdc_rdev]ra hdc message process failed ret(-67)
phy_id(3)
[ERROR] HCCP(46430,alltoallv_test):2023-09-21-03:43:49.546.488
[ra_host.c:621]tid:46430,ra_rdev_init(621) : [init][ra_rdev]ra rdev init failed. ret(-67)
```

原因分析

网卡down导致初始化时，HCCP调用HDC接口返回-67，对应错误码定义（ #define ENOLINK 67 /* Link has been severed */ ）

解决方法

1. 执行如下命令，检查网口状态。

   ```
   for i in {0..7}; do hccn_tool -i $i -link -g ; done
   ```

2. 用户自行排查物理链路是否连通，检查软件配置是否正确。
   - 重新配置ip和netmask（有可能未配置ip）

     ```
     hccn_tool -a -cfg recovery
     ```

     基于/etc/hccn.conf中配置恢复环境配置
   - 查询光模块是否在位（evb环境无光模块）

     ```
     for i in {0..7}; do hccn_tool -i $i -optical -g; done
     ```
   - 查询交换机信息（交换机信息会有缺失）

     ```
     for i in {0..7}; do hccn_tool -i $i -lldp -g; done
     ```


- 咨询环境人员光纤类型，是否交换机打开了对应的FEC策略、CDR版本是否过老，需要升级、是否光模块问题，将有问题的光模块+光纤和无问题的交换验证

10.2.3 EJ0003 打屏报错

问题现象

建链阶段报EJ0003的错误，HCCP调系统函数bind失败，常见报错**error: 98**或**error: 99**。

该报错对应Linux标准报错定义如下：

```
# define EADDRINUSE      98      /* Address already in use */
# define EADDRNOTAVAIL   99      /* Cannot assign requested address */
```

查看plog日志中关于HCCP的报错：

grep -rn ERROR * | grep HCCP | head

报错1：

```
plog/plog-3541138_20231113190511833.log:30551:[ERROR]
HCCP(3541138,host_cpu_executor):2023-11-13-19:05:49.833.681
[rs_socket.c:671]tid:3541138,rs_socket_listen_bind_listen(671) : bind fail! family:2, IP:127.0.0.1, port:50000,
sock:27, ret:0xffffffff, error:98, Possible Cause: the IP address and port have been bound already
plog/plog-3541138_20231113190511833.log:30552:[ERROR]
HCCP(3541138,host_cpu_executor):2023-11-13-19:05:49.833.687
[rs_socket.c:860]tid:3541138,rs_socket_listen_start(860) : bind and listen fail, err_no:98, listen_fd:27, listen
state:1, IP(127.0.0.1) server_port:50000
plog/plog-3541138_20231113190511833.log:30554:[ERROR]
HCCP(3541138,host_cpu_executor):2023-11-13-19:05:49.833.704
[ra_peer.c:114]tid:3541138,ra_peer_socket_listen_start(114) : [listen_start][ra_peer_socket]ra listen start
failed ret(-98).
```

报错2：

```
[ERROR] HCCP(160639,all_gather_test):2023-10-17-07:28:13.178.137
[rs_socket.c:671]tid:160639,rs_socket_listen_bind_listen(671) : bind fail! ::350d:755b:add4:ada6, port:60000,
sock:25, ret:0xffffffff, error:99, Possible Cause: the IP address and port have been bound already
```

报错3：

```
75346:[ERROR] HCCP(13854,python3):2023-09-04-14:37:34.444.964
[rs_socket.c:1124]tid:21717,rs_connect_bind_client(1124) : client bind fail! IP:fe80::4b79:e6c9:31eb:f019,
sock:530, ret:-1, error:99
75347:[ERROR] HCCP(13854,python3):2023-09-04-14:37:34.444.968
[rs_socket.c:1164]tid:21717,rs_socket_state_reset(1164) : rs_connect_bind_client failed, ret[-99]
```

原因分析

- 配置错误。

  例如未配置网卡IP，或者网卡IP与业务配置IP不匹配，或者对一个IP地址及端口重复监听了多次。
- Host侧未指定网卡建链，导致进程绑定IP地址报错。
- 网卡状态异常。

解决方法

步骤1 检查配置。

- 查看Device网卡的IP地址是否存在。

  执行**hccn_tool -i dev_id -ip -g**命令查询对应的Device网卡的IP地址是否存在。

  若不存在，请执行如下命令进行配置。

  hccn_tool -i dev_id -ip -s address 192.168.0.3 netmask 255.255.255.0
- 检查ranktable表中配置的IP地址与网卡实际IP是否一致，若不一致，请修改。

  说明：可以在Device日志中搜索关键字“listen fail”查看监听失败的IP地址
- 检查应用程序端口与HCCL系统进程端口是否冲突。

  分布式训练场景下，HCCL默认会使用Host服务器的60000-60015端口收集集群信息，若通过环境变量HCCL_IF_BASE_PORT指定了Host网卡起始端口，则需要预留以该端口起始的16个端口。

  可通过**lsof -i:60000**确认占用端口的进程信息，通过**netstat -ant | grep WAIT**命令查看端口状态。

  若确认是应用程序端口与HCCL系统进程端口冲突，可通过如下命令预留系统端口。

  ```
  sysctl -w net.ipv4.ip_local_reserved_ports=60000-60015
  ```
- 检查是否存在IP与端口重复监听的问题。

  设置日志级别为EVENT，然后在日志中**grep -rn "socket bind" ***，确认当前训练端口监听情况。

步骤2 若绑定网卡报错，可通过环境变量HCCL_SOCKET_IFNAME指定初始化Host通信网卡，HCCL可通过该网卡名获取Host IP，完成通信域创建。

配置示例如下：

```
export HCCL_SOCKET_IFNAME==eth0
```

步骤3 执行**hccn_tool -i dev_id -link -g**命令查看对应的Device网卡状态，若网卡状态异常，请排查网卡错误。

----结束

10.2.4 EJ0004 打屏报错

问题现象

训练拉起失败，报**EJ0004**错误，查看日志，HCCP首报错如下：

```
[ERROR] HCCP(1609,python3):2023-10-19-12:33:13.313.594
[ra_host.c:544]tid:1609,ra_rdev_init_check_ip(544) : [check][ip]fail, ret(-16) the IP address(29.4.84.36) in the ranktable is inconsistent with the IP(29.4.76.85)address of the network adapter, please make sure they're consistent. num(1)
```

搜索Error日志：**grep -rn ERROR | grep HCCP | grep inconsistent**

发现当前节点ranktable中的ip和device ip不一致，如下所示。

