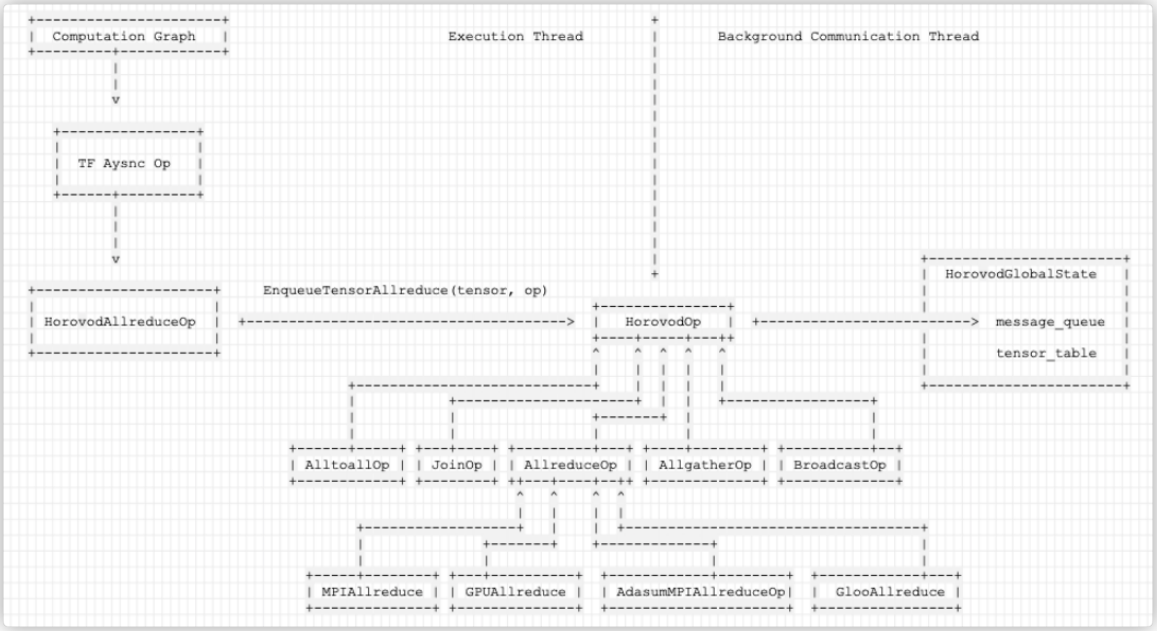
(6) --- 后台线程架构

0x01 引子

在前文我们看到，当训练时，Execution Thread 会通过一系列操作，把 Tensor & Operation 传递给后台线程，其流程大致如下：

```
   IndexedSlices
        +
        |
        |
        v
     allreduce
        +
        |
        |
        v
     allgather
        +
        |
        |
        v
  HorovodAllgather
        +
        |
        v
  HorovodAllgatherOp
        +
        |
        |
        v
 EnqueueTensorAllgather
        +
        |
        |
        v
+-------+-------------+
| HorovodGlobalState  |
|                     |
|      message_queue  |
|                     |
|       tensor_table  |
|                     |
+---------------------+
```

或者如下图，左面是 执行线程，就是训练线程，右面是后台线程，用来做 ring-allreduce：



我们下面继续看看后台是如何运作的。

0x02 设计要点

2.1 问题

因为计算框架往往采用多线程执行训练的计算图，所以在多节点情况下，拿allreduce操作来举例，我们不能保证每个节点上的 allreduce 请求是有序的。因此MPI_Allreduce并不能直接用。

2.2 方案

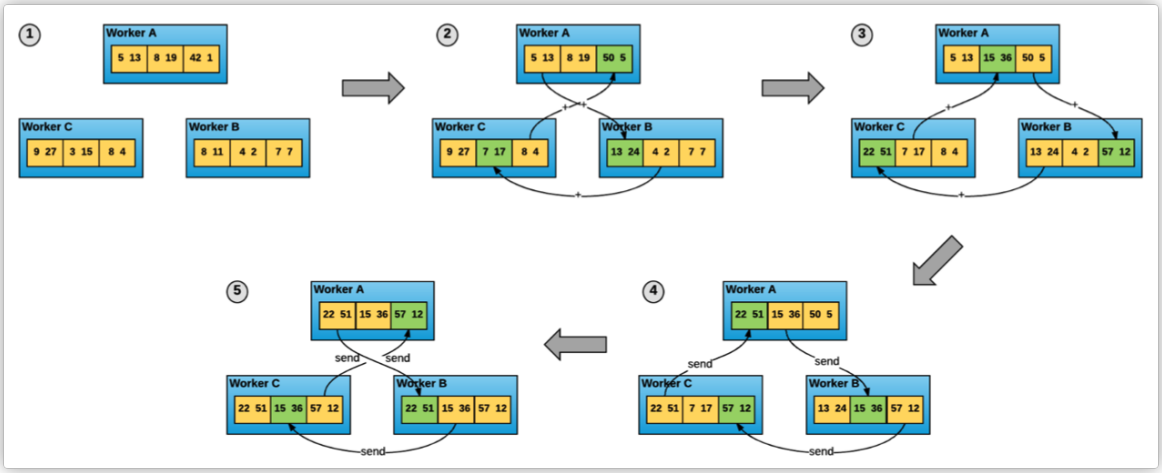
为了解决这个问题，hvd 设计了一个主从模式，rank 0 为 master 节点，rank 1 ~ rank n 为 worker 节点。

- master 节点进行同步协调，保证对于某些 tensor 的 allreduce 请求最终有序 & 完备，可以继续处理。
- 在决定了哪些 tensor 以后，master又会将可以进行通信的tensor 名字和顺序发还给各个节点。
- 当所有的节点都得到了即将进行的MPI的tensor和顺序，MPI通信得以进行。

首先回顾下同步梯度更新这个概念，其表示的是等待 所有Rank的梯度都计算完毕后，再统一做全局梯度累加，这就涉及到在集群中做消息通信，为此HVD做了两个方面的工作。

- 在Horovod中，每张卡都对应一个训练进程，称之为rank。如4张卡，对应的各个进程的rank则为[0,1,2,3]。
- **协调工作**：HVD里面将 Rank0 作为coordinator（master），其余的进程为worker。由Rank0来协调所有Rank的进度。
- **后台线程**：为了不block正常OP的计算，HVD里面创建 background communication 线程，专门用来Rank间的消息同步和AllReduce操作。

在 Horovod 中，训练进程是平等的参与者，每个进程既负责梯度的分发，也负责具体的梯度计算。如下图所示，三个 Worker 中的梯度被均衡地划分为三份，通过 4 次通信，能够完成集群梯度的计算和同步。



2.3 协调

2.3.1 设计

对于协调的过程，文档中也有非常详细的讲述，我也一起翻译。

> coordinator 目前采用master-worker paradigm。Rank 0 作为master（即 "coordinator"），其他的rank是 worker。每个 rank 在自己的后台线程中运行，时间片循环调度处理。在每个时间片中会进行如下操作：
>
> - Workers 会发送 MPIRequests 给 coordinator。MPIRequests 显式注明 worker 希望做什么（比如在哪个 tensor 上做什么操作，是 gather 还是 reduce，以及 tensor 的形状和类型）。在 tensor 的 collective op 已经执行完 ComputeAsync 之后，worker 就会对于每个 tensor 发送MPIRequest。
> - 当没有更多处理的 tensors 之后，workers 会向 coordinator 发送一个空的 "DONE" 消息；
> - coordinator 从 worker 收到 MPIRequests 以及 coordinator本身的 TensorFlow ops 之后，将它们存储在请求表中（request table）。协调器继续接收MPIRequest，直到收到了 MPI_SIZE 个 "DONE" 消息；
> - Coordinator 收集所有准备缩减，gather 的张量，或所有导致错误的操作。对于每一个向量或者操作。Coordinator 向所有工作人员发送MPIResponse。当没有更多的MPIResponse 时，Coordinator将向工人发送“完成”响应。如果进程正在关闭，它将发送一个“shutdown”响应。
> - Workers 监听MPIResponse消息，逐个做所要求的reduce或gather操作，直到他们收到"DONE" resposne。此时，时间片结束。如果接收到的不是“DONE”，而是“SHUTDOWN”，则退出background loop

简单来讲就是：

- Coordinator 收集所有 worker（包括Coordinator自己，因为自己也在进行训练）的 MPIRequests，把他们放入request table。
- 当收集到 MPI_SIZE 个 "DONE" 消息之后，Coordinator 会找出就绪的 tensor （在 message_table 里面查找）构造出一个 read_to_reduce 的列表，然后发出 size 个 MPIResponse 告知进程进行计算。
- worker 接受到 response 开始真正的计算过程(通过 op_manager 具体执行)。
- 这是整体同步的过程，如果打开 horovod 的 trace log(`HOROVOD_LOG_LEVEL=trace`) 就能看到同步的过程。

2.3.2 实现

我们再具体看看实现。

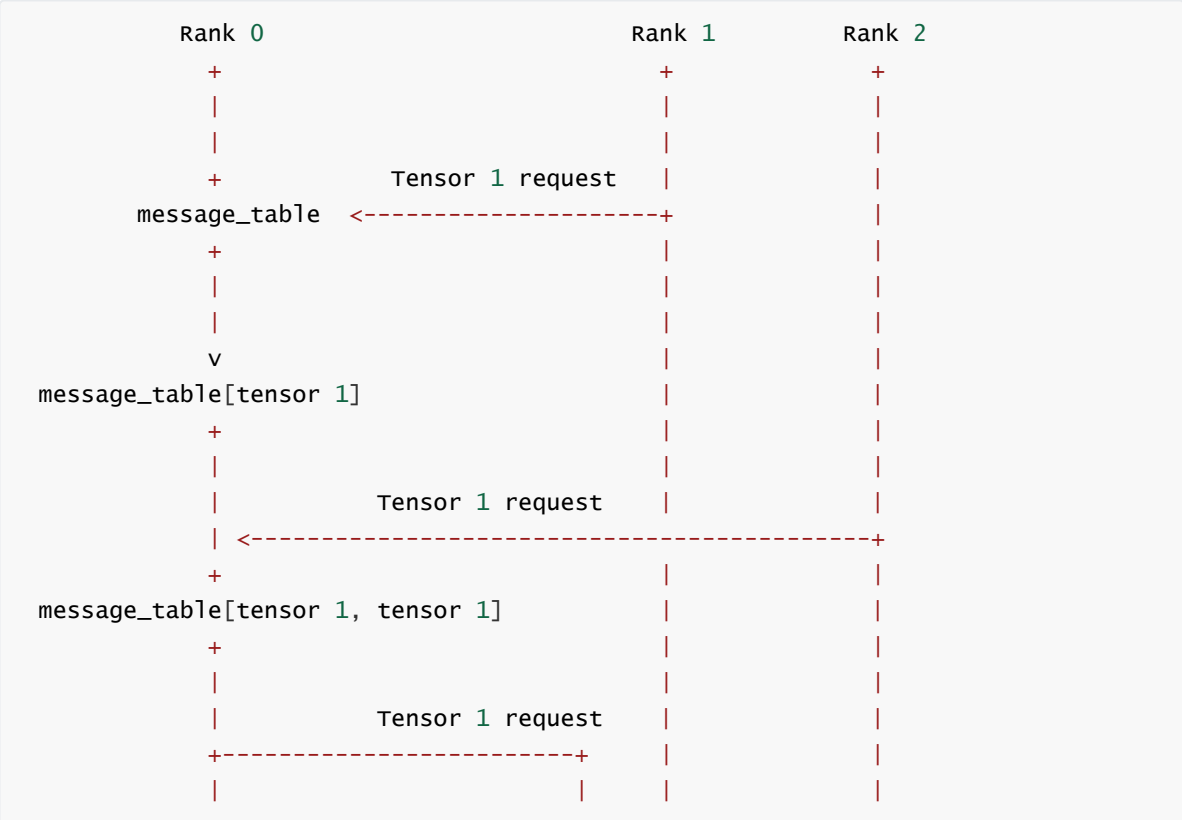
在Horovod中，每张卡都对应一个训练进程，称之为rank。如4张卡，对应的各个进程的rank则为[0,1,2,3]。

