horovod (7) --- DistributedOptimizer

0x00 摘要

我们需要一些问题或者说是设计要点来引导分析，而且因为读者可能没有看过本系列其他文章，因此问题点会和其他文章有部分重复：

- 第一个技术难点是：Horovod 如何从 TF 的执行流程中获取到 梯度（gradients）进行处理？
  - 在 TensorFlow 1.x 中，深度学习计算过程被表示成为一个计算图（graph），并且由 TensorFlow runtime 负责解释和执行，所以 Horovod 为了获得每个进程计算的梯度并且对于它们进行 AllReduce，就得用黑客的办法进入到 TF 图执行过程之中去获取梯度。
- 第二个技术难点是：Horovod 可以自己定义 AllReduce操作, 但是它的AllReduce操作怎么能够嵌入到 TF 的处理流程之中？
  - 因为 Horovod 自定义的这套HVD Operation 是跟TF OP 无关的，因此是无法直接插入到TF Graph之中进行执行，所以还需要有一个办法来把HVD OP注册到TF的OP之中。

0x01 背景概念

我们回忆一下背景概念。

1.1 深度学习框架

深度学习训练的核心问题是过反向梯度计算来拟合f()，反向梯度计算的目的是计算梯度和更新参数。而计算梯度的方式则主要是通过链式求导。一次链式求导只是一次的前向和后向的计算结果。模型训练的重点过程就是：前向传播和反向传播。

以简单的深度神经网络为例，为了完成对损失的优化，我们把数据分成batch，不断把数据送入模型网络中进行如下迭代过程，目的是使最终优化网络达到收敛：

- 一个batch的数据被送入网络进行前向传播，前向传播就是一系列的矩阵+激活函数等的组合运算。
- 前向传播输出的预测值会同真实值 label 进行对比之后，使用损失函数计算出此次迭代的损失；
- 把这个损失进行反向传播，送入神经网络模型中之前的每一层进行反向梯度计算，更新每一层的权值矩阵和bias；

深度学习框架帮助我们解决的核心问题之一就是反向传播时的梯度计算和更新。如果不用深度学习框架，就需要我们自己写方法以进行复杂的梯度计算和更新。

1.2 Tensorflow Optimizer

Tensorflow的底层结构是由张量组成的计算图。计算图就是底层的编程系统，每一个计算都是图中的一个节点，计算之间的依赖关系则用节点之间的边来表示。计算图构成了前向/反向传播的结构基础。

给定一个计算图, TensorFlow 使用自动微分 (反向传播) 来进行梯度运算。tf.train.Optimizer允许我们通过minimize()函数自动进行权值更新，此时tf.train.Optimizer.minimize()做了两件事：

- 计算梯度。即调用compute_gradients (loss, var_list ...) 计算loss对指定val_list的梯度，返回元组列表 `list(zip(grads, var_list))`。
- 用计算得到的梯度来更新对应权重。即调用 apply_gradients(grads_and_vars, global_step=global_step, name=None) 将 compute_gradients (loss, var_list ...) 的返回值作为输入对权重变量进行更新；

将minimize()分成两个步骤的原因是：可以在某种情况下对梯度进行修正，防止梯度消失或者梯度爆炸。

tensorflow也允许用户自己计算梯度，在用户做了中间处理之后，这个梯度会应用给权值进行更新，此时就会细分为以下三个步骤：

- 利用tf.train.Optimizer.compute_gradients计算梯度；
- 用户对梯度进行自定义处理。这里其实就是 Horovod 可以做手脚的地方；
- 对于用户计算后的梯度，利用tf.train.Optimizer.apply_gradients更新权值；