```
plog/plog-1601_20231019123251864.txt:21:[ERROR] HCCP(1601,python3):2023-10-19-12:33:11.645.514
[ra_host.c:544]tid:1601,ra_rdev_init_check_ip(544) : [check][ip]fail, ret(192) the IP address(29.4.47.87) in the ranktable is inconsistent with the IP(29.4.76.155)address of the network adapter, please make sure they're consistent. num(1)
plog/plog-1590_20231019123251766.txt:21:[ERROR] HCCP(1590,python3):2023-10-19-12:33:11.313.520
[ra_host.c:544]tid:1590,ra_rdev_init_check_ip(544) : [check][ip]fail, ret(160) the IP address(29.4.159.125) in the ranktable is inconsistent with the IP(29.4.67.170)address of the network adapter, please make sure they're consistent. num(1)
plog/plog-1612_20231019123251802.txt:21:[ERROR] HCCP(1612,python3):2023-10-19-12:33:11.365.521
[ra_host.c:544]tid:1612,ra_rdev_init_check_ip(544) : [check][ip]fail, ret(161) the IP address(29.4.156.176) in the ranktable is inconsistent with the IP(29.4.69.45)address of the network adapter, please make sure they're consistent. num(1)
```



```
plog/plog-1585_20231019123251689.txt:21:[ERROR] HCCP(1585,python3):2023-10-19-12:33:11.377.587
[ra_host.c:544]tid:1585,ra_rdev_init_check_ip(544) : [check][ip]fail, ret(-162) the IP address(29.4.59.254) in
the ranktable is inconsistent with the IP(29.4.0.99)address of the network adapter, please make sure they're
consistent. num(1)
plog/plog-1593_20231019123251798.txt:21:[ERROR] HCCP(1593,python3):2023-10-19-12:33:11.785.581
[ra_host.c:544]tid:1593,ra_rdev_init_check_ip(544) : [check][ip]fail, ret(-80) the IP address(29.4.187.47) in
the ranktable is inconsistent with the IP(29.4.132.121)address of the network adapter, please make sure
they're consistent. num(1)
plog/plog-1609_20231019123251900.txt:21:[ERROR] HCCP(1609,python3):2023-10-19-12:33:13.313.594
[ra_host.c:544]tid:1609,ra_rdev_init_check_ip(544) : [check][ip]fail, ret(-16) the IP address(29.4.84.36) in the
ranktable is inconsistent with the IP(29.4.76.85)address of the network adapter, please make sure they're
consistent. num(1)
plog/plog-1598_20231019123251821.txt:21:[ERROR] HCCP(1598,python3):2023-10-19-12:33:11.735.192
[ra_host.c:544]tid:1598,ra_rdev_init_check_ip(544) : [check][ip]fail, ret(64) the IP address(29.4.146.67) in
the ranktable is inconsistent with the IP(29.4.184.7)address of the network adapter, please make sure
they're consistent. num(1)
plog/plog-1606_20231019123251824.txt:21:[ERROR] HCCP(1606,python3):2023-10-19-12:33:11.525.543
[ra_host.c:544]tid:1606,ra_rdev_init_check_ip(544) : [check][ip]fail, ret(96) the IP address(29.4.21.78) in the
ranktable is inconsistent with the IP(29.4.85.223)address of the network adapter, please make sure they're
consistent. num(1)
```

原因分析

此错误通常由于业务配置错误导致。

解决方法

请检查ranktable配置，确认是否存在以下问题：

- 存在重复的rank id。
  若存在重复rank id，请修改，确保rank id全局唯一。
- 配置的Device IP地址与实际Device的网卡地址不一致。
  可通过如下命令查询Device IP：
  ```
  for i in `seq 0 7`; do echo "==================> dev$i, NPU$((i+1))"; hccn_tool -i $i -ip -g; done
  ```
  若不一致，请修改配置的Device IP地址为实际的网卡地址。

10.2.5 参与集合通信的服务器 TLS 信息不一致，HCCL 初始化失败

问题现象

分布式训练场景下，集合通信建链失败，HCCL关键日志信息如下：

```
[EVENT] HCCP(23457,all_reduce_test):2023-10-16-08:39:56.291.195
[ra_host.c:1672]tid:23468,ra_socket_white_list_add(1672) : Input parameters: phy_id[0],
local_ip[192.168.100.101], num[1]
[ERROR] HCCL(23457,all_reduce_test):2023-10-16-08:41:56.291.748 [comm_base.cc:967][23468]
_____________________LINK_ERROR_INFO_____________________
[ERROR] HCCL(23457,all_reduce_test):2023-10-16-08:41:56.291.771 [comm_base.cc:968][23468]  | comm
error, device[0] num[1]
[ERROR] HCCL(23457,all_reduce_test):2023-10-16-08:41:56.291.777 [comm_base.cc:969][23468]  |
dest_ip(rank_id) | src_ip(rank_id) | Role | Status |
[ERROR] HCCL(23457,all_reduce_test):2023-10-16-08:41:56.291.783 [comm_base.cc:970][23468]
|--------------------|--------------------|----------|------------|
[ERROR] HCCL(23457,all_reduce_test):2023-10-16-08:41:56.291.797 [comm_base.cc:1018][23468]  |
192.168.100.100(1)  |  192.168.100.101(0)  | server | no connect |
[ERROR] HCCL(23457,all_reduce_test):2023-10-16-08:41:56.291.804 [comm_base.cc:983][23468]
_____________________________________________________
[ERROR] HCCL(23457,all_reduce_test):2023-10-16-08:41:56.291.809 [comm_base.cc:984][23468]the
connection failure between this device and target device may be due to the following reasons:
[ERROR] HCCL(23457,all_reduce_test):2023-10-16-08:41:56.291.815 [comm_base.cc:985][23468]1. the
connection between this device and the target device is abnormal.
[ERROR] HCCL(23457,all_reduce_test):2023-10-16-08:41:56.291.820 [comm_base.cc:986][23468]2. an
```