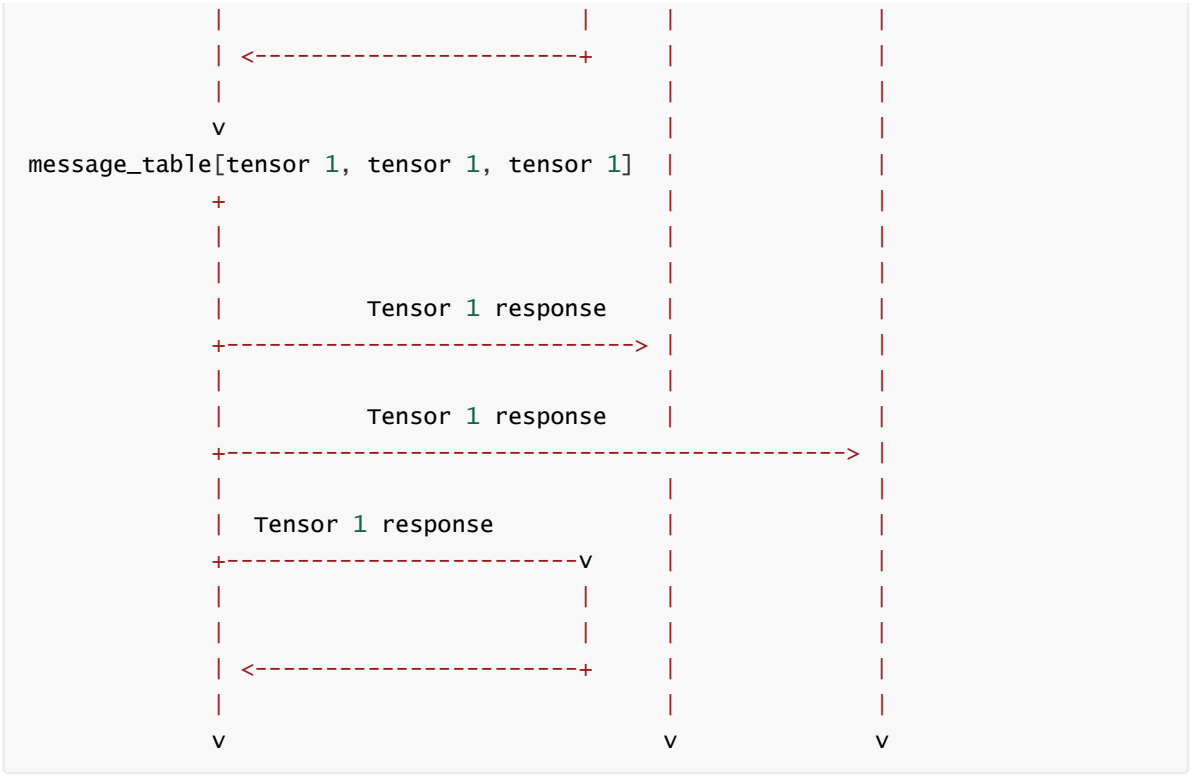
hvd 设计了一个主从模式，将 Rank0 作为coordinator（master），其余的进程为worker，由Rank0来协调所有Rank的进度。每个worker节点上都有一个消息队列，而在master节点上除了一个消息队列，还有一个消息map。

每当计算框架发来通信请求时，hvd并不直接执行MPI，而是封装了这个消息并推入自己的消息队列。

- 整体采用消息的 Request 和 Response 机制；
- 当某个 OP 的 gradient 计算完成并且等待全局的 AllReduce，该 Rank 就会包装一个 Request 请求，调用 ComputeResponseList 将 Request （就是说，**这是个 ready tensor**）放入这个 rank 的 message_queue 中，每个 Rank 的 后台线程 定期轮训自己的 message_queue，然后把 queue 里面的 request 发送到 Rank 0。因为是同步MPI，所以每个节点会阻塞等待MPI完成。
- Rank 0 拥有 message_table，用来保存其他 rank 的 request 信息，rank 0 会处理 message_table 里面所有的 request。
- 当 rank 0 收到 所有 rank 对于某个 op allreduce 的 request 之后，就说明 这个 tensor 在所有的 rank中都已经ready。说明 所有的节点都已经发出了对该tensor的通信请求，那这个tensor就需要且能够进行通信。
- 决定了tensor以后，master又会将可以进行通信的tensor 名字和顺序发还给各个节点。
    - Rank 0 节点会挑选出所有符合要求的tensor进行MPI通信：
    - 不符合要求的tensor继续留在消息map中，等待条件符合。
    - 当有符合要求的 tensor，Rank 0 然后就会发送 Response 给其他 rank，表明当前 op & tensor 的所有局部梯度已经 Ready，可以对这个tensor执行**collective操作**，比如可以执行 allReduce 操作。
- 至此，所有的节点都得到了即将进行的MPI的tensor和顺序，MPI通信得以进行。

大致逻辑如下：





2.4 Background Thread

每个rank有两个thread，我们通常在python文件中使用hvd.init()来初始化hvd，实际上是开了一个后台线程和一个MPI线程。

- Execution thread（MPI线程） 是用来做机器学习计算的。
- background thread 是 rank 之间同步通讯和做allreduce操作的。百度在设计时候，就有了一个 MPI background thread，Horovod沿用了这个设计，名字就是BackgroundThreadLoop。

2.4.1 设计

关于设计的思考，百度在源码注释（tensorflow-allreduce-master/tensorflow/contrib/mpi_collectives/mpi_ops.cc）里面写的非常清楚，我大致翻译出来。

> MPI background thread 是为了协调所有的 MPI 进程和tensor reduction。这个设计是处于几个考虑：
>
> 1. 一些MPI实现要求所有的MPI调用必须在一个单独线程中。因为 Tensorflow 在处理图的时候可能会用到几个线程，所以我们必须使用自己的特定的线程来处理MPI；
> 2. 对于某些错误（比如不匹配的types），MPI 有时候会没有一个确定的处理方式，但是我们还想优雅的处理这些错误。为了做到优雅处理，就要求 MPI 进程需要知道其他进程上tensor的形状和类型；
> 3. MPI reductions and gathers 也许会和其他操作一起并行处理。因为 MPI 使用一个与TF GPUDevice streams分离的内部（inaccessible）的GPU stream，我们不能显式进行同步memcpys或者kernels。因此，MPIAllreduce and MPIAllgather 必须是 AsyncOpKernels 类型 以便 确保memcpys或者kernels的合理顺序；
> 4. 注意：我们无法确保所有的MPI进程以同样的顺序reduce他们的tensors。因此，必须有一个办法来确保可以同时跨越所有的ranks来做reduction memcpys and kernels。我们使用 rank ID 0 作为 coordinator 来协调那些已经准备好的，可以执行的操作（gather and trigger the reduction operations）；

精简下：

1. 一些MPI的实现机制要求所有的MPI调用必须在一个单独线程中。
2. 为了处理错误，MPI 进程需要知道其他进程上tensor的形状和类型。

3. MPIAllreduce and MPIAllgather 必须是 AsyncOpKernels 类型 以便 确保memcpys或者kernels的合理顺序。

因此，一个后台线程是有必要的。horovod_global.message_queue 以及 horovod_global.tensor_table 都是在Horovod的后台线程BackgroundThreadLoop 中被处理的。

2.4.2 实现

在底层，AllReduce 被注册为 Op，在 ComputeAsync 中，计算请求被入队到一个队列中。这一队列会被一个统一的后台线程处理。

在这个后台线程的初始化过程中，它会利用进程内共享的全局状态在自己的内存里创建一些对象，以及一些逻辑判断。比如要不要进行 Hierarchical AllReduce，要不要 AutoTune等。这里是初始化阶段的日志。

在初始化的过程中，有一些比较重要的对象会被构造出来，比如各种 Controller。

我们接下来就具体分析后台线程。

0x03 辅助功能

我们首先介绍一些辅助功能。

3.1 如何判断是 coordinator

因为后台线程代码是所有worker公用，所以需要区分 rank0 还是其他 worker，从而执行不同的代码流程。

这里采用 is_coordinator 用来判断是否是 Rank0。

is_coordinator_ 的赋值如下：

```
void MPIController::DoInitialization() {
  ......

  // Get MPI rank to determine if we are rank zero.
  MPI_Comm_rank(mpi_ctx_.mpi_comm, &rank_);
  is_coordinator_ = rank_ == 0;
```

is_coordinator_ 的使用方式示例如下，可以看出来，在同步参数的时候，是从 rank 0 获取参数，然后广播给其他 rank，即 workers：

```
void Controller::SynchronizeParameters() {
  ParameterManager::Params param;
  if (is_coordinator_) { // rank 0 执行操作
    param = parameter_manager_.GetParams();
  }

  void* buffer = (void*)(&param);
  size_t param_size = sizeof(param);
  Bcast(buffer, param_size, 0, Communicator::GLOBAL);

  if (!is_coordinator_) { // worker 执行操作
    parameter_manager_.SetParams(param);
  }
}
```

3.2 协调缓存&信息

在 ComputeResponseList 函数中，会使用以下代码来协调缓存，作用就是整理出来所有 rank 共有的 tensor。

```
CoordinateCacheAndState(cache_coordinator);
```

主要还是用到了cache_coordinator 操作。

```
void Controller::CoordinateCacheAndState(CacheCoordinator& cache_coordinator) {
  // Sync cache and state information across workers.
  cache_coordinator.sync(shared_from_this(), timeline_enabled_);
}
```