0x02 总体架构

2.1 总体思路

Horovod 作业的每个进程都调用单机版 TensorFlow 做本地计算，然后收集梯度，并且通过 AllReduce 来汇聚梯度并且更新每个进程中的模型。

Horovod 需要从 TensorFlow 截取梯度。

- TensorFlow 1.x
    - 在 TensorFlow 1.x 中，深度学习计算是一个计算图，由 TensorFlow 运行时负责解释执行。
    - Horovod 为了获得每个进程计算的梯度并且可以对它们进行 AllReduce，就必须潜入图执行的过程。为此，**Horovod 通过对用户Optimizer 进行封装组合方式完成了对梯度的 AllReduce 操作**，即， Horovod 要求开发者使用Horovod自己定义的

hvd.DistributedOptimizer 代替 TensorFlow 官方的 optimizer，从而可以在优化模型阶段得到梯度。

- TensorFlow 2.0
  - TensorFlow 2.0 的 eager execution模式 采用完全不同的计算方式。其前向计算过程把对基本计算单元（operator）的调用记录在一个数据结构 tape 里，随后进行反向计算过程时候可以回溯这个 tape，以此调用 operator 对应的 gradient operator。Tape 提供一个操作让用户可以获取每个参数的梯度。
  - Horovod 调用 TensorFlow 2.0 API 可以直接获取梯度。然后**Horovod 通过封装 tape 完成 AllReduce 调用。**

3.2 总体调用关系

我们先给出总体调用关系：hvd.DistributedOptimizer继承keras Optimizer，然后hvd.DistributedOptimizer在其重载的get_gradients中把获取到的梯度传给hvd.allreduce(gradients, ...)，从而实现整个horovod集群的梯度集体归并。

具体计算梯度的逻辑是：

- TF 调用 hvd.DistributedOptimizer 的 compute_gradients 方法：
  - hvd.DistributedOptimizer 首先会利用 TF 官方 optimizer.compute_gradients 计算出本地梯度；
  - 然后利用 AllReduce 来得到各个进程平均后的梯度；
  - compute_gradients 返回一个(梯度，权值)对的列表。由apply_gradients使用；
- TF 调用 hvd.DistributedOptimizer 的 apply_gradients 方法：
  - 调用 TF 官方 optimizer.apply_gradients 对传入的参数进行处理，返回一个更新权值的op。TF 可以用这个返回值进行后续处理；

因为 TF 的版本问题，所以我们区分 1.x, 2.x 来分析。

0x04 TensorFlow 1.x

前面提到了，Horovod 要求开发者使用Horovod自己定义的 hvd.DistributedOptimizer 代替 TensorFlow 官方的 optimizer，从而可以在优化模型阶段得到梯度，所以我们从_DistributedOptimizer进行分析。

4.1 _DistributedOptimizer

以 `horovod/tensorflow/__init__.py` 为例。

```
try:
    # TensorFlow 2.x
    _LegacyOptimizer = tf.compat.v1.train.Optimizer
except AttributeError:
    try:
        # TensorFlow 1.x
        _LegacyOptimizer = tf.train.Optimizer
    except AttributeError:
        # Future TensorFlow versions
        _LegacyOptimizer = None
```

可以看到，对于 TensorFlow 1.x，我们后续使用的基础是 `_LegacyOptimizer`。

`_DistributedOptimizer` 就继承了 `_LegacyOptimizer`。其封装了另外一个tf.optimizer，在模型应用梯度之前使用allreduce操作收集梯度值并求其均值。这个被封装的tf.optimizer就是用户在使用时候指定的TF官方优化器。

具体可以回忆用户如何使用：

```
# TF官方Optimizer
opt = tf.optimizers.Adam(scaled_lr)

# 把常规TensorFlow Optimizer通过Horovod包装起来，进而使用 ring-allreduce 来得到平均梯度
opt = hvd.DistributedOptimizer(
    opt, backward_passes_per_step=1, average_aggregated_gradients=True)

# 最后模型使用的是hvd.DistributedOptimizer
mnist_model.compile(loss=tf.losses.SparseCategoricalCrossentropy(),
                    optimizer=opt, metrics=['accuracy'],
                    experimental_run_tf_function=False)
```