```
exception occurred at the target devices.
[ERROR] HCCL(23457,all_reduce_test):2023-10-16-08:41:56.291.825 [comm_base.cc:988][23468]3. the time difference between the execution of hcom on this device and the target device exceeds the timeout threshold. make sure this by keyworld [Entry-]
[ERROR] HCCL(23457,all_reduce_test):2023-10-16-08:41:56.291.830 [comm_base.cc:990][23468]4. the behavior of executing the calculation graph on this device and the target device is inconsistent.
[ERROR] HCCL(23457,all_reduce_test):2023-10-16-08:41:56.291.836 [comm_base.cc:991][23468]5. The TLS switch is inconsistent, or the TLS certificate expires.
[ERROR] HCCL(23457,all_reduce_test):2023-10-16-08:41:56.291.841 [comm_base.cc:992][23468]6. If src and dst IP address can be pinged, check whether the device IP address conflicts.
[ERROR] HCCL(23457,all_reduce_test):2023-10-16-08:41:56.291.847 [comm_base.cc:437][23468][Get][RaSocket]in comm, get rasocket error role[0], rank[0], num[1], goten[0], timeout[120]
[ERROR] HCCL(23457,all_reduce_test):2023-10-16-08:41:56.291.853 [comm_base.cc:768][23468]call trace: hcclRet -> 9
[ERROR] HCCL(23457,all_reduce_test):2023-10-16-08:41:56.291.861 [comm_base.cc:827][23468]call trace: hcclRet -> 9
[ERROR] HCCL(23457,all_reduce_test):2023-10-16-08:41:56.291.867 [comm_base.cc:199][23468]call trace: hcclRet -> 9
[ERROR] HCCL(23457,all_reduce_test):2023-10-16-08:41:56.291.873 [comm_base.cc:82][23468]call trace: hcclRet -> 9
```

查询HCCP Debug日志信息，关键报错信息如下：

```
[ERROR] HCCP(6594,hccp_service.bin):2023-10-16-08:39:56.335.828 [rs_ssl.c:206]tid:6610,rs_ssl_err_string(206) ； ssl fd 42 err 1 errno 0, err code 167773208, err msg error:0A000418:SSL routines::tlsv1 alert unknown ca, The possible cause is that the TLS switch is inconsistent.
```

原因分析

参与集合通信的各服务器TLS状态开关不一致，或者当TLS状态开关统一打开时，TLS证书信息不一致。

解决方法

步骤1 查询集合通信的各服务器TLS状态开关。

在服务器中执行如下命令，获取TLS开关使能状态。

```
hccn_tool -i <device_id> -tls -g [host]
```

其中<device_id>为Device设备的逻辑ID，您也可以通过如下for语句，一次性查询所有Device设备的TLS信息。

```
for i in `seq 0 7`; do hccn_tool -i $i -tls -g; done    # 0，7分别为需要查询的Device ID的起始与结束值。
```

回显信息如下所示：

```
dev_id:0, tls switch[0](0:disable, 1:enable), tls alarm time threshold[60]days
dev_id:0, [pub cert] info:
        issuer[/C=CN/ST=GD/O=HUAWEI/OU=2012/CN=2_1thCA]
        start_time[Wed Feb 19 03:19:21 2020 GMT]
        end_time[Sat Feb 16 03:19:21 2030 GMT]
dev_id:0, [ca1 cert] info:
        issuer[/C=CN/ST=GD/L=SZ/O=HUAWEI/CN=1thCA]
        start_time[Wed Feb 19 03:19:07 2020 GMT]
        end_time[Sat Feb 16 03:19:07 2030 GMT]
dev_id:0, [ca2 cert] info:
        issuer[/C=CN/ST=GD/L=SZ/O=HUAWEI/CN=1thCA]
        start_time[Wed Feb 19 03:19:10 2020 GMT]
        end_time[Sat Feb 16 03:19:10 2030 GMT]
dev_id:1, tls switch[0](0:disable, 1:enable), tls alarm time threshold[60]days
dev_id:1, [pub cert] info:
        issuer[/C=CN/ST=GD/O=HUAWEI/OU=2012/CN=2_1thCA]
        start_time[Wed Feb 19 03:19:21 2020 GMT]
        end_time[Sat Feb 16 03:19:21 2030 GMT]
dev_id:1, [ca1 cert] info:
```



```
        issuer[/C=CN/ST=GD/L=SZ/O=HUAWEI/CN=1thCA]
        start_time[Wed Feb 19 03:19:07 2020 GMT]
        end_time[Sat Feb 16 03:19:07 2030 GMT]
dev_id:1, [ca2 cert] info:
        issuer[/C=CN/ST=GD/L=SZ/O=HUAWEI/CN=1thCA]
        start_time[Wed Feb 19 03:19:10 2020 GMT]
        end_time[Sat Feb 16 03:19:10 2030 GMT]
... ...
```

其中tls switch[0]，代表TLS状态为关闭，switch[1]代表TLS状态为使能。

步骤2 判断各服务器中所有Device的TLS状态开关是否一致。

- 若不一致，建议统一修改TLS状态为使能。若TLS开关关闭，集合通信时会存在信息被窃听、篡改、仿冒的风险。

  您可以通过如下命令修改TLS状态开关：

  ```
  hccn_tool -i <device_id> -tls -s enable 1
  ```

  enable为使能开关，配置为1代表使能，配置为0代表关闭。
- 若一致且状态为使能，建议您继续执行**步骤3**判断各节点的TLS证书信息是否一致。

步骤3 查看所有服务器中各Device的TLS证书信息是否一致。

您可以通过**步骤1**中的信息判断各Device TLS证书信息是否一致。若不一致，您可以通过如下命令替换证书套件。