3.2.1 计算共有 tensor

CoordinateCacheAndState 函数如下：

- 每个worker都整理自己的bitvector;
- 使用 CrossRankBitwiseAnd 整理出来共有的 tensor；
- 使用 CrossRankBitwiseOr 整理出来共有的无效 tensor；

```
void CacheCoordinator::sync(std::shared_ptr<Controller> controller,
                            bool timeline_enabled) {

  // Resize and initialize bit vector.
  int nbits = num_active_bits_ + NUM_STATUS_BITS;
  int count = (nbits + sizeof(long long) * CHAR_BIT - 1) /
              (sizeof(long long) * CHAR_BIT);

  ......

  // 每个worker都整理自己的bitvector
  // For each cache hit on this worker, flip associated bit in bit vector.
  for (auto bit : cache_hits_) {
    int shifted_bit = bit + NUM_STATUS_BITS;
    int shift = shifted_bit / (sizeof(long long) * CHAR_BIT);
    bitvector_[shift] |=
        (1ull << (shifted_bit % (sizeof(long long) * CHAR_BIT)));
    if (timeline_enabled) {
      // Set corresponding bit in extended section for timeline if needed.
      bitvector_[count + shift] ^=
          (1ull << (shifted_bit % (sizeof(long long) * CHAR_BIT)));
    }
  }

  // 整理出来共有的 tensor
  // Global AND operation to get intersected bit array.
  controller->CrossRankBitwiseAnd(bitvector_, fullcount);

  // Search for flipped bits to populate common cache hit set. There will never
  // be invalid bits in this set.
  cache_hits_.clear();
  for (int i = 0; i < count; ++i) {
    int shift = i * sizeof(long long) * CHAR_BIT;
    long long ll = bitvector_[i];
```

```
    while (ll) {
      int idx = __builtin_ffsll(ll);
      int shifted_bit = shift + idx - 1;
      cache_hits_.insert(shifted_bit - NUM_STATUS_BITS);
      ll &= ~(1ull << (idx - 1));
    }
  }

  ......

  // If any worker has invalid cache entries, communicate invalid bits across
  // workers using a second bit-wise allreduce operation.
  if (invalid_in_queue_) {
    std::memset(&bitvector_[0], 0, count * sizeof(long long));
    for (auto bit : invalid_bits_) {
      int shift = bit / (sizeof(long long) * CHAR_BIT);
      bitvector_[shift] |= (1ull << (bit % (sizeof(long long) * CHAR_BIT)));
    }

    // Global OR operation to get common invalid bits.
    controller->CrossRankBitwiseOr(bitvector_, count);
    // Search for flipped bits to populate common invalid bit set.
    invalid_bits_.clear();
    for (int i = 0; i < count; ++i) {
      int shift = i * sizeof(long long) * CHAR_BIT;
      long long ll = bitvector_[i];
      while (ll) {
        int idx = __builtin_ffsll(ll);
        int bit = shift + idx - 1;
        invalid_bits_.insert(bit);
        ll &= ~(1ull << (idx - 1));
      }
    }
  }

  synced_ = true;
}
```

3.2.2 MPI操作

CrossRankBitwiseAnd 作用是 调用 MPI 归并 共有的 bitvector。

```
void MPIController::CrossRankBitwiseAnd(std::vector<long long>& bitvector,
                                        int count) {
  int ret_code = MPI_Allreduce(MPI_IN_PLACE, bitvector.data(), count,
                               MPI_LONG_LONG_INT, MPI_BAND, mpi_ctx_.mpi_comm);
}
```

3.3 MPIContext

mpi_context 是在加载 C++ 的代码时候就已经创建了，同时创建的还有其他 context （ nccl_context, gpu_context），主要是维护一些节点上 mpi 通信的必要环境信息和设置，如：

- 3 个 MPI communicator，mpi_comm，local_comm，cross_comm 分别负责 horovod mpi 传输，节点内传输，和节点间分层传输（主要用于 hierarchical allreduce）。
- mpi_float16_t： horovod 主要以 float16 传输。

- mpi_float16_sum： float16 对应的sum 操作。

在 horovod 使用 mpi 的时候，都会使用上面的 communicator 进行数据传输。

3.4 Parameter_manager

Parameter_manager 主要是 GlobalState 的一个用于管理一些调节 horovod 性能的参数的管理器，在 BackgroundThreadLoop 中跟其他的 GlobalState 的元素一同初始化，然后会读取下面这些对应的环境变量，然后进行设置。

- **HOROVOD_FUSION_THRESHOLD**：指传输数据切片的大小，默认是64M，如果切片太大，传输的时候就不能很好地 pipeline 传输，如果太小，一个 tensor 需要传输多次，增加 IO 的 overhead。
- **HOROVOD_CYCLE_TIME**：指 RunLoopOnce 的睡眠时长，默认是 5ms，比较理想的睡眠时间应该是 RunLoopOnce 其余逻辑处理的时间 + HOROVOD_CYCLE_TIME 刚好等于一次前向传播和后向传播所用的时间，因为睡太久前端会在等 RunLoopOnce 睡醒；如果睡太短，不断地跑一次 RunLoopOnce，tensor_queue 也不会有新的元素，只是白跑。
- **HOROVOD_CACHE_CAPACITY**：指 cache 的大小，这个可能跟 model 层数参数量相关了。
- **HOROVOD_HIERARCHICAL_ALLGATHER**：是否使用分层的 allgather 的方式等

Parameter_manager 也提供了对这些参数自动调节的功能。通过 Parameter_manager.SetAutoTuning 进行设置，设置后会在初始的几个 batch 尝试不同的参数组合进行通信，后面会收敛到一组最优的参数值。

0x04 总体代码

4.1 后台线程

BackgroundThreadLoop 是训练过程中的后台线程，主要负责跟其他节点的通信，和处理前端过来的通信需求（request），会轮询调用 RunLoopOnce，不断查看 tensor_queue 中有没有需要通信的 tensor，如果有跟其他节点同步更新，然后执行通信操作。

在 BackgroundThreadLoop 函数 可以看到基本逻辑：

- 依据编译配置，决定如何初始化，比如 mpi_context.Initialize 只有在 MPI 编译时候才初始化。
- 初始化 controller，会根据加载的集合通讯库（mpi 或者 gloo）为 globalstate 创建对应的 controller；
- 得到各种配置，比如 local_rank；
- 设置 background thread affinity；
- 设置 GPU stream；
- 设置 timeline 配置；
- 设置 Tensor Fusion threshold，cycle time，response cache capacity，flag for hierarchical allreduce.....；
- 设置 auto-tuning， chunk size；
- 重置 operation manager；
- 进入关键代码 RunLoopOnce；

缩减版代码如下：

```
BackgroundThreadLoop(HorovodGlobalState& state) {
 ......

#if HAVE_MPI
  // Initialize mpi context
#if HAVE_DDL
  // If DDL is enabled, let DDL ops manage MPI environment.
```

```
  auto mpi_ctx_manager = DDL_MPIContextManager(ddl_context, gpu_context);
#else
  // Otherwise, let MPI ops be in charge.
  auto mpi_ctx_manager = MPIContextManager();
#endif
  // mpi_context 会根据前端和环境变量传过来的信息，创建 mpi 线程，和一些 mpiOps
  mpi_context.Initialize(state.controller->GetRanks(), mpi_ctx_manager);
#endif

  ......

  // 会同步不同 node 的 global_size, local_size, rank, is_coordinator 等信息
  // Initialize controller
  state.controller->Initialize();

  int local_size = state.controller->GetLocalSize();
  int local_rank = state.controller->GetLocalRank();

  ......

  // 设置op_manager，这里主要是注册不同的集合通信库的 ops
  op_manager.reset(CreateOperationManager(state));

  // Signal that initialization is completed.
  state.initialization_done = true;

  // Iterate until shutdown.
  try {
    while (RunLoopOnce(state));
  } catch (const std::exception& ex) {
    LOG(ERROR) << "Horovod background loop uncaught exception: " << ex.what();
  }
}
```

4.2 哪里建立环

也许大家会有疑问，既然 Horovod 是 ring Allreduce，但是究竟是在哪里建立了环？我们选几种实现来大致看看。因为如果细致研究就需要深入MPI，gloo等，这已经超出了本文范畴，所以我们只是大致了解。

4.2.1 NCCL 调用

我们首先看看 NCCL。

4.2.1.1 NCCL

NCCL是Nvidia Collective multi-GPU Communication Library的简称，它是一个实现多GPU的collective communication通信（all-gather, reduce, broadcast）库，Nvidia做了很多优化，以在PCIe、Nvlink、InfiniBand上实现较高的通信速度。

4.2.1.2 Horovod

在 NCCLAllreduce::Execute 我们可以看到，调用了ncclAllReduce，这是 nccl 的 API，因此我们可以推断，其参数 `*nccl_op_context_.nccl_comm_` 应该是关键。

```
Status NCCLAllreduce::Execute(std::vector<TensorTableEntry>& entries,
                              const Response& response) {

  // Do allreduce.
  auto nccl_result = ncclAllReduce(fused_input_data, buffer_data,
                                   (size_t) num_elements,
                                   GetNCCLDataType(first_entry.tensor), ncclSum,
                                   *nccl_op_context_.nccl_comm_,
*gpu_op_context_.stream);
}
```

nccl_op_context_ 是 NCCLOpContext 类型，NCCLOpContext 简化版定义如下：

```
class NCCLOpContext {
public:
  void InitNCCLComm(const std::vector<TensorTableEntry>& entries,
                    const std::vector<int32_t>& nccl_device_map);

  ncclComm_t* nccl_comm_;
};
```

所以我们来看其参数 `nccl_comm_` 是如何初始化的，可以看到其调用了 ncclCommInitRank 进行初始化。

```
void NCCLOpContext::InitNCCLComm(const std::vector<TensorTableEntry>& entries,
                                 const std::vector<int32_t>& nccl_device_map) {
  // Ensure NCCL communicator is in the map before executing operation.
  ncclComm_t& nccl_comm = nccl_context_->nccl_comms[global_state_-
>current_nccl_stream][nccl_device_map];
  if (nccl_comm == nullptr) {
    auto& timeline = global_state_->timeline;
    timeline.ActivityStartAll(entries, INIT_NCCL);

    int nccl_rank, nccl_size;
    Communicator nccl_id_bcast_comm;
    // 获取rank相关信息
    PopulateNCCLCommStrategy(nccl_rank, nccl_size, nccl_id_bcast_comm);

    ncclUniqueId nccl_id;
    global_state_->controller->Bcast((void*)&nccl_id, sizeof(nccl_id), 0,
                                     nccl_id_bcast_comm);

    ncclComm_t new_nccl_comm;
    // 这里调用了nccl，传递了rank信息
    auto nccl_result = ncclCommInitRank(&new_nccl_comm, nccl_size, nccl_id,
nccl_rank);
    nccl_context_->ErrorCheck("ncclCommInitRank", nccl_result, nccl_comm);
    nccl_comm = new_nccl_comm;

    // Barrier helps NCCL to synchronize after initialization and avoid
    // deadlock that we've been seeing without it.
    global_state_->controller->Barrier(Communicator::GLOBAL);
    timeline.ActivityEndAll(entries);
  }

  nccl_comm_ = &nccl_comm;
```

```
}
```

PopulateNCCLCommStrategy就是从全局状态中获取rank信息。

```
void NCCLOpContext::PopulateNCCLCommStrategy(int& nccl_rank, int& nccl_size,
                                             Communicator& nccl_id_bcast_comm) {
  if (communicator_type_ == Communicator::GLOBAL) {
    nccl_rank = global_state_->controller->GetRank();
    nccl_size = global_state_->controller->GetSize();
  } else if (communicator_type_ == Communicator::LOCAL) {
    nccl_rank = global_state_->controller->GetLocalRank();
    nccl_size = global_state_->controller->GetLocalSize();
  } else {
    throw std::logic_error("Communicator type " +
std::to_string(communicator_type_) +
                           " is not supported in NCCL mode.");
  }
  nccl_id_bcast_comm = communicator_type_;
}
```