opt 被传给DistributedOptimizer的optimizer，在构造函数`__init__.py`中被赋值给了self._optimizer。

```
if _LegacyOptimizer is not None:
    class _DistributedOptimizer(_LegacyOptimizer):
        """An optimizer that wraps another tf.Optimizer, using an allreduce to
        combine gradient values before applying gradients to model weights."""

        def __init__(self, optimizer, name=None, use_locking=False,
device_dense='',
                     device_sparse='', compression=Compression.none,
                     sparse_as_dense=False, op=Average,
gradient_predivide_factor=1.0,
                     backward_passes_per_step=1,
average_aggregated_gradients=False,
                     groups=None):

            self._optimizer = optimizer # 在构造函数中被赋值给了self._optimizer
            self._allreduce_grads = _make_allreduce_grads_fn( # 设置归并函数
                name, device_dense, device_sparse, compression, sparse_as_dense,
op,
                gradient_predivide_factor, groups)

            self._agg_helper = None
            if backward_passes_per_step > 1:
                # 可以先做本地梯度累积，再夸进程合并
                self._agg_helper = LocalGradientAggregationHelper(
                    backward_passes_per_step=backward_passes_per_step,
                    allreduce_func=self._allreduce_grads,
                    sparse_as_dense=sparse_as_dense,
                    average_aggregated_gradients=average_aggregated_gradients,
                    rank=rank(),

 optimizer_type=LocalGradientAggregationHelper._OPTIMIZER_TYPE_LEGACY,
                )
```

4.2 compute_gradients

计算梯度的第一步是 调用 compute_gradients 计算loss对指定val_list的梯度，返回元组列表 `list(zip(grads, var_list))`。

每一个worker的 tensor 模型都会调用 compute_gradients，对于每个model来说，

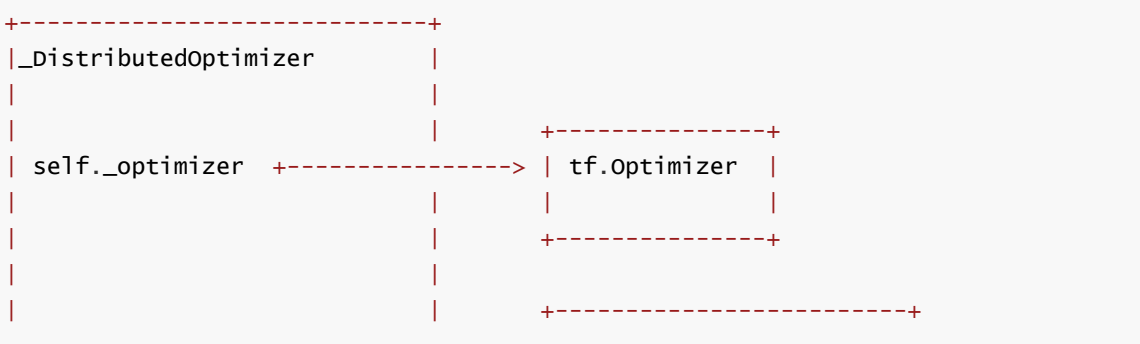
`gradients = self._optimizer.compute_gradients(*args, **kwargs)` 就是本 model 本地计算得到的梯度。