```
hccn_tool -i 0 -tls -s path /root pri pri.pem pub pub.pem ca1 ca1.pem ca2 ca2.pem crl xxx.crl
```

-i为Device ID，-path为指定证书/私钥/吊销列表存放路径；pri为私钥名字；pub为设备证书文件名；ca1/ca2/crl分别为根证书、二级根证书、吊销列表文件名。

关于hccn_tool工具的更多用法及参数解释，可查看对应设备的《 HCCN Tool 接口参考 》。

《 HCCN Tool接口参考 》的获取方式为：单击**Link**进入企业业务网站的“**昇腾计算 文档中心**”，然后在“**中心训练硬件**”栏选择对应的硬件型号，单击进入对应的文档页面，即可在其中找到配套版本的《 HCCN Tool接口参考 》。

----结束

10.3 HCCL Test 常见问题总结

10.3.1 单机带宽低

问题现象

单机Hccl test带宽峰值稳定低于预期值（ 预期值可以参见**Ascend Training Solution** 中对应产品的《 集群交付一本通 》中“单机测试 > 测试方案 > 测试规格” ）。

原因分析

1. 日志等级设置不是error。

   若某些卡的日志等级设置不是error，可能会导致对应卡的带宽降低（ 某些场景下，可能会导致带宽降低2GB左右 ）。
2. 某条链路单宽低。

某条链路带宽低会导致单机整机单宽低。

解决步骤

1. 若是日志级别设置不是error导致的带宽降低，可通过重新设置Host侧与Device侧日志级别为error解决。

   Host侧日志级别：

   ```
   export ASCEND_GLOBAL_LOG_LEVEL=3    // (0:debug 1:info 2:warning 3:error)
   ```

   Device侧日志操作：

   ```
   # 查询
   for i in {0..7}; do /usr/local/Ascend/driver/tools/msnpureport -r -d $i; done
   # 设置
   for i in {0..7}; do msnpureport -g error -d $i; done
   ```

2. 若重设日志级别后，带宽仍然较低，可分析是否某条链路HCCS带宽低，若是单机16卡机型，是否PCIe链路带宽低。

   可通过ascend-dmi命令测p2p的带宽

   测选卡测某条链路：

   ascend-dmi --bw -t p2p --ds 0 --dd 8

   测全量p2p：

   ascend-dmi --bw -t p2p #测试时间较长预计20分钟

   然后在**昇腾社区**提issue解决。

10.3.2 多机峰值带宽低

问题现象

多机峰值带宽低，多机进行带宽测试时，测试结果峰值带宽稳定低于预期值

原因分析

- 使用HCCL Test性能测试工具时，未设置HCCL_BUFFSIZE环境变量，或者测试命令中设置的测试数据大小较小。
- 开启了Profiling或日志等级未设置为默认的ERROR级别，会导致带宽较低。
- 某台机器带宽较低，导致多机带宽低
- 交换机负载分担方式配置不合理，产生拥塞
- spine和leaf交换机之间的某个端口down

解决步骤

步骤1 确认设置**HCCL_BUFFSIZE环境变量**是否设置。

环境变量HCCL_BUFFSIZE用于调整两个NPU之间共享数据的缓存区大小，两个NPU之间共享数据的缓存区大小默认为200M，使用HCCL Test工具进行性能测试的场景下，往往通信数据量较大，此种场景下，可适当增大HCCL_BUFFSIZE的值，提升数据通信效率与带宽。

配置示例：

```
export HCCL_BUFFSIZE=2048
```


步骤2 确认hccl_test测试命令中的参数“-e”是否配置过小。

“-e”代表测试数据大小的结束值，若“-e”较小，则带宽会较小，建议-e后面的数据量写到4G，例如：

mpirun -n 8 ./bin/all_reduce_test -b 8K -e 4G -f 2 -d fp32 -o sum -p 8

步骤3 查看是否开启了Profiling性能数据采集功能。

Porfiling性能数据采集功能会导致带宽变低，请关闭此功能。

每种业务场景的性能数据采集功能开启方法不同，详细的方法可参见《性能分析工具使用指南》中的“性能分析”章节。

步骤4 查看日志级别是否为“ERROR”。

查看Host与Device的日志级别：

- Host侧日志级别可通过**echo $ASCEND_GLOBAL_LOG_LEVEL**命令查看，0代表debug，1代表info，2代表warning，3代表error。
- Device侧日志级别可通过如下命令查看：

  for i in {0..7}; do /usr/local/Ascend/driver/tools/msnpureport -r -d $i; done

若日志级别不为默认值ERROR，则执行如下命令分别修改HOST与DEVICE的日志级别为“ERROR”。

- Host侧日志级别修改

  在测试主机设置即可，MPI会将环境变量带到所有测试机器：

  export ASCEND_GLOBAL_LOG_LEVEL=3 // (0:debug 1:info 2:warning 3:error)
- Device日志级别修改

  for i in {0..7}; do msnpureport -g error -d $i; done

步骤5 多机带宽较低还可能是某台机器带宽较低或网络配置不一致导致的。

此种场景下，可以通过二分法找到这台机器，然后参见**10.3.1 单机带宽低**进行可能原因的排查，若单机测试均无问题可尝试检查慢机器的配置“cat /etc/hccn.conf”与其他正常服务器配置是否一样，某台机器网络配置不一致，有可能导致单机测试无问题（单机不使用外部网络），但引起多机带宽低。

步骤6 检查交换机负载分担方式是否配置合理。

执行如下命令，查看服务器统计信息：

for i in $(seq 0 15); do echo "==============> $i"; hccn_tool -i $i -stat -g | grep pfc ;done

统计信息中有很多“rx pfc”，标识交换机负载分担不均衡，产生了拥塞。

可通过如下方法尝试解决：

- 首先解决交换机的pfc问题，要做到pfc很少甚至没有pfc。
- 查看交换机流量是否不均衡，有没有出现多打一（多条流量走一个端口出），有些端口流量大有些端口流量小，产生拥塞的情况。
- 假如是通过udp端口进行哈希选路，查看是否某些机器的udp port没配置。

----结束

10.3.3 小数据量时延大

问题现象

小数据量时延大，如小于4M数据量时单机8卡allreduce时延大于300us

原因分析

Device的日志级别如果不是默认值ERROR，会导致时延较大。

可通过如下命令查询Device日志等级：

```
for i in {0..7}; do /usr/local/Ascend/driver/tools/msnpureport -r -d $i; done
```

解决步骤

执行如下命令，将Device日志等级设置为error。