于是我们得去 NCCL 源码中看看。

4.2.1.3 In NCCL

在 init.cc 中可以看到

```
NCCL_API(ncclResult_t, ncclCommInitRank, ncclComm_t* newcomm, int nranks,
ncclUniqueId commId, int myrank);
ncclResult_t ncclCommInitRank(ncclComm_t* newcomm, int nranks, ncclUniqueId
commId, int myrank) {
  NVTX3_FUNC_RANGE_IN(nccl_domain);
  int cudaDev;
  CUDACHECK(cudaGetDevice(&cudaDev));
  // 这里初始化
  NCCLCHECK(ncclCommInitRankDev(newcomm, nranks, commId, myrank, cudaDev));
  return ncclSuccess;
}
```

继续看，调用了 ncclAsyncInit 来完成最后初始化，传入了总体rank数目，进程自身的myrank。

```
static ncclResult_t ncclCommInitRankDev(ncclComm_t* newcomm, int nranks,
ncclUniqueId commId, int myrank, int cudaDev) {
  ncclResult_t res;
  char* env = getenv("NCCL_COMM_ID");

  NCCLCHECKGOTO(ncclInit(), res, end);
  // Make sure the CUDA runtime is initialized.
  CUDACHECKGOTO(cudaFree(NULL), res, end);
  NCCLCHECKGOTO(PtrCheck(newcomm, "CommInitRank", "newcomm"), res, end);

  if (ncclAsyncMode()) {
    // 调用了 ncclAsyncInit 来完成最后初始化，传入了总体rank数目，进程自身的myrank
    NCCLCHECKGOTO(ncclAsyncInit(ncclCommInitRankSync, newcomm, nranks, commId,
myrank, cudaDev), res, end);
  } else {
```

```
    NCCLCHECKGOTO(ncclCommInitRankSync(newcomm, nranks, commId, myrank,
cudaDev), res, end);
  }

end:
  if (ncclAsyncMode()) return ncclAsyncErrCheck(res);
  else return res;
}
```

ncclComm_t 实际是 ncclComm 的typedef，因此我们看看ncclComm定义，其中就包括了总体rank数目，进程自身的myrank。

```
struct ncclComm {
  struct ncclChannel channels[MAXCHANNELS];
  ...
  // Bitmasks for ncclTransportP2pSetup
  int connect;
  uint32_t* connectSend;
  uint32_t* connectRecv;

  int rank;    // my rank in the communicator
  int nRanks;  // number of GPUs in communicator
  int cudaDev; // my cuda device index
  int64_t busId;   // my PCI bus ID in int format

  int node;
  int nNodes;
  int localRanks;

  // Intra-process sync
  int intraRank;
  int intraRanks;
  int* intraBarrier;
  int intraPhase;
  ....
};
```

因此，我们大致可以了解，horovod 把 rank 信息传进来，NCCL 会据此组环。

4.2.2 GLOO

在 GlooContext::Initialize 之中可以看到，Horovod 通过 Rendezvous 把 rank 信息发给了 Rendezvous Server。

Gloo 内部会进行组环。

其中，cross_rank 是hierarchical allreduce所需要的。

```
void GlooContext::Initialize(const std::string& gloo_iface) {

  attr device_attr;
  device_attr.iface = gloo_iface;

  device_attr.ai_family = AF_UNSPEC;
  auto dev = CreateDevice(device_attr);
  auto timeout = GetTimeoutFromEnv();
```

```
  auto host_env = std::getenv(HOROVOD_HOSTNAME);
  std::string hostname = host_env != nullptr ? std::string(host_env) :
std::string("localhost");

  int rank = GetIntEnvOrDefault(HOROVOD_RANK, 0);
  int size = GetIntEnvOrDefault(HOROVOD_SIZE, 1);
  int local_rank = GetIntEnvOrDefault(HOROVOD_LOCAL_RANK, 0);
  int local_size = GetIntEnvOrDefault(HOROVOD_LOCAL_SIZE, 1);
  int cross_rank = GetIntEnvOrDefault(HOROVOD_CROSS_RANK, 0);
  int cross_size = GetIntEnvOrDefault(HOROVOD_CROSS_SIZE, 1);

  auto rendezvous_addr_env = std::getenv(HOROVOD_GLOO_RENDEZVOUS_ADDR);
  auto rendezvous_port = GetIntEnvOrDefault(HOROVOD_GLOO_RENDEZVOUS_PORT, -1);

  bool elastic = GetBoolEnvOrDefault(HOROVOD_ELASTIC, false);
  if (elastic && reset_) {
    std::string server_addr = rendezvous_addr_env;
    std::string scope = HOROVOD_GLOO_GET_RANK_AND_SIZE;
    HTTPStore init_store(server_addr, rendezvous_port, scope, rank);

    auto key = hostname + ":" + std::to_string(local_rank);
    std::vector<char> result = init_store.get(key);
    std::string s(result.begin(), result.end());
    std::stringstream ss(s);

    int last_rank = rank;
    int last_size = size;
    int last_local_rank = local_rank;
    int last_local_size = local_size;
    int last_cross_rank = cross_rank;
    int last_cross_size = cross_size;

    rank = ParseNextInt(ss);
    size = ParseNextInt(ss);
    local_rank = ParseNextInt(ss);
    local_size = ParseNextInt(ss);
    cross_rank = ParseNextInt(ss);
    cross_size = ParseNextInt(ss);

    SetEnv(HOROVOD_RANK, std::to_string(rank).c_str());
    SetEnv(HOROVOD_SIZE, std::to_string(size).c_str());
    SetEnv(HOROVOD_LOCAL_RANK, std::to_string(local_rank).c_str());
    SetEnv(HOROVOD_LOCAL_SIZE, std::to_string(local_size).c_str());
    SetEnv(HOROVOD_CROSS_RANK, std::to_string(cross_rank).c_str());
    SetEnv(HOROVOD_CROSS_SIZE, std::to_string(cross_size).c_str());
  }

  // 设定了不同的 Rendezvous server
  ctx = Rendezvous(HOROVOD_GLOO_GLOBAL_PREFIX,
                   rendezvous_addr_env, rendezvous_port,
                   rank, size, dev, timeout);

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

  cross_ctx = Rendezvous(HOROVOD_GLOO_CROSS_PREFIX + std::to_string(local_rank),
                         rendezvous_addr_env, rendezvous_port,
```

```
                          cross_rank, cross_size, dev, timeout);
}
```

4.2.3 MPI

MPIContext::Initialize 中可以看到，在这会设置各种 rank。

```
void MPIContext::Initialize(const std::vector<int>& ranks,
                            MPIContextManager& ctx_manager) {

  auto mpi_threads_disable = std::getenv(HOROVOD_MPI_THREADS_DISABLE);
  int required = MPI_THREAD_MULTIPLE;
  if (mpi_threads_disable != nullptr &&
      std::strtol(mpi_threads_disable, nullptr, 10) > 0) {
    required = MPI_THREAD_SINGLE;
  }
  int is_mpi_initialized = 0;
  MPI_Initialized(&is_mpi_initialized);
  if (is_mpi_initialized) {
    int provided;
    MPI_Query_thread(&provided);
  } else {
    // MPI environment has not been created, using manager to initialize.
    ctx_manager.EnvInitialize(required);
    should_finalize = true;
  }

  if (!ranks.empty()) {
    MPI_Group world_group;
    MPI_Comm_group(MPI_COMM_WORLD, &world_group);
    MPI_Group work_group;
    MPI_Group_incl(world_group, ranks.size(), ranks.data(), &work_group);
    MPI_Comm_create_group(MPI_COMM_WORLD, work_group, 0, &(mpi_comm));
    if (mpi_comm == MPI_COMM_NULL) {
      mpi_comm = MPI_COMM_WORLD;
    }
    MPI_Group_free(&world_group);
    MPI_Group_free(&work_group);
  } else if (!mpi_comm) {
    // No ranks were given and no communicator provided to horovod_init() so use
    // MPI_COMM_WORLD
    MPI_Comm_dup(MPI_COMM_WORLD, &mpi_comm);
  }

  // Create local comm, Determine local rank by querying the local communicator.
  MPI_Comm_split_type(mpi_comm, MPI_COMM_TYPE_SHARED, 0, MPI_INFO_NULL,
                      &local_comm);

  // Get local rank and world rank for cross comm establishment.
  int local_rank, world_rank;
  MPI_Comm_rank(mpi_comm, &world_rank);
  MPI_Comm_rank(local_comm, &local_rank);

  // Create cross node communicator.
  MPI_Comm_split(mpi_comm, local_rank, world_rank, &cross_comm);

  // Create custom MPI float16 data type.
```

```
  MPI_Type_contiguous(2, MPI_BYTE, &mpi_float16_t);
  MPI_Type_commit(&mpi_float16_t);

  // Create custom MPI float16 summation op.
  MPI_Op_create(&float16_sum, 1, &mpi_float16_sum);
}
```

0x05 业务逻辑

我们具体看看业务逻辑。

5.1 RunLoopOnce 总体业务

RunLoopOnce 负责总体业务逻辑，其功能如下：

- 计算是否还需要sleep，即检查从上一个cycle开始到现在，是否已经超过一个cycle时间；
- 利用 ComputeResponseList 来让 rank 0 与 worker 协调，获取 Request，计算 response；

  rank 0 会 遍历 response_list，对于 response 逐一执行操作。

  response_list 是 rank 0 处理，response cache 是其他 rank 处理。
- 利用 PerformOperation 对于每个response，做collective的操作
- 如果需要 auto tune，就同步参数；

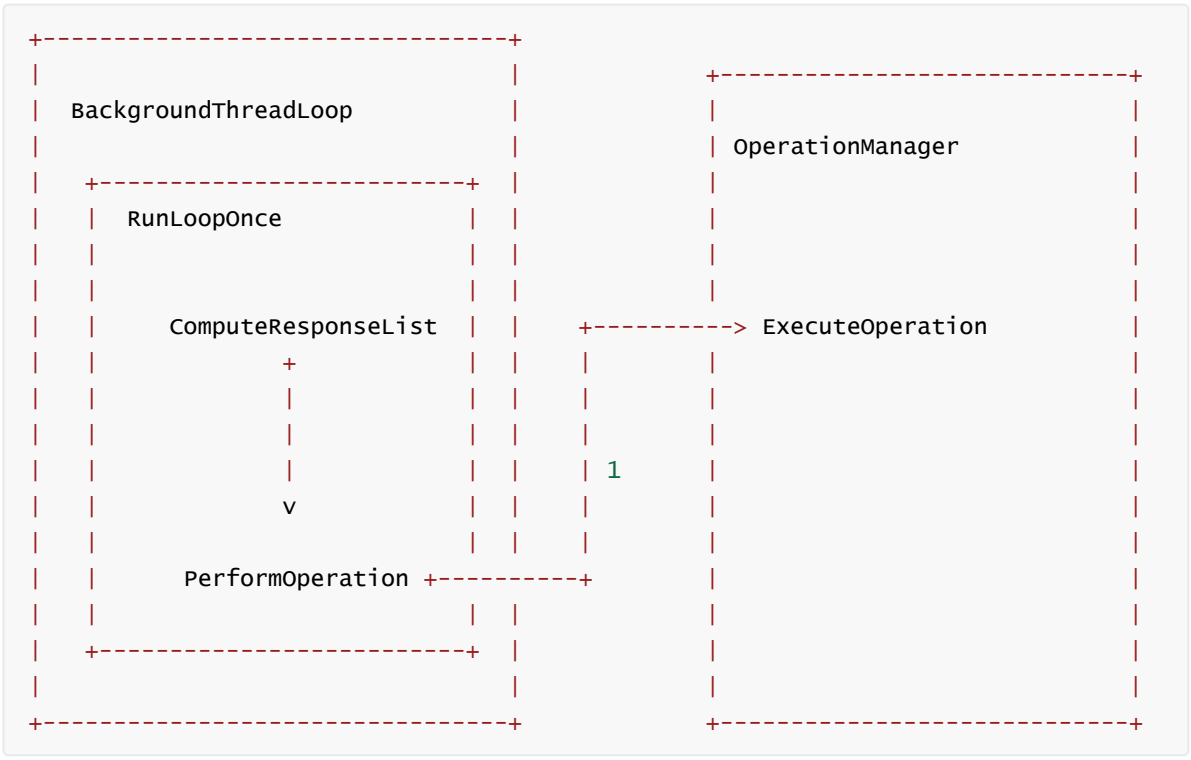
我们可以看到Horovod的工作流程大致如之前所说的，是一个生产者和消费者的模式。controller在这里是做协调的工作：会互通各个 rank 有哪些 request 已经就绪，对于就绪的 request，执行collective的操作。