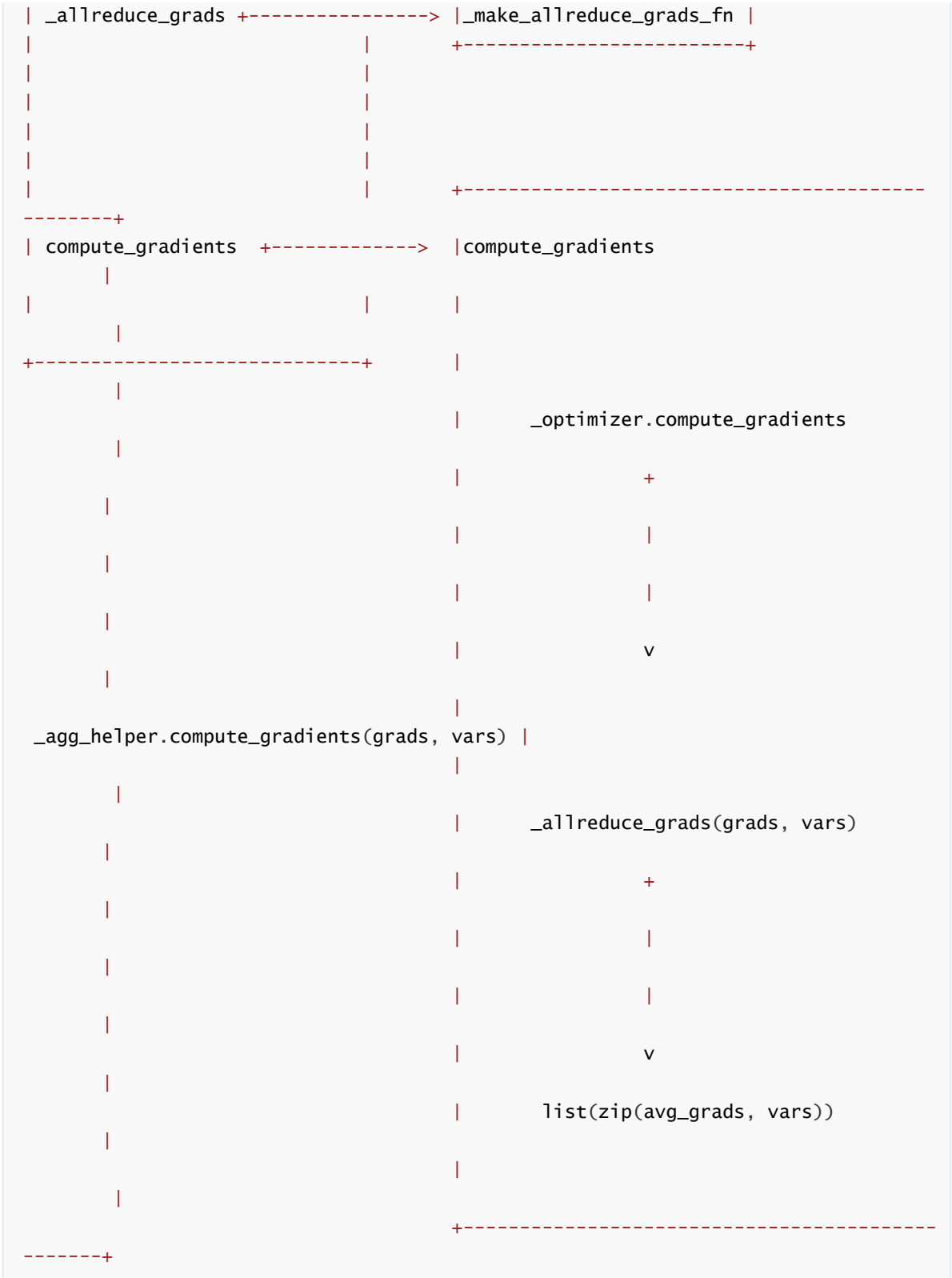
DistributedOptimizer 重写Optimizer类compute_gradients()方法。