```
for i in {0..7}; do msnpureport -g error -d $i; done
```

10.3.4 多机测试大数据量带宽骤降为 0

问题现象

多机大数据量进行性能测试时出现以下情况：

- 带宽骤降为0点几。
- 查看统计数据，重传次数（roce_new_pkt_rty_num）会增长。
- 通过环境变量“HCCL_RDMA_TIMEOUT”缩短重传等待时间后，带宽不会再骤降为0点几，例如：
  ```
  export HCCL_RDMA_TIMEOUT=10
  ```

原因分析

数据转发过程中发生丢包，产生了重传，从而拖慢时间。

解决步骤

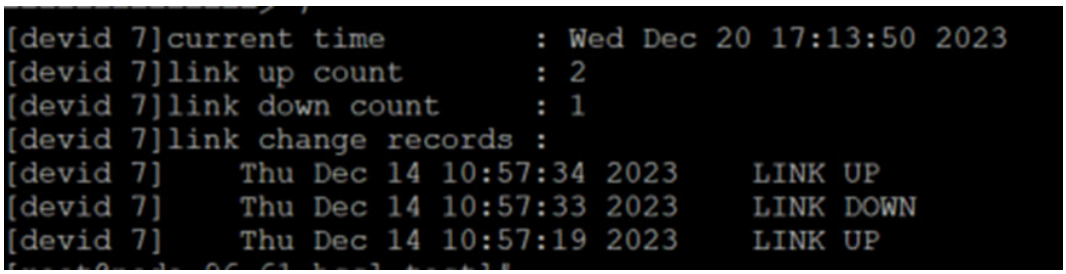
解决此问题的根因是找到丢包所在的位置，例如spine交换机 、leaf交换机、 服务器、线缆等，可按照如下顺序排查丢包的地方。

步骤1 丢包在线缆，端口闪断丢包。

查询端口link记录

```
for i in $(seq 0 7); do echo "==============> $i"; hccn_tool -i $i -link_stat -g ;done
```

找到闪断的端口，然后排查硬件闪断原因。



步骤2 丢包在交换机 ——交换机配置原因。

需要排查交换机配置，分别登录spine和leaf交换机检查丢包原因，可能存在如下原因：

- 交换机buffer水线设置不合理，流控方式不对。
- 只配了spine的PFC，没有配leaf交换机PFC，spine反压给leaf ，leaf没有降速，导致丢包。
- ECN和PFC水线配置配合不合理导致丢包，若不是一定要用ECN，建议关掉ECN。

步骤3 丢包在交换机——交换机和服务器配合原因。

交换机和服务器流控参数配置不匹配，流量进错队列丢包。

交换机和服务器的dscp、pfc优先级队列等需要协商一致，原理请参见《**Ascend Training Solution 组网指南**》中的“**拥塞控制与纠错配置策略**”章节。

排查交换机配置：

可以在交换机侧看对应的报文dscp对不对，有没有走到对应的队列，命令：

display qos queue statistics interface 10ge 1/0/1

排查服务器流控参配置：

```
查询 dscp_to_tc
for i in $(seq 0 7); do echo "===============> $i"; hccn_tool -i $i -dscp_to_tc -g ;done
查询 PFC
for i in $(seq 0 7); do echo "===============> $i"; hccn_tool -i $i -pfc -g;done
设置dscp_to_tc
for i in $(seq 0 7); do echo "===============> $i"; hccn_tool -i $i -dscp_to_tc -s dscp 33 tc 2;done
设置PFC
for i in $(seq 0 7); do echo "===============> $i";hccn_tool -i $i -pfc -s bitmap 0,0,0,0,1,0,0,0;done
```

排查流控环境变量：

测试之前要设确保这两个环境变量已经设置，不配置采用默认参数（TC=132 SL=4）

```
export HCCL_RDMA_TC=132 #数值为dscp左移两位（乘以4），根据和交换机协商配置
export HCCL_RDMA_SL=4 #数值为PFC的队列，根据和交换机协商配置
```

步骤4 丢包在服务器（TC buffer被打爆导致丢包）。

原因可能是服务器buffer设置不合理，或者是交换机未响应服务器的PFC，最大使用的buffer达到了TCbuffer最大值，把buffer打爆导致服务器丢包。

说明：默认buffer水线经过多个场景最佳实践一般不需要调整。

- 排查服务器统计是否有tx pfc
  ```
  for i in $(seq 0 15); do echo "===============> $i"; hccn_tool -i $i -stat -g ;done | grep pfc
  ```
- 排查buffer和上下水线，判断使用的最大buffer是否达到了设置的buffer上限
  ```
  #查询tc buffer配置
  for i in $(seq 0 15); do hccn_tool -i $i -tc_cfg -g;done
  #查询某个TC使用的最大buffer（读清）
  for i in $(seq 0 15); do hccn_tool -i $i -cur_tc_buf -g tc 1;done #tc根据使用的tc改
  ```

----结束

10.3.5 测试命令卡数与实际卡数不一致，返回 retcode 11

问题现象

执行HCCL Test测试命令时，返回“return code 11”错误，例如：

“hccl interface return errreturn err ./common/src/hccl_test_common.cc:499, retcode: 11”

原因分析

HCCL Test测试命令中配置的卡数与实际的卡数不一致。如下错误命令所示：

```
mpirun -n 16 ./bin/all_reduce_test -b 8K -e 1G -f 2 -d int8 -o sum -p 8 -c 0
```

-n：需要启动的NPU总数。

-p：单个计算节点上参与训练的NPU个数。

示例命令错误的原因为：这是一个单机测试命令，“ -n 16”说明要启动NPU总数为16个，“-p 8 ”单个节点参与训练NPU为8，导致这个节点的卡数不够总共要启动的卡数。

解决步骤

修改测试命令，检查“-n”是否与“-p”的NPU个数（进程数）是否书写正确。

- -n：需要启动的NPU总数
- -p：单个计算节点上参与训练的NPU个数

10.3.6 未设置 hostname 与 ip 对应关系导致 MPI error 报错

问题现象

报错MPI error，报错类似如下：