缩减版代码如下：

```
bool RunLoopOnce(HorovodGlobalState& state) {
  // This delay determines thread frequency and communication message latency
  .....

  // 让 rank 0 与 worker 协调，获取 Request，计算 response
  auto response_list =
      state.controller->ComputeResponseList(horovod_global.shut_down, state);

  // Get tensor name and size data for autotuning.
  .....

  // Perform the collective operation. All nodes should end up performing
  // the same operation.
  // 对于每个response，做collective的操作
  int rank = state.controller->GetRank();
  for (auto& response : response_list.responses()) {
    PerformOperation(response, horovod_global);
  }

  // 如果需要 auto tune，就同步参数
  if (state.parameter_manager.IsAutoTuning()) {
    bool should_sync =
        state.parameter_manager.Update(tensor_names, total_tensor_size);

    if (should_sync) {
      state.controller->SynchronizeParameters();
    }
```

```
  }

  return !response_list.shutdown();
}
```

流程如下：



5.2 ComputeResponseList 计算 response

在后台线程里，最重要的一个函数调用是 `ComputeResponseList`。ComputeResponseList 实现了协调过程，即来让 rank 0 与 worker 协调，获取 Request，计算 response。

Horovod 也遵循着 Coordinator 的设计，与百度类似。无论是百度还是 Horovod 中的 Coordinator 都类似是 Actor 模式，主要起来协调多个进程工作的作用。在真正执行计算的时候，Horovod 同样引入了一个新的抽象 op_manager。从某种程度来说，我们可以把 controller 看做是对通信和协调管理能力的抽象，而 op_manager 是对实际计算的抽象。

5.2.1 总体思路

Controller::ComputeResponseList 的功能就是：worker 发送请求给 rank 0，然后coordinator 处理所有 worker 的请求，找到 ready 的，进行融合，最后结果发送给其他 rank：

- 利用 PopMessagesFromQueue 从 从自己进程的 GlobalState 的 Tensor Quene 中把目前的 Request 都取出来，进行处理，具体处理时使用了缓存，然后经过一系列处理缓存到 **message_queue_tmp** 中；
- 彼此同步cache信息，目的是得到每个worker 共同存储的 response列表；
- 判断是否需要进一步同步，比如是否response全都在cache之中；
- 如果不需要同步，则
  - 说明队列中所有消息都在缓存之中，不需要其他的协调。于是直接把缓存的response进行融合，放入response_list，下一轮时间片会继续处理；
- 如果需要同步，则
  - 如果是rank 0，

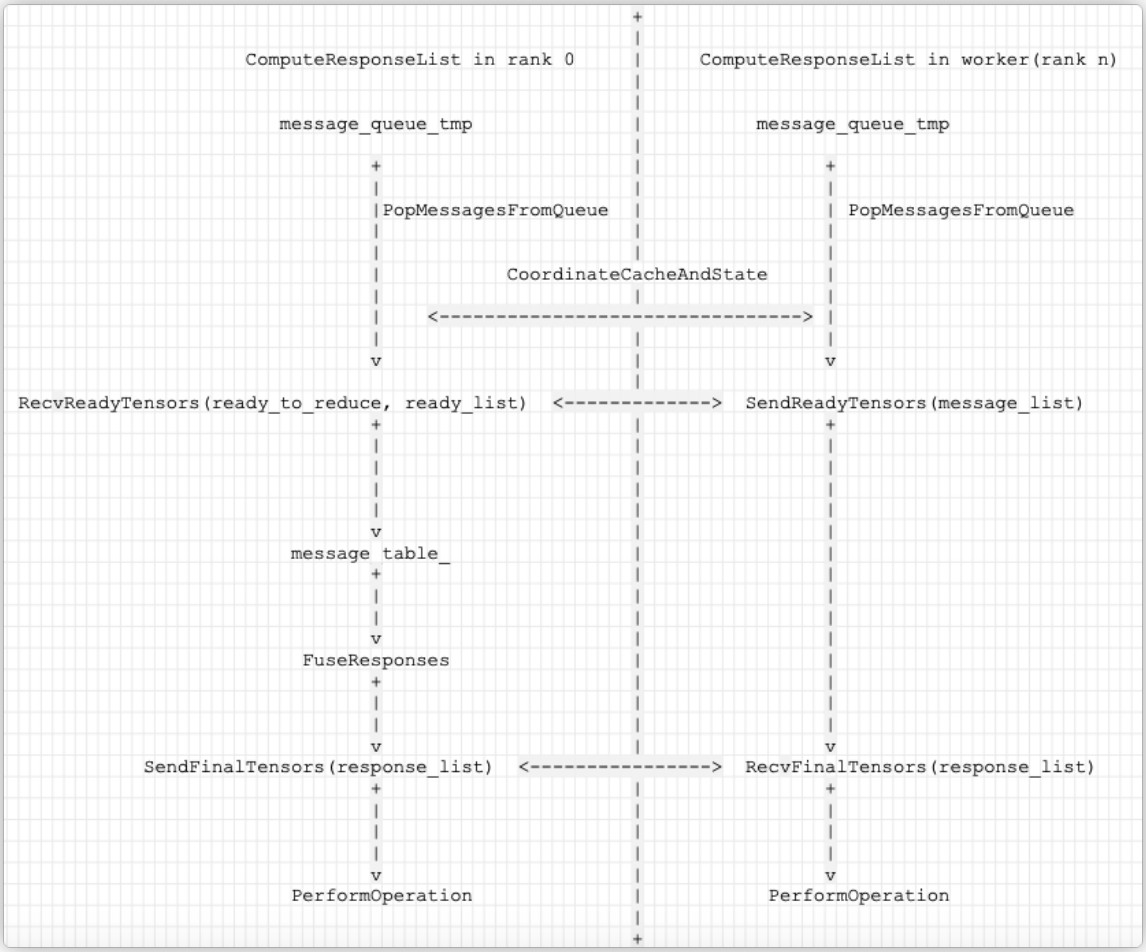
        - 因为rank 0 也会参与机器学习的训练，所以需要把rank 0的request也加入到message table之中。接受其他 rank 的 Request，把其他 rank 的 Request 加入到 message_table_ 之中。此处就同步阻塞了。
        - Rank 0 利用 RecvReadyTensors 接受其他 rank 的 Request，把其他 rank 的 Request 加入到 ready_to_reduce。此处就同步阻塞了。coordinator 会持续接收这些信息，直到获取的 Done 的数目等于 global_size。
        - 然后遍历 rank 0+1 ~ rank n，逐一处理每个 rank 的 response；
        - 最后，message table 之中已经有了所有的可以reduce的列表，responses 的来源是以下三部分：
            - 来源1，response_cache_ in rank 0；
            - 来源2，逐一处理 ready_to_reduce；
            - 来源3，join_response
        - 利用 FuseResponses 对tensor做fusion：即将一些tensor合并成一个大的tensor，再做collective的操作。
        - coordinator 会找到所有准备好 reduce 的 tensors，通过 **SendFinalTensors(response_list)** 返回一个 response 给所有的 worker，如果信息有误会返回一个 error，发送完成也会发送一个 Done。
    - 如果是其他 rank，则：
        - 当 worker 到达了前端 all_reduce 这句的时候，会用 **message_queue_tmp** 整理成一个 message_list通过 **SendReadyTensors** 函数往主节点(coordinator，Rank 0) 发送一个请求表明我打算reduce 的 Request，然后会把准备 reduce 的 tensor 信息通过 message_list 迭代地送过去，最后有一个 Done 的请求，然后同步阻塞。
        - Worker 利用 RecvFinalTensors(response_list) 监听 response 的信息，从 Rank 0 接受 ready response list，同步阻塞。当收到 Done，会尝试调用 performation 去进行 reduce 。
    - coordinator 和 worker 都会把同步的信息整理成一个 responses 的数组给到后面的 PerformOperation 操作。

这里说一下mpi是怎么实现的，就是 coordinator 和 对应的 worker 会阻塞地到同一条指令：

- SendReadyTensors 和 RecvReadyTensors 阻塞到 MPI_Gather；
- SendFinalTensors 和 RecvFinalTensors 到 MPI_Bcast；

可以这样分辨：如果是 coordinator 发送的就是 MPI_Bcast，如果是worker 发送的是 MPI_Gather。通信都是先同步需要通信message的大小 length，再同步message。