- `_DistributedOptimizer` 初始化时候有配置 `self._allreduce_grads = _make_allreduce_grads_fn`。这里很重要。
- compute_gradients()方法首先调用原始配置TF官方 optimizer 的 compute_gradients()。compute_gradients()返回值是一个元祖列表，列表的每个元素是 (gradient，variable)，gradient是每一个变量变化的梯度值；
- 如果设置了 _agg_helper，即 LocalGradientAggregationHelper，就调用 LocalGradientAggregationHelper 来做本地梯度累积（本地累积目的是为了减少跨进程次数，只有当到了一定阶段之后才会进行跨进程合并），否则调用 _allreduce_grads 计算，即直接跨进程合并（用MPI对计算出来的分布式梯度做allreduce）；

```
        def compute_gradients(self, *args, **kwargs):
            """Compute gradients of all trainable variables.

            See Optimizer.compute_gradients() for more info.

            In DistributedOptimizer, compute_gradients() is overriden to also
            allreduce the gradients before returning them.
            """

            # _optimizer是原始配置的官方优化器，先调用其compute_gradients方法来计算所有训
练参数的梯度
            # 官方优化器的compute_gradients()方法返回一个元组(gradient，variable)的列表

            # gradients 被赋值为这个元组(gradient，variable)列表
            gradients = self._optimizer.compute_gradients(*args, **kwargs)
            grads, vars = zip(*gradients)

            if self._agg_helper: # 是否本地先累积
                avg_grads = self._agg_helper.compute_gradients(grads, vars)
            else:
                avg_grads = self._allreduce_grads(grads, vars)
            return list(zip(avg_grads, vars))
```

逻辑如下：





4.3 LocalGradientAggregationHelper

前面提到，如果设置了 _agg_helper，即 LocalGradientAggregationHelper，就调用 LocalGradientAggregationHelper 来做本地累积梯度（本地累积之后也会进行跨进程合并）。所以我们讲讲 LocalGradientAggregationHelper。

LocalGradientAggregationHelper 会在本地更新梯度，但是因为在初始化时候，成员函数 `self._allreduce_grads = allreduce_func` 就是跨进程allreduce函数。所以 LocalGradientAggregationHelper 之中也会进行跨进程 allreduce。即每次 backward_passes_per_step 时候跨机器更新一次。

这里需要注意的是：`allreduce_func=self._allreduce_grads`，其实`LocalGradientAggregationHelper`内部调用`self._allreduce_grads`也是调用到了_make_allreduce_grads_fn。

```
LocalGradientAggregationHelper(
                        backward_passes_per_step=backward_passes_per_step,
                        allreduce_func=self._allreduce_grads, # 就是
_make_allreduce_grads_fn
                        sparse_as_dense=sparse_as_dense,

 average_aggregated_gradients=average_aggregated_gradients,
                        rank=rank(),

 optimizer_type=LocalGradientAggregationHelper._OPTIMIZER_TYPE_KERAS,
                    )
```

具体是调用了`LocalGradientAggregationHelper.compute_gradients`完成功能，其中：

- _init_aggregation_vars 函数会 遍历 本地元组（gradient，variable）的列表，累积在locally_aggregated_grads。
- allreduce_grads 会做一个遍历 tensor & 应用 tensor 的操作，对于每个 tensor，_allreduce_grads_helper 函数会进行跨进程合并。