```
Fatal error in MPI_Init_thread: Other MPI error, error stack:
MPIR_Init_thread(474)
MPID_Init(190)  channel initialization failed
MPIDI_CH3_Init(89)
MPID_nem_init(320).
MPID_nem_tcp_init(173)
MPID_nem_tcp_get_business_card(42θ):
MPID_nem_tcp_init(379).
.: gethostbyname failed, Georgreens-MacBook-Pro.local (errno 1)
```

原因分析

“/etc/hosts” 文件中未配置对应的主机IP地址和主机名。

解决步骤

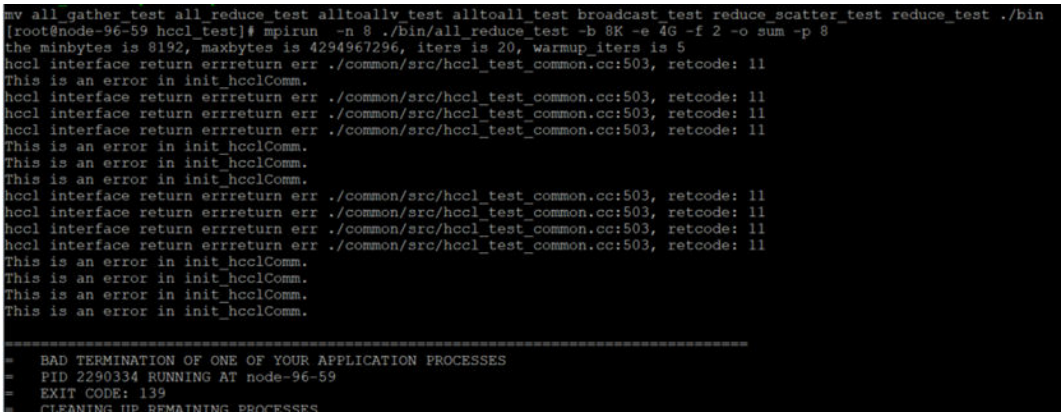
/etc/hosts添加主机IP地址（Host IP）和主机名（Host Name），如下所示：

```
172.16.0.100 HW-AI-LC-1-1
```


10.3.7 HCCLComm 初始化失败

问题现象

执行HCCL Test测试工具时，报“This is an error in init_hcclComm”的错误，如下图所示：



原因分析

某些卡被进程占用，导致无法使用HCCL Test工具进行测试。

解决步骤

步骤1 执行“npu-smi info”命令查看卡的占用情况。

某些场景下，npu-smi info未显示卡被占用，但片上内存使用非常高，此种情况下，也会引起HCCL Test测试工具执行失败。

步骤2 确认被占用卡上的进程释放后，再重新执行HCCL Test测试工具。

通常停掉训练任务脚本，会进入资源释放销毁释放流程，可在Host侧执行如下命令，确认进程完全退出后，再执行测试工具。

```
for i in {0..7}; do hccn_tool -i $i -process -g ; done
```

----结束

10.3.8 防火墙未关闭

问题现象

测试报错如下，查询机器防火墙发现防火墙未关闭

```
[root@node-87-66 hccl_test]# mpirun -f hostfile -n 16 ./bin/all_reduce_test -b 8K -e 4G -f 2 -d fp32 -p 8
the minbytes is 8192, maxbytes is 4294967296, iters is 20, warmup_iters is 5
Fatal error in PMPI_Barrier: Unknown error class, error stack:
PMPI_Barrier(425)..............: MPI_Barrier(MPI_COMM_WORLD) failed
MPIR_Barrier_impl(332).........: Failure during collective
MPIR_Barrier_impl(327).........:
MPIR_Barrier(292)..............:
MPIR_Barrier_intra(150)........:
barrier_smp_intra(96)..........:
MPIR_Barrier_impl(332).........: Failure during collective
MPIR_Barrier_impl(327).........:
MPIR_Barrier(292)..............:
```



```
MPIR_Barrier_intra(169).....:
MPIDI_CH3U_Recvq_FDU_or_AEP(629): Communication error with rank 8
barrier_smp_intra(111).....:
MPIR_Bcast_impl(1452).....:
MPIR_Bcast(1476).....:
MPIR_Bcast_intra(1287).....:
MPIR_Bcast_binomial(310).....: Failure during collective
```

原因分析

发生此问题的原因一般是防火墙未关闭。

不同系统防火墙查询命令略有不同，例如：

systemctl status firewalld

解决步骤

通过systemctl命令关闭防火墙，不同系统设置命令略有不同，命令示例如下：

- 关闭防火墙：
  systemctl stop firewalld
- 禁用防火墙（开机不启动）：
  systemctl disable firewalld
  sudo ufw status

10.3.9 SSH 连接超时错误

问题现象

多机场景下，执行HCCL Test测试工具时，报如下错误：

```
ssh：connect to host xx.xx.xx.xx port 22: Connection time out
```

原因分析

此错误原因是测试主机无法远程登录到其他所有机器导致。

多机测试场景，需要配置操作节点到集群通信所有节点的SSH信任关系，以支持集群通信节点远程登录。

解决步骤

1. 在当前操作节点生成密钥信息（如若环境中存在，可不重复执行）：
   ```
   ssh-keygen -t rsa
   ```
   例如密钥信息生成后，存储在/root/.ssh/id_rsa.pub文件中。
2. 将操作节点公钥复制到集群通信其他节点，实现SSH密钥登录远程主机。
   ```
   ssh-copy-id -i /root/.ssh/id_rsa.pub node1_address
   ssh-copy-id -i /root/.ssh/id_rsa.pub node2_address
   ```
3. 在操作节点上SSH远程登录其他节点，确认是否可以直接登录。

10.3.10 多台机器 MPI 版本不一致问题

问题现象

多机场景下，HCCL Test测试工具执行过程中发生概率性失败，coredump等随机问题。

原因分析

可能是多台机器的MPI软件版本不一致导致，可通过如下命令查询MPI软件版本：

```
which mpirun find / -name mpirun
```

解决步骤

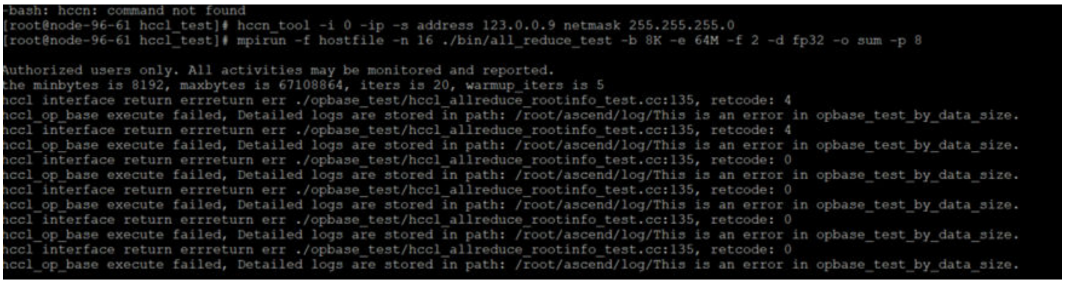
当前版本，如果通信网卡使用IPv4协议，建议安装**MPI 3.2.1**版本；如果通信网卡使用IPv6协议，建议安装**Open MPI-4.1.5**版本。

且所有机器的MPI软件版本需要保持一致。

10.3.11 Device 网络不通报错 retcode 4

问题现象

多机场景下，HCCL Test工具执行时，报错“retcode: 4”，如下图所示：



原因分析

Device网络不通，导致建链失败。

解决步骤

在Host侧执行如下命令，分别ping每张卡，确认是否网络连通。