具体如下图：



5.2.2 详细分析

下面是比较详细的分析，参考了网上的资料，自己也做了解读。

```
ResponseList Controller::ComputeResponseList(std::atomic_bool& shut_down,
                                             HorovodGlobalState& state) {
  // Update cache capacity if autotuning is active.
  if (parameter_manager_.IsAutoTuning()) {
    response_cache_.set_capacity((int)parameter_manager_.CacheEnabled() *
                                 cache_capacity_);
  }

  // Copy the data structures out from parameters.
  // However, don't keep the lock for the rest of the loop, so that
  // enqueued stream callbacks can continue.

  CacheCoordinator cache_coordinator(response_cache_.num_active_bits());

  // 从 Tensor Quene 中把目前的 Request 都取出来，进行处理
  // message queue used only in this cycle
  std::deque<Request> message_queue_tmp;
  tensor_queue_.PopMessagesFromQueue(message_queue_tmp);
  for (auto& message : message_queue_tmp) {
    if (message.request_type() == Request::JOIN) {
      state.joined = true;
      // set_uncached_in_queue 记录没有cache的
      cache_coordinator.set_uncached_in_queue(true);
      continue;
    }
```

```
    // 这里使用了缓存，就是为了缓存本rank已经得到了多少response。
    // Keep track of cache hits
    if (response_cache_.capacity() > 0) {
      // 需要看看这个tensor是否已经得到了对应的response。为啥要缓存呢？不是都 ready 之后，就立刻进行 all reduce 了嘛。
      // cached 函数比较复杂，不但要看是否已经缓存，还要看新 tensor 是否和已经缓存的同名 tensor 的各种参数一致，比如device，dtype，shape等等。如果不一致，则标识缓存的是 INVALID。难道深度学习训练中，这些会变更？
      auto cache_ = response_cache_.cached(message);
      if (cache_ == ResponseCache::CacheState::HIT) {
        uint32_t cache_bit = response_cache_.peek_cache_bit(message);
        cache_coordinator.record_hit(cache_bit);

        // Record initial time cached tensor is encountered in queue.
        stall_inspector_.RecordCachedTensorStart(message.tensor_name());

      } else {
        // 如果没有缓存
        if (cache_ == ResponseCache::CacheState::INVALID) {
          // 处理无效缓存记录
          uint32_t cache_bit = response_cache_.peek_cache_bit(message);
          cache_coordinator.record_invalid_bit(cache_bit);
        }
        // 如果没有缓存，则添加到 set_uncached_in_queue
        cache_coordinator.set_uncached_in_queue(true);

        // 从stall 移除
        // Remove timing entry if uncached or marked invalid.
        stall_inspector_.RemoveCachedTensor(message.tensor_name());
      }
    }
  }

  if (state.joined && response_cache_.capacity() > 0) {
    for (uint32_t bit : response_cache_.list_all_bits()) {
      cache_coordinator.record_hit(bit);
    }
  }

  // Flag indicating that the background thread should shut down.
  bool should_shut_down = shut_down;

  // 处理 stalled
  // Check for stalled tensors.
  if (stall_inspector_.ShouldPerformCheck()) {
    if (is_coordinator_) {
      should_shut_down |= stall_inspector_.CheckForStalledTensors(size_);
    }

    if (response_cache_.capacity() > 0) {
      stall_inspector_.InvalidateStalledCachedTensors(cache_coordinator);
    }
    stall_inspector_.UpdateCheckTime();
  }

  cache_coordinator.set_should_shut_down(should_shut_down);
```

```
  if (response_cache_.capacity() > 0) {
    // 为什么要彼此同步cache信息？
    // Obtain common cache hits and cache invalidations across workers. Also,
    // determine if any worker has uncached messages in queue or requests
    // a shutdown. This function removes any invalid cache entries, if they
    // exist.
    // 这里会同步，也会从 response_cache_ 之中移除 invalid 的。
    // 目的是得到每个worker 共同存储的 response列表
    CoordinateCacheAndState(cache_coordinator);

    // Remove uncommon cached tensors from queue and replace to state
    // queue for next cycle. Skip adding common cached tensors to
    // queue as they are handled separately.

    // 此时 cache_coordinator 已经是所有worker 共有的response 列表了。需要移除那些 不在共有response 列表中的 response。
    // 为什么有的worker会没有某种response?
    // 会从 tensor request messages 之中看看是否已经有cache的了，然后相应更新 tensor_queue_。
    std::deque<Request> messages_to_replace;
    size_t num_messages = message_queue_tmp.size();
    for (size_t i = 0; i < num_messages; ++i) {
      auto& message = message_queue_tmp.front();
      if (response_cache_.cached(message) == ResponseCache::CacheState::HIT) {
        uint32_t cache_bit = response_cache_.peek_cache_bit(message);
        if (cache_coordinator.cache_hits().find(cache_bit) ==
            cache_coordinator.cache_hits().end()) {
          // Try to process again in next cycle.
          messages_to_replace.push_back(std::move(message));
        } else {
          // Remove timing entry for messages being handled this cycle.
          stall_inspector_.RemoveCachedTensor(message.tensor_name());
        }
      } else {
        // Remove timing entry for messages being handled this cycle.
        stall_inspector_.RemoveCachedTensor(message.tensor_name());
        message_queue_tmp.push_back(std::move(message));
      }
      message_queue_tmp.pop_front();
    }
    tensor_queue_.PushMessagesToQueue(messages_to_replace);
  }
  // End of response_cache_.capacity()


  ResponseList response_list;
  response_list.set_shutdown(cache_coordinator.should_shut_down());

  bool need_communication = true;
  // 判断是否需要进一步同步，比如response全都在cache之中。
  if (response_cache_.capacity() > 0 &&
      !cache_coordinator.uncached_in_queue()) {
    // if cache is enabled and no uncached new message coming in, no need for
    // additional communications
    need_communication = false;

    // If no messages to send, we can simply return an empty response list;
    if (cache_coordinator.cache_hits().empty()) {
```

```
      return response_list;
    }
    // otherwise we need to add cached messages to response list.
  }

  if (!need_communication) {
    // 队列中所有消息都在缓存之中，不需要其他的协调。于是直接把缓存的response进行融合，放入
response_list
    // If all messages in queue have responses in cache, use fast path with
    // no additional coordination.

    // If group fusion is disabled, fuse tensors in groups separately
    if (state.disable_group_fusion && !group_table_.empty()) {
      // Note: need group order to be based on position in cache for global
consistency
      std::vector<int> common_ready_groups;
      std::unordered_set<int> processed;
      for (auto bit : cache_coordinator.cache_hits()) {
        const auto& tensor_name =
response_cache_.peek_response(bit).tensor_names()[0];
        int group_id = group_table_.GetGroupIDFromTensorName(tensor_name);
        if (group_id != NULL_GROUP_ID && processed.find(group_id) ==
processed.end()) {
          common_ready_groups.push_back(group_id);
          processed.insert(group_id);
        }
      }

      for (auto id : common_ready_groups) {
        std::deque<Response> responses;
        for (const auto &tensor_name : group_table_.GetGroupTensorNames(id)) {
          auto bit = response_cache_.peek_cache_bit(tensor_name);
          responses.push_back(response_cache_.get_response(bit));
          // Erase cache hit to avoid processing a second time.
          cache_coordinator.erase_hit(bit);
        }

        FuseResponses(responses, state, response_list);
      }
    }

    std::deque<Response> responses;
    // Convert cache hits to responses. Populate so that least
    // recently used responses get priority. All workers call the code
    // here so we use the get method here to consistently update the cache
    // order.
    for (auto bit : cache_coordinator.cache_hits()) {
      responses.push_back(response_cache_.get_response(bit));
    }

    // Fuse responses as normal.
    FuseResponses(responses, state, response_list);
    response_list.set_shutdown(cache_coordinator.should_shut_down());
  } else {
    // 有没有缓存的消息进入，需要找出来这些是不是可以reduce的。
    // There are uncached messages coming in, need communication to figure out
    // whether those are ready to be reduced.
```

```
    // Collect all tensors that are ready to be reduced. Record them in the
    // tensor count table (rank zero) or send them to rank zero to be
    // recorded (everyone else).
    std::vector<std::string> ready_to_reduce;

    if (is_coordinator_) {
      // 我是 rank 0，对于master进程，记录已经ready的tensor。
      // rank 0 也会参与机器学习的训练，所以需要把rank 0的request也加入到message table之
中。
      while (!message_queue_tmp.empty()) { // 注意此时message_queue_tmp中的request
是来自master进程
        // Pop the first available message
        Request message = message_queue_tmp.front();
        message_queue_tmp.pop_front();

        if (message.request_type() == Request::JOIN) {
          state.joined_size++;
          continue;
        }

        bool reduce = IncrementTensorCount(message, state.joined_size);
        stall_inspector_.RecordUncachedTensorStart(
            message.tensor_name(), message.request_rank(), size_);
        if (reduce) {
          ready_to_reduce.push_back(message.tensor_name());
        }
      }

      // 接受其他 rank 的 Request，把其他 rank 的 ready Request 加入到 message_table_
之中。
      // 此处就同步阻塞了
      // Receive ready tensors from other ranks
      std::vector<RequestList> ready_list;
      RecvReadyTensors(ready_to_reduce, ready_list);

      // 处理所有 rank 的 Request。
      // Process messages.
      // 遍历 rank 0+1 ~ rank n，逐一处理每个 rank 的 response
      for (int i = 1; i < size_; ++i) { // size_是指有多少个rank

        // 每一个 rank 的 response list。
        auto received_message_list = ready_list[i];
        for (auto& received_message : received_message_list.requests()) {
          auto& received_name = received_message.tensor_name();

          // Join类型消息是指有新的rank加入，Horovod支持弹性
          if (received_message.request_type() == Request::JOIN) {
            state.joined_size++;  // 增加该tensor已经ready的rank的个数，如果所有rank都
ready，则发给其他rank
            continue;
          }

          bool reduce = IncrementTensorCount(received_message,
state.joined_size);
          stall_inspector_.RecordUncachedTensorStart(
              received_message.tensor_name(), received_message.request_rank(),
              size_);
```

```
          // 如果已经达到了最大数值，则可以 reduce 了，加入到 ready_to_reduce。
          if (reduce) {
            ready_to_reduce.push_back(received_name);
          }
        }
        if (received_message_list.shutdown()) {
          // Received SHUTDOWN request from one of the workers.
          should_shut_down = true;
        }
      }

      // Check if tensors from previous ticks are ready to reduce after Joins.
      // 遍历 message_table_，目的是看看上一轮处理的 response 在本轮是否可以 reduce
      if (state.joined_size > 0) {
        for (auto& table_iter : message_table_) {
          int count = (int)table_iter.second.size();
          if (count == (size_ - state.joined_size) &&
              std::find(ready_to_reduce.begin(), ready_to_reduce.end(),
                        table_iter.first) == ready_to_reduce.end()) {
            state.timeline.NegotiateEnd(table_iter.first);
            ready_to_reduce.push_back(table_iter.first);
          }
        }
      }

      // Fuse tensors in groups before processing others.
      if (state.disable_group_fusion && !group_table_.empty()) {

        // Extract set of common groups from coordinator tensor list and cache
hits.
        std::vector<int> common_ready_groups;
        std::unordered_set<int> processed;

        for (const auto& tensor_name : ready_to_reduce) {
          int group_id = group_table_.GetGroupIDFromTensorName(tensor_name);
          if (group_id != NULL_GROUP_ID && processed.find(group_id) ==
processed.end()) {
            common_ready_groups.push_back(group_id);
            processed.insert(group_id);
            // Leaving name in list, to be skipped later.
          }
        }

        if (response_cache_.capacity() > 0) {
          for (auto bit : cache_coordinator.cache_hits()) {
            const auto& tensor_name =
response_cache_.peek_response(bit).tensor_names()[0];
            int group_id = group_table_.GetGroupIDFromTensorName(tensor_name);
            if (group_id != NULL_GROUP_ID && processed.find(group_id) ==
processed.end()) {
              common_ready_groups.push_back(group_id);
              processed.insert(group_id);
            }
          }
        }

        // For each ready group, form and fuse response lists independently
        for (auto id : common_ready_groups) {
```

```
          std::deque<Response> responses;
          for (const auto &tensor_name : group_table_.GetGroupTensorNames(id)) {
            if (message_table_.find(tensor_name) != message_table_.end()) {
              // Uncached message
              Response response = ConstructResponse(tensor_name,
state.joined_size);
              responses.push_back(std::move(response));

            } else {
              // Cached message
              auto bit = response_cache_.peek_cache_bit(tensor_name);
              responses.push_back(response_cache_.get_response(bit));
              // Erase cache hit to avoid processing a second time.
              cache_coordinator.erase_hit(bit);
            }
          }

          FuseResponses(responses, state, response_list);
        }
      }

      // 此时，message table 之中已经有了所有的可以reduce的列表

      // At this point, rank zero should have a fully updated tensor count
      // table and should know all the tensors that need to be reduced or
      // gathered, and everyone else should have sent all their information
      // to rank zero. We can now do reductions and gathers; rank zero will
      // choose which ones and in what order, and will notify the other ranks
      // before doing each reduction.
      std::deque<Response> responses;

      // responses 的来源是以下三部分

      // 来源1，response_cache_ in rank 0
      if (response_cache_.capacity() > 0) {
        // Prepopulate response list with cached responses. Populate so that
        // least recently used responses get priority. Since only the
        // coordinator rank calls this code, use peek instead of get here to
        // preserve cache order across workers.
        // No need to do this when all ranks did Join.
        if (state.joined_size < size_) {
          for (auto bit : cache_coordinator.cache_hits()) {
            responses.push_back(response_cache_.peek_response(bit));
          }
        }
      }

      // 来源2，逐一处理 ready_to_reduce
      for (auto& tensor_name : ready_to_reduce) {
        // Skip tensors in group that were handled earlier.
        if (state.disable_group_fusion &&
            !group_table_.empty() &&
            group_table_.GetGroupIDFromTensorName(tensor_name) != NULL_GROUP_ID)
{
          continue;
        }

        Response response = ConstructResponse(tensor_name, state.joined_size);
```

```
        responses.push_back(std::move(response));
      }

      // 来源3，join_response
      if (state.joined_size == size_) {
        // All ranks did Join(). Send the response, reset joined size.
        Response join_response;
        join_response.set_response_type(Response::JOIN);
        join_response.add_tensor_name(JOIN_TENSOR_NAME);
        responses.push_back(std::move(join_response));
        state.joined_size = 0;
      }

      // 进行融合
      FuseResponses(responses, state, response_list);
      response_list.set_shutdown(should_shut_down);

      // Broadcast final results to other ranks.
      SendFinalTensors(response_list);

    } else {
      // 我是其他的 rank，非master，则发送自己已经ready的tensor给master，再接收已经ready
的tensor列表
      RequestList message_list;
      message_list.set_shutdown(should_shut_down);
      while (!message_queue_tmp.empty()) {
        message_list.add_request(message_queue_tmp.front());
        message_queue_tmp.pop_front();
      }

      // 给 Rank 0 发送 Request，同步阻塞
      // Send ready tensors to rank zero
      SendReadyTensors(message_list);

      // 从 Rank 0 接受 ready response list，同步阻塞
      // Receive final tensors to be processed from rank zero
      RecvFinalTensors(response_list);
    }
  }

  if (!response_list.responses().empty()) {
    std::string tensors_ready;
    for (const auto& r : response_list.responses()) {
      tensors_ready += r.tensor_names_string() + "; ";
    }
  }

  // If need_communication is false, meaning no uncached message coming in,
  // thus no need to update cache.
  if (need_communication && response_cache_.capacity() > 0) {
    // All workers add supported responses to cache. This updates the cache
    // order consistently across workers.
    for (auto& response : response_list.responses()) {
      if ((response.response_type() == Response::ResponseType::ALLREDUCE ||
           response.response_type() == Response::ResponseType::ADASUM ||
           response.response_type() == Response::ResponseType::ALLTOALL) &&
          (int)response.devices().size() == size_) {
        response_cache_.put(response, tensor_queue_, state.joined);
```

```
      }
    }
  }

  // Reassign cache bits based on current cache order.
  response_cache_.update_cache_bits();

  return response_list;
}
```