4.3.1 _init_aggregation_vars

_init_aggregation_vars 函数会 遍历 本地元组（gradient，variable）的列表，累积在locally_aggregated_grads。

```
def _init_aggregation_vars(self, grads):
    """
    Initializes the counter that is used when to communicate and aggregate
gradients
    and the tensorflow variables that store the locally aggregated gradients.
    """
    variable_scope_name = "aggregation_variables_" + str(self.rank)
    with tf.compat.v1.variable_scope(variable_scope_name,
reuse=tf.compat.v1.AUTO_REUSE):
        self.counter = tf.compat.v1.get_variable(
            "aggregation_counter", shape=(), dtype=tf.int32,
            trainable=False, initializer=tf.compat.v1.zeros_initializer(),
            collections=[tf.compat.v1.GraphKeys.LOCAL_VARIABLES],
        )
        # 遍历本地的梯度
        for idx, grad in enumerate(grads):
            # Handle IndexedSlices.
            # 如果是IndexedSlices，则转换为张量
            if self.sparse_as_dense and isinstance(grad, tf.IndexedSlices):
                grad = tf.convert_to_tensor(grad)
            elif isinstance(grad, tf.IndexedSlices):
                raise ValueError(
                    "IndexedSlices are not supported when "
                    "`backward_passes_per_step` > 1 and "
                    "`sparse_as_dense` is False."
                )

            # Handle grads that are None.
```

```
            # 如果为空，则跳过
            if grad is None:
                self.num_none_grad_updates += 1
                continue
            self.not_none_indexes[idx] = len(self.locally_aggregated_grads)

            # Create shadow variable.
            grad_aggregation_variable_name = str(idx)
            zero_grad = tf.zeros(shape=grad.get_shape().as_list(),
dtype=grad.dtype)
            grad_aggregation_variable = tf.compat.v1.get_variable(
                grad_aggregation_variable_name,
                trainable=False,
                initializer=zero_grad,
                collections=[
                    tf.compat.v1.GraphKeys.LOCAL_VARIABLES,
                    "aggregating_collection"],
            )
            # 添加到本地累积变量 locally_aggregated_grads 之中
            self.locally_aggregated_grads.append(grad_aggregation_variable)
        assert len(self.locally_aggregated_grads) + \
            self.num_none_grad_updates == len(grads)

    # We expect to get a `sess` when we need to manually do a `sess.run(...)`
    # for the variables to be initialized. This is the `tf.keras`
    # optimizers.
    # 遍历locally_aggregated_grads的变量，如果需要则进行初始化
    if self.optimizer_type == self._OPTIMIZER_TYPE_KERAS:
        session = tf.compat.v1.keras.backend.get_session(op_input_list=())
        vars_init_op = tf.compat.v1.variables_initializer(
            [self.counter,
*get_not_none_from_list(self.locally_aggregated_grads)]
        )
        session.run(vars_init_op)
```