```
hccn_tool -i 0 -ping -g address 192.169.150.60
```

同平面卡需要两两互通，即所有机器的同号卡间要互相ping通（如：两两之间0卡ping0卡，1卡ping1卡，以此类推），同时如果为单机16卡机型，所有机器的0卡和8卡，1卡和9卡，2卡和11卡，以此类推都需要两两互通

命令如下：

hccn_tool -i 0 -ping -g address 192.169.150.60 #当前机器的0卡ping另外一台机器的一张卡的device ip。

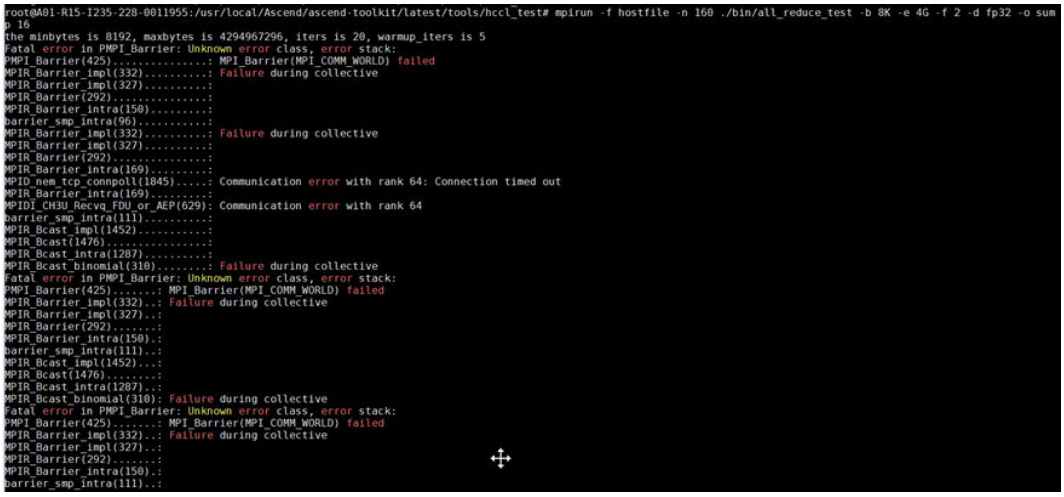
若不通，联系相关网络同事排查网络连通性。

说明：有可能改完ip，没有对应改网关gateway，也会导致device之间不通，注意ip netmask gateway要对应配置。

10.3.12 host ip 之间两两不通报 MPI 错误

问题现象

多机场景下，执行HCCL Test测试工具时，报错误：“Fatal error in PMPI_Barrier: Unknown error class, error stack”，如下图所示。



原因分析

HCCL Test测试场景，要求所有机器的Host网卡两两ping通，此问题一般是由于某台机器的Host IP与其他机器网络不通导致。

解决步骤

步骤1 定位到网络不通的Host机器。

可通过二分的方法，逐步添加机器进行测试，当添加到某台机器报错时，可登录此机器，然后ping其他机器的Host IP，若不通，即确认是此机器网络原因。

步骤2 解决此机器与其他机器的Host IP无法ping通的问题。

----结束

10.3.13 端口 down 报错 ret(-67)

问题现象

HCCL Test工具执行报错，在Host日志目录“~/ascend/log”下执行命令**grep -rn “ERROR”**，有关键报错：ra hdc message process failed ret(-67)

原因分析

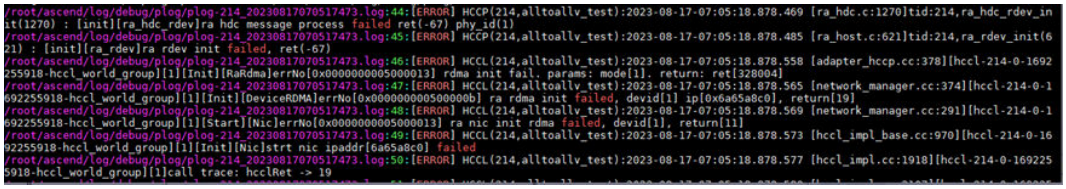
可能是某网口状态为down导致其他正常卡与其通信时，建链超时。

解决步骤

在Host侧执行如下命令，查询网口状态：

```
for i in {0..7}; do hccn_tool -i $i -link -g ; done
```

若某网卡link状态为down，初始化网卡时会失败，如下图所示，从而导致其他正常卡与其通信时，上报建链超时的错误。



定位到问题网口后，即可联系硬件工程师排查网口问题；若紧急恢复可尝试重启或重新插拔光模块。

10.3.14 Device IP 冲突导致 get socket 超时

问题现象

网络正常连通，且TLS状态一致的场景下，日志存在get socket超时的错误，使用hccn_tool进行卡之间ping操作，正向ping与反向ping存在丢包情况。

原因分析

可能是由于两台Device IP地址冲突导致。

解决步骤

重新配置Device IP，避免IP地址冲突。

10.3.15 未指定网卡名报错 retcode 9

问题现象

多机测试场景下，执行HCCL Test测试工具时，进程执行卡住，然后报错retcode 9，如下图所示：

```
# mpirun -f hostfile -n 32 ./bin/all_reduce_test -b 1 -e 2G -p 16 -f 2 -o sum -c 0
the minbytes is 1, maxbytes is 2147483648, iters is 20,warmup_iters is 5
hccl interface return errreturn err ./common/src/hccl_test_common.cc:499, retcode:9
hccl interface return errreturn err ./common/src/hccl_test_common.cc:499, retcode:9
hccl interface return errreturn err ./common/src/hccl_test_common.cc:499, retcode:9
hccl interface return errreturn err ./common/src/hccl_test_common.cc:499, retcode:9
...
This is an error in init_hcclComm.
```

原因分析

未配置初始化root通信网卡名导致。

解决步骤

多机场景下，执行HCCL Test测试工具前需要在操作节点配置HCCL初始化root通信网卡的网卡名，HCCL可通过该网卡名获取Host IP，完成通信域创建。

配置方式如下：