我们接下来重点看几个函数。

5.2.3 IncrementTensorCount

IncrementTensorCount 的作用是计算是否所有的 tensor 都已经准备好。

如果 `bool ready_to_reduce = count == (size_ - joined_size) ,` 就会知道这个是可以 allreduce 的。

```
bool Controller::IncrementTensorCount(const Request& msg, int joined_size) {
  auto& name = msg.tensor_name();
  auto table_iter = message_table_.find(name);
  if (table_iter == message_table_.end()) {
    std::vector<Request> messages = {msg};
    messages.reserve(static_cast<unsigned long>(size_));
    message_table_.emplace(name, std::move(messages));
    table_iter = message_table_.find(name);
  } else {
    std::vector<Request>& messages = table_iter->second;
    messages.push_back(msg);
  }

  std::vector<Request>& messages = table_iter->second;
  int count = (int)messages.size();
  bool ready_to_reduce = count == (size_ - joined_size); // 判断是否可以 allreduce

  return ready_to_reduce;
}
```

具体调用 就是 rank 0 来负责，看看是不是 allreduce了。

即 如果 IncrementTensorCount 了，就说明齐全了，可以把 Request 加入到 message_table_ 之中。

```
    if (is_coordinator_) {

      while (!message_queue_tmp.empty()) {
        // Pop the first available message
        Request message = message_queue_tmp.front();
        message_queue_tmp.pop_front();

        if (message.request_type() == Request::JOIN) {
          state.joined_size++;
          continue;
        }

        // 这里调用
        bool reduce = IncrementTensorCount(message, state.joined_size);
```

```
        stall_inspector_.RecordUncachedTensorStart(
            message.tensor_name(), message.request_rank(), size_);
        if (reduce) {
          ready_to_reduce.push_back(message.tensor_name());
        }
      }
```

5.2.4 RecvReadyTensors

该函数的作用是收集其他 rank 的 Request。

- 使用 MPI_Gather 确定消息长度；
- 使用 MPI_Gatherv 收集消息；
- 因为 rank 0 已经被处理了，所以这里不处理 rank 0；

```
void MPIController::RecvReadyTensors(std::vector<std::string>& ready_to_reduce,
                                     std::vector<RequestList>& ready_list) {
  // Rank zero has put all its own tensors in the tensor count table.
  // Now, it should count all the tensors that are coming from other
  // ranks at this tick.

  // 1. Get message lengths from every rank.
  auto recvcounts = new int[size_];
  recvcounts[0] = 0;
  MPI_Gather(MPI_IN_PLACE, 1, MPI_INT, recvcounts, 1, MPI_INT, RANK_ZERO,
             mpi_ctx_.mpi_comm);

  // 2. Compute displacements.
  auto displcmnts = new int[size_];
  size_t total_size = 0;
  for (int i = 0; i < size_; ++i) {
    if (i == 0) {
      displcmnts[i] = 0;
    } else {
      displcmnts[i] = recvcounts[i - 1] + displcmnts[i - 1];
    }
    total_size += recvcounts[i];
  }

  // 3. Collect messages from every rank.
  auto buffer = new uint8_t[total_size];
  MPI_Gatherv(nullptr, 0, MPI_BYTE, buffer, recvcounts, displcmnts, MPI_BYTE,
              RANK_ZERO, mpi_ctx_.mpi_comm);

  // 4. Process messages.
  // create a dummy list for rank 0
  ready_list.emplace_back();
  for (int i = 1; i < size_; ++i) {
    auto rank_buffer_ptr = buffer + displcmnts[i];
    RequestList received_message_list;
    RequestList::ParseFromBytes(received_message_list, rank_buffer_ptr);
    ready_list.push_back(std::move(received_message_list));
  }

  // 5. Free buffers.
  delete[] recvcounts;
  delete[] displcmnts;
```

```
  delete[] buffer;
}
```

5.2.5 SendReadyTensors

该函数是 其他 rank 同步 Request 给 rank 0。

- 使用 MPI_Gather 确定消息长度；
- 使用 MPI_Gatherv 收集消息；

```
void MPIController::SendReadyTensors(RequestList& message_list) {
  std::string encoded_message;
  RequestList::SerializeToString(message_list, encoded_message);
  int encoded_message_length = (int)encoded_message.length() + 1;
  int ret_code = MPI_Gather(&encoded_message_length, 1, MPI_INT, nullptr, 1,
                            MPI_INT, RANK_ZERO, mpi_ctx_.mpi_comm);

  ret_code = MPI_Gatherv((void*)encoded_message.c_str(), encoded_message_length,
                         MPI_BYTE, nullptr, nullptr, nullptr, MPI_BYTE,
                         RANK_ZERO, mpi_ctx_.mpi_comm);
}
```

5.2.6 SendFinalTensors

该函数作用是 rank 0 把最后结果发送给其他 rank；

```
void MPIController::SendFinalTensors(ResponseList& response_list) {
  // Notify all nodes which tensors we'd like to reduce at this step.
  std::string encoded_response;
  ResponseList::SerializeToString(response_list, encoded_response);
  int encoded_response_length = (int)encoded_response.length() + 1;
  MPI_Bcast(&encoded_response_length, 1, MPI_INT, RANK_ZERO, mpi_ctx_.mpi_comm);

  MPI_Bcast((void*)encoded_response.c_str(), encoded_response_length, MPI_BYTE,
            RANK_ZERO, mpi_ctx_.mpi_comm);
}
```

5.2.7 RecvFinalTensors

该函数作用是 worker 从 Rank 0 接受 ready response list，同步阻塞

```
void MPIController::RecvFinalTensors(ResponseList& response_list) {
  int msg_length;
  int ret_code =
      MPI_Bcast(&msg_length, 1, MPI_INT, RANK_ZERO, mpi_ctx_.mpi_comm);

  auto buffer = new uint8_t[msg_length];
  ret_code =
      MPI_Bcast(buffer, msg_length, MPI_BYTE, RANK_ZERO, mpi_ctx_.mpi_comm);

  ResponseList::ParseFromBytes(response_list, buffer);
  delete[] buffer;
}
```

5.3 根据 response 执行操作

我们接下来要看看另一个重要操作 PerformOperation，就是根据 response 执行操作。

其调用顺序是：

- BackgroundThreadLoop 调用 RunLoopOnce；
- RunLoopOnce 如果是 rank 0， 则处理 response_list，然后调用 PerformOperation；
- PerformOperation 进而 调用 op_manager -> ExecuteOperation------ ExecuteAllreduce；

我们可以看到，ComputeResponseList 返回了 response_list，就是这些 response 对应的 tensor 可以做 allreduce了。然后会遍历每一个 response，进行 PerformOperation。

```
auto response_list =
  state.controller->ComputeResponseList(horovod_global.shut_down, state);

 int rank = state.controller->GetRank();
 for (auto& response : response_list.responses()) {
   PerformOperation(response, horovod_global);
 }
```

5.3.1 PerformOperation

从 ComputeResponseList 继续跑 RunLoopOnce， worker node 会根据前面 ComputeResponseList 返回的 response_list 对每个 response 轮询调用 PerformOperation 完成对应的 reduce 工作。