4.3.2 compute_gradients

compute_gradients方法具体如下：

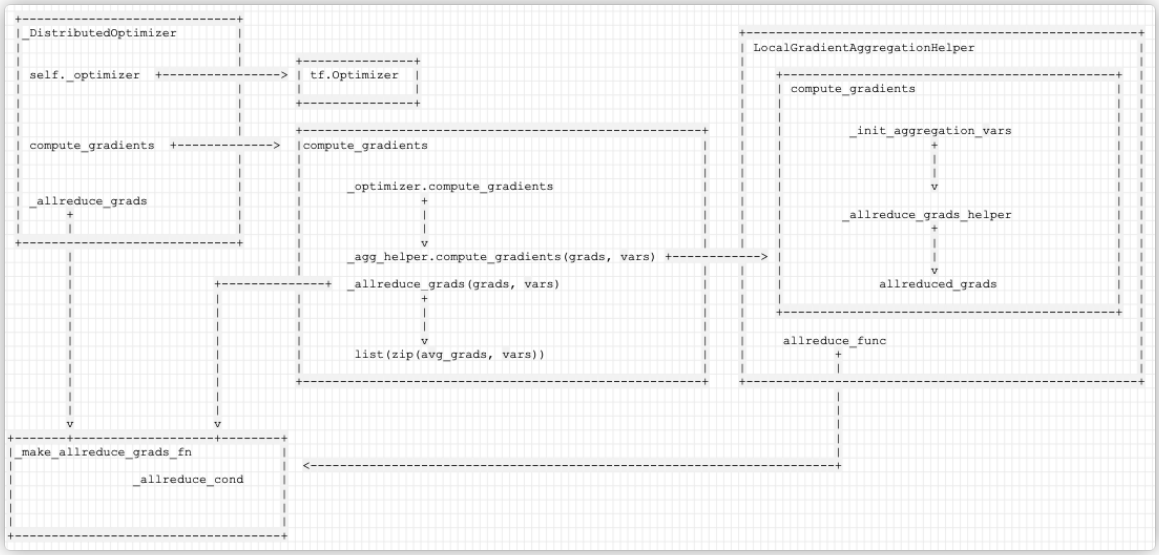
```
    def compute_gradients(self, grads, vars):
        """
        Applies the new gradient updates the locally aggregated gradients, and
        performs cross-machine communication every backward_passes_per_step
        times it is called.
        """
        # 遍历 本地元组（gradient，variable）的列表，累积在 locally_aggregated_grads
        self._init_aggregation_vars(grads)

        # Clear the locally aggregated gradients when the counter is at zero.
        # 如果计数器为0，则清理本地累积梯度
        clear_op = tf.cond(
            pred=tf.equal(self.counter, 0),
            true_fn=lambda: self._clear_grads(),
            false_fn=tf.no_op
        )

        # Add new gradients to the locally aggregated gradients.
```

```
        # 本地累积梯度
        with tf.control_dependencies([clear_op]):
            aggregation_ops_list = self._aggregate_grads(grads)

        # Increment the counter once new gradients have been applied.
        # 一旦本地梯度已经被应用，则把计数器加1
        aggregation_ops = tf.group(*aggregation_ops_list)
        with tf.control_dependencies([aggregation_ops]):
            update_counter = self.counter.assign_add(tf.constant(1))

        # 应用梯度
        with tf.control_dependencies([update_counter]):
            grads = get_not_none_from_list(grads)
            assert len(grads) == len(self.locally_aggregated_grads)

            # Allreduce locally aggregated gradients when the counter is
equivalent to
            # `backward_passes_per_step`. This the condition is true, it also
resets
            # the counter back to 0.
            allreduced_grads = tf.cond(
                tf.equal(self.counter, self.backward_passes_per_step), #判断是否可
以allreduce
                lambda: self._allreduce_grads_helper(grads, vars), #  # 如果
counter等于backward_passes_per_step，就对本地累积的梯度来应用跨进程allreduce
                lambda: grads, # 否则直接赋值为输入梯度
            )

            # Handle case where there is only one variable.
            if not isinstance(allreduced_grads, (list, tuple)):
                allreduced_grads = (allreduced_grads,)

            # Insert gradients that are None back in.
            # 对于本地累积的梯度，进行跨进程合并，locally_aggregated_grads是本地累积的梯
度
            allreduced_grads = [
                allreduced_grads[self.not_none_indexes[idx]] if idx in
self.not_none_indexes else None
                for idx in range(len(self.locally_aggregated_grads) +
self.num_none_grad_updates)
            ]

        # If gradients have not been allreduced this batch, we return the
gradients
        # that were submitted as the updates (the input).
        return allreduced_grads # # 返回跨进程合并之后的梯度/或者原来的输入梯度
```

逻辑拓展如下，这里需要注意的是 _agg_helper 或者 _allreduce_grads 选一个执行：

- 如果设置了 _agg_helper，即 LocalGradientAggregationHelper，就调用 _agg_helper 来计算梯度（本地累积之后也会进行跨进程合并）；
- 否则调用 _allreduce_grads，即 _make_allreduce_grads_fn 计算，即跨进程合并（用MPI来对计算出来的分布式梯度做allreduce操作）；

具体如下：



4.4 _make_allreduce_grads_fn

_make_allreduce_grads_fn 就是调用了 _make_cached_allreduce_grads_fn 完成功能。

```
def _make_allreduce_grads_fn(name, device_dense, device_sparse,
                             compression, sparse_as_dense, op,
                             gradient_predivide_factor, groups):
    groups = vars_to_refs(groups) if isinstance(groups, list) else groups
    return _make_cached_allreduce_grads_fn(name, device_dense, device_sparse,
                                           compression, sparse_as_dense, op,
                                           gradient_predivide_factor, groups)
```