```
配置HCCL的初始化root通信网卡使用的IP协议版本，AF_INET：IPv4；AF_INET6：IPv6
export HCCL_SOCKET_FAMILY=AF_INET       #IPv4

# 支持以下格式的网卡名配置（4种规格自行选择1种即可，环境变量中可配置多个网卡，取最先匹配到的网卡作为root网卡）

# 精确匹配网卡
export HCCL_SOCKET_IFNAME==eth*,enp***   # 使用指定的eth*或enp**网卡
export HCCL_SOCKET_IFNAME=^=eth*,enp***     # 不使用指定的eth*或enp**网卡

# 模糊匹配网卡
export HCCL_SOCKET_IFNAME=eth,enp       # 使用所有以eth或enp为前缀的网卡
export HCCL_SOCKET_IFNAME=^eth,enp      # 不使用任何以eth或enp为前缀的网卡
```

需要注意：

MPI工具执行时，会将环境变量同步到所有节点，如果参与集合通信的不同节点的网卡名字不同，例如node1的网卡名为eth1，node2的网卡名为eth2，则需要使用模糊匹配网卡的方式进行环境变量配置。

10.3.16 hostfile 与测试命令不匹配报错 retcode 11

问题现象

多机场景下执行HCCL Test测试工具时，报错：This is an error in init_hcclComm.retcode: 11，如下所示：

原因分析

- hostfile文件中配置的每个节点使用的卡数和测试命令-p不匹配，“-p”代表每个节点使用多少张卡（即每个节点的进程数）。
- hostfile文件中配置的节点数量和mpirun -n数量不匹配，“-n”代表总共使用多少张卡（即节点数量*每个节点的卡数）。

mpirun命令示例：

mpirun -f hostfile **-n 16** ./bin/all_reduce_test -b 8K -e 4G -f 2 -d fp32 -o sum **-p 8**

解决步骤

修改hostfile配置文件或者mpirun测试命令，使二者配置的总卡数以及每个节点使用的卡数保持一致。

10.3.17 大规模集群场景报建链失败错误

问题现象

大规模集群场景测试失败，报建链失败的错误，但是拆分成小规模集群分别测试正常。

原因分析

- Master节点需要建链和处理数据量较大，建链默认等待超时时间120s无法满足需求，导致建链失败。
- Master节点允许处理的并发建链数受linux内核参数“somaxconn”与“tcp_max_syn_backlog”的限制，所以，针对大规模集群组网，若“somaxconn”与“tcp_max_syn_backlog”取值较小会导致部分客户端概率性提前异常退出，导致集群初始化失败。

解决步骤

步骤1 调大建链超时等待时间。

通过环境变量HCCL_CONNECT_TIMEOUT（单位为s）进行设置，默认值为120s，建议根据集群组网的规模适当调大。例如：

```
# 3K卡场景
export HCCL_EXEC_TIMEOUT=240

# 5K卡场景
export HCCL_EXEC_TIMEOUT=600
```

步骤2 根据集群数量调整“somaxconn”与“tcp_max_syn_backlog”参数的值。

需要注意，所有机器的OS都需要配置，包括裸机、镜像环境，配置示例如下：

```
sysctl -w net.core.somaxconn=65535
sysctl -w net.ipv4.tcp_max_syn_backlog=65535
```

----结束

10.3.18 大规模集群场景 HCCP 报 errno 24

问题现象

大规模集群场景测试失败，HCCP报错errno:24，错误截图如下所示，但是拆分成小规模集群分别测试正常。

```
[ERROR] HCCP（2726987,al1_reduce_test)）:2024-01-12-17:52:59.187.615
[rs_socket.c:570]tid:2727604,rs_epoll_even_listen_in_handle(570) : IP:*.*.*.* accept( ) failed!errno:24
[ERROR] HCCP（2726987,al1_reduce_test)）:2024-01-12-17:52:59.187.615
[rs_socket.c:570]tid:2727604,rs_epoll_even_listen_in_handle(570) : IP:*.*.*.* accept( )
failed!errno:24
```

执行“ulimit -a”命令查询“open files”的值（默认为1024），而测试卡数的规模接近open files的值。


```
[root@hccl137 ~]# ulimit -a
real-time non-blocking time  (microseconds, -R) unlimited
core file size              (blocks, -c) 0
data seg size               (kbytes, -d) unlimited
scheduling priority                 (-e) 0
file size                   (blocks, -f) unlimited
pending signals                     (-i) 3092387
max locked memory           (kbytes, -l) 65536
max memory size             (kbytes, -m) unlimited
open files                          (-n) 1024
pipe size                (512 bytes, -p) 8
POSIX message queues         (bytes, -q) 819200
real-time priority                  (-r) 0
stack size                  (kbytes, -s) 8192
cpu time                   (seconds, -t) unlimited
max user processes                  (-u) 3092387
virtual memory              (kbytes, -v) unlimited
file locks                          (-x) unlimited
[root@hccl137 ~]#
```

原因分析

open files太小，测试卡数规模接近open files，导致测试报错

解决步骤

修改所有机器（包括裸机、镜像环境）的open files大小，建议配置为1000000，配置方法如下：

```
echo “ulimit -n 1000000”>> /etc/profile
source /etc/profile
```

【解决步骤】

所有机器裸机和docker均修改open files大小，建议配置成1000000，例如：

```
echo “ulimit -n 1000000”>> /etc/profile
source /etc/profile
```

执行“cat/proc/<PID>/limits”查看某个进程的soft limit和hard limit限制 ，有可能soft limit少但ulimit -a查出来是很大，PID通过npu-smi看卡的进程PID，该情况需要同步修改。

```
(base) root@box-24:~# cat /proc/27512/limits
Limit                     Soft Limit           Hard Limit           Units
Max cpu time              unlimited            unlimited            seconds
Max file size             unlimited            unlimited            bytes
Max data size             unlimited            unlimited            bytes
Max stack size            8388608              unlimited            bytes
Max core file size        0                    unlimited            bytes
Max resident set          unlimited            unlimited            bytes
Max processes             2062502              2062502              processes
Max open files            1024                 1048576              files
Max locked memory         67598909440          67598909440          bytes
Max address space         unlimited            unlimited            bytes
Max file locks            unlimited            unlimited            locks
Max pending signals       2062502              2062502              signals
Max msgqueue size         819200               819200               bytes
Max nice priority         0                    0
Max realtime priority     0                    0
Max realtime timeout      unlimited            unlimited            us
```