主要调用 status = op_manager->ExecuteOperation(entries, response); 具体如下：

- PerformOperation 会从 horovod_global.tensor_queue 通过函数 GetTensorEntriesFromResponse 取出对应的 TensorEntry；
- 如果还没初始化buffer，调用 horovod_global.fusion_buffer.InitializeBuffer 初始化；
- 然后 status = op_manager->ExecuteOperation(entries, response) 会调用不同的 op->Execute(entries, response) 执行reduce 运算；
- 然后调用不同 entries 的 callback，这里 callback 一般是前端作相应的操作；

```
// Process a Response by doing a reduction, a gather, a broadcast, or
// raising an error.
void PerformOperation(Response response, HorovodGlobalState& state) {
  std::vector<TensorTableEntry> entries;
  auto& timeline = horovod_global.timeline;
  if (response.response_type() != Response::JOIN) {
    horovod_global.tensor_queue.GetTensorEntriesFromResponse(response, entries,
                                                             state.joined);

    if (entries.size() > 1) { // 如果多于1个，则可以进行fuse，以提高throughput
      auto first_entry = entries[0];
      Status status = horovod_global.fusion_buffer.InitializeBuffer(
          horovod_global.controller->TensorFusionThresholdBytes(),
          first_entry.device, first_entry.context,
          horovod_global.current_nccl_stream,
          [&]() { timeline.ActivityStartAll(entries, INIT_FUSION_BUFFER); },
          [&]() { timeline.ActivityEndAll(entries); });
      if (!status.ok()) {
        for (auto& e : entries) {
          timeline.End(e.tensor_name, nullptr);
          // Callback can be null if the rank sent Join request.
          if (e.callback != nullptr) {
```

```
            e.callback(status);
          }
        }
        return;
      }
    }

    // On GPU data readiness is signalled by ready_event.
    // 即使tensor可以进行操作了，但需要等待数据同步到显存
    std::vector<TensorTableEntry> waiting_tensors;
    for (auto& e : entries) {
      if (e.ready_event != nullptr) {
        timeline.ActivityStart(e.tensor_name, WAIT_FOR_DATA);
        waiting_tensors.push_back(e);
      }
    }
    while (!waiting_tensors.empty()) {
      for (auto it = waiting_tensors.begin(); it != waiting_tensors.end();) {
        if (it->ready_event->Ready()) {
          timeline.ActivityEnd(it->tensor_name);
          timeline.ActivityStart(it->tensor_name, WAIT_FOR_OTHER_TENSOR_DATA);
          it = waiting_tensors.erase(it);
        } else {
          ++it;
        }
      }
      std::this_thread::sleep_for(std::chrono::nanoseconds(100));
    }
  }

  Status status;
  try {
    // 进行collective的操作
    status = op_manager->ExecuteOperation(entries, response);
  } catch (const std::exception& ex) {
    status = Status::UnknownError(ex.what());
  }

  ... // 调用 callback 函数
}
```

5.3.2 ExecuteOperation

然后 status = op_manager->ExecuteOperation(entries, response) 会调用不同的 op->Execute(entries, response) 执行reduce 运算。

这里来到了 OperationManager。

```
Status OperationManager::ExecuteOperation(std::vector<TensorTableEntry>&
entries,
                                          const Response& response) const {
  if (response.response_type() == Response::ALLREDUCE) {
    return ExecuteAllreduce(entries, response);
  } else if (response.response_type() == Response::ALLGATHER) {
    return ExecuteAllgather(entries, response);
  } else if (response.response_type() == Response::BROADCAST) {
    return ExecuteBroadcast(entries, response);
```

```
  } else if (response.response_type() == Response::ALLTOALL) {
    return ExecuteAlltoall(entries, response);
  } else if (response.response_type() == Response::JOIN) {
    return ExecuteJoin(entries, response);
  } else if (response.response_type() == Response::ADASUM) {
    return ExecuteAdasum(entries, response);
  } else if (response.response_type() == Response::ERROR) {
    return ExecuteError(entries, response);
  } else {
    throw std::logic_error("No operation found for response type provided");
  }
}
```

5.3.3 ExecuteAllreduce

op->Execute(entries, response); 就是调用了类似 MPIAllreduce . Execute。

```
Status OperationManager::ExecuteAllreduce(std::vector<TensorTableEntry>&
entries,
                                          const Response& response) const {
  for (auto& op : allreduce_ops_) {
    if (op->Enabled(*param_manager_, entries, response)) {
      return op->Execute(entries, response);
    }
  }
}
```

allreduce_ops_ 是从哪里来的？在 OperationManager 构建函数中有。

```
allreduce_ops_(std::move(allreduce_ops)),
```

所以我们看看allreduce_ops。

5.3.4 allreduce_ops

在 CreateOperationManager 之中对 allreduce_ops 进行添加。

可以看到，添加的类型大致如下：

- MPI_GPUAllreduce
- NCCLHierarchicalAllreduce
- NCCLAllreduce
- DDLAllreduce
- GlooAllreduce
- CCLAllreduce
- MPIAllreduce
- ……

```
OperationManager* CreateOperationManager(HorovodGlobalState& state) {
  // Order of these operations is very important. Operations will be checked
  // sequentially from the first to the last. The first 'Enabled' operation will
  // be executed.
  std::vector<std::shared_ptr<AllreduceOp>> allreduce_ops;
  std::vector<std::shared_ptr<AllgatherOp>> allgather_ops;
  std::vector<std::shared_ptr<BroadcastOp>> broadcast_ops;
  std::vector<std::shared_ptr<AllreduceOp>> adasum_ops;
```

```
  std::vector<std::shared_ptr<AlltoallOp>> alltoall_ops;

#if HAVE_MPI && HAVE_GPU // 如果构建了 MPI，就添加对应MPI_GPUAllreduce
  if (mpi_context.IsEnabled()) {
#if HOROVOD_GPU_ALLREDUCE == 'M'
    allreduce_ops.push_back(std::shared_ptr<AllreduceOp>(
        new MPI_GPUAllreduce(&mpi_context, &gpu_context, &state)));

#elif HAVE_NCCL && HOROVOD_GPU_ALLREDUCE == 'N' // 如果编译了NCCL，就添加
AdasumGpuAllreduceOp
    adasum_ops.push_back(std::shared_ptr<AllreduceOp>(new
AdasumGpuAllreduceOp(&mpi_context, &nccl_context, &gpu_context, &state)));

    allreduce_ops.push_back(
        std::shared_ptr<AllreduceOp>(new NCCLHierarchicalAllreduce(
            &nccl_context, &mpi_context, &gpu_context, &state)));

#elif HAVE_DDL && HOROVOD_GPU_ALLREDUCE == 'D'// 如果编译了DDL，就添加DDLAllreduce
    allreduce_ops.push_back(std::shared_ptr<AllreduceOp>(
        new DDLAllreduce(&ddl_context, &gpu_context, &state)));
#endif

#if HAVE_NCCL && HOROVOD_GPU_ALLREDUCE == 'N'// 如果编译了NCCL，就添加NCCLAllreduce
  allreduce_ops.push_back(std::shared_ptr<AllreduceOp>(
      new NCCLAllreduce(&nccl_context, &gpu_context, &state)));
#endif
```

5.3.5 MPIAllreduce

因为 allreduce_ops 类型很多，所以我们以 MPIAllreduce 举例如下：

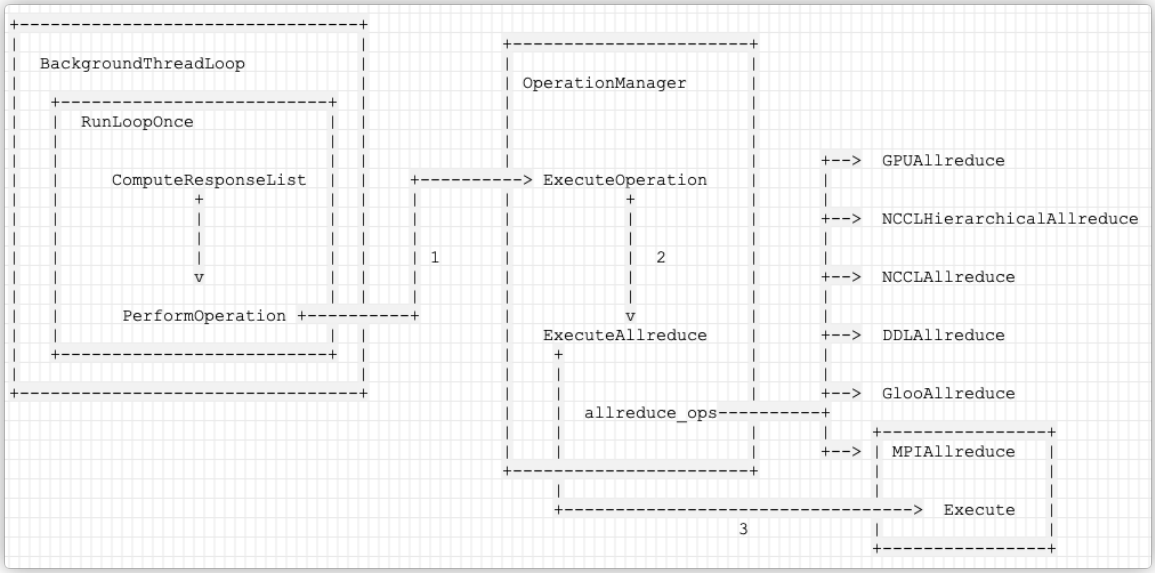
```
class MPIAllreduce : public AllreduceOp {
public:
  MPIAllreduce(MPIContext* mpi_context, HorovodGlobalState* global_state);

  virtual ~MPIAllreduce() = default;

  Status Execute(std::vector<TensorTableEntry>& entries, const Response&
response) override;

  bool Enabled(const ParameterManager& param_manager,
               const std::vector<TensorTableEntry>& entries,
               const Response& response) const override;

protected:
  MPIContext* mpi_context_;
};
```

`MPIAllreduce::Execute` 这里使用到了 MPI_Allreduce，也处理了 fusion，比如 MemcpyOutFusionBuffer。

```
#include "mpi_operations.h"

Status MPIAllreduce::Execute(std::vector<TensorTableEntry>& entries, const
Response& response) {
  auto& first_entry = entries[0];
```

```
  const void* fused_input_data;
  void* buffer_data;
  size_t buffer_len;
  int64_t num_elements = NumElements(entries);

  // Copy memory into the fusion buffer.
  auto& timeline = global_state_->timeline;
  if (entries.size() > 1) {
    timeline.ActivityStartAll(entries, MEMCPY_IN_FUSION_BUFFER);
    MemcpyInFusionBuffer(entries, fused_input_data, buffer_data, buffer_len);
    timeline.ActivityEndAll(entries);
  } else {
    fused_input_data = first_entry.tensor->data();
    buffer_data = (void*) first_entry.output->data();
    buffer_len = (size_t) first_entry.output->size();
  }

  if (response.prescale_factor() != 1.0) {
    // Execute prescaling op
    ScaleBuffer(response.prescale_factor(), entries, fused_input_data,
buffer_data, num_elements);
    fused_input_data = buffer_data; // for unfused, scale is done out of place
  }

  // Do allreduce.
  timeline.ActivityStartAll(entries, MPI_ALLREDUCE);
  const void* sendbuf = entries.size() > 1 || fused_input_data == buffer_data
                        ? MPI_IN_PLACE : fused_input_data;
  int op = MPI_Allreduce(sendbuf, buffer_data,
                         (int) num_elements,
                         mpi_context_->GetMPIDataType(first_entry.tensor),
                         mpi_context_->GetMPISumOp(first_entry.tensor->dtype()),
                         mpi_context_-
>GetMPICommunicator(Communicator::GLOBAL));
  timeline.ActivityEndAll(entries);

  if (response.postscale_factor() != 1.0) {
    // Execute postscaling op
    ScaleBuffer(response.postscale_factor(), entries, buffer_data, buffer_data,
num_elements);
  }

  // Copy memory out of the fusion buffer.
  if (entries.size() > 1) {
    timeline.ActivityStartAll(entries, MEMCPY_OUT_FUSION_BUFFER);
    MemcpyOutFusionBuffer(buffer_data, entries);
    timeline.ActivityEndAll(entries);
  }

  return Status::OK();
}
```

此时具体逻辑如下：



至此，后台线程架构基本理清，我们下一篇需要再返回去看看优化器如何实现。