_make_cached_allreduce_grads_fn 的作用是：

- 获取所有grads；
- 遍历元组(gradient，variable)的列表，对于每个grad，使用_allreduce_cond与其他worker进行同步；
- 最后返回同步好的梯度列表；

```
@_cache
def _make_cached_allreduce_grads_fn(name, device_dense, device_sparse,
                                    compression, sparse_as_dense, op,
                                    gradient_predivide_factor, groups):
    groups = refs_to_vars(groups) if isinstance(groups, tuple) else groups
    ......
    def allreduce_grads(grads, vars=None):
        with tf.name_scope(name + "_Allreduce"): # 设置名称空间
            ......
            # 获取所有的 grads
            # 因为grads列表致为((grad0,var0),(grad1,var1)…)，里面可能有很多None，所以提取出grad不为None的var进行梯度计算。
            return [_allreduce_cond(grad,
                                    device_dense=device_dense,
                                    device_sparse=device_sparse,
                                    compression=compression,
                                    op=op,
                                    prescale_factor=prescale_factor,
                                    postscale_factor=postscale_factor)
                    if grad is not None else grad
```

```
                    for grad in grads]

    if _executing_eagerly():
        return _make_subgraph(allreduce_grads)
    else:
        return allreduce_grads
```

_allreduce_cond 函数中就是调用到 allreduce 进行集合通信操作。

```
def _allreduce_cond(tensor, *args, **kwargs):
    def allreduce_fn():
        return allreduce(tensor, *args, **kwargs)

    def id_fn():
        return tensor

    return tf.cond((size_op() > 1) if int(os.environ.get("HOROVOD_ELASTIC", 0))
else tf.convert_to_tensor(size() > 1),
                   allreduce_fn, id_fn)
```

4.5 allreduce

allreduce()方法之中，会依据所需要传输的张量类型是Tensor还是 IndexedSlices 做不同处理。

- 如果 tensor类型是IndexedSlices，则只需要做allgather操作，是否需要其他操作需要看具体附加配置。
    - 因为对于分布在不同worker上的IndexedSlices，其values和indices彼此没有重复。
    - 假设在 worker 1上分布的indices是[1, 3, 5, 7, 9]，在worker 2上分布的indices是[2, 4, 6, 8, 10]。只需要使用allgather方法将其收集汇总得到 [1,2,3,4,5,6,7,8,9,10] 即可，不需要做求和/平均的操作。
    - 如果有附加操作，才需要进一步处理。
- 如果是 Tensor 类型，则需要调用_allreduce方法处理：先求张量的和，再取平均。

```
def allreduce(tensor, average=None, device_dense='', device_sparse='',
              compression=Compression.none, op=None,
              prescale_factor=1.0, postscale_factor=1.0,
              name=None):
    """Perform an allreduce on a tf.Tensor or tf.IndexedSlices.
    """
    op = handle_average_backwards_compatibility(op, average)

    if isinstance(tensor, tf.IndexedSlices): # 对于IndexedSlices类型
        # TODO: Need to fix this to actuall call Adasum
        if op == Adasum:
        with tf.device(device_sparse):
            # For IndexedSlices, do two allgathers instead of an allreduce.
            # 做两个allgathers操作即可
            horovod_size = tf.cast(size_op() if
int(os.environ.get("HOROVOD_ELASTIC", 0)) else size(),
                                   dtype=tensor.values.dtype)
            values = allgather(tensor.values) # 一个 allgeathers对value进行处理
            indices = allgather(tensor.indices) # 一个allgather对index进行处理

            # To make this operation into an average, divide allgathered values
by
```

```
            # the Horovod size.
                  # 如果op是Average，则需要计算所有value的均值，否则不做操作
            new_values = (values / horovod_size) if op == Average else values
        return tf.IndexedSlices(new_values, indices,
                                dense_shape=tensor.dense_shape)
    else: # 对于Tensor类型
        average_in_framework = False
        if rocm_built():
            # For ROCm, perform averaging at framework level
            average_in_framework = op == Average or op == Adasum
            op = Sum if op == Average else op

        with tf.device(device_dense):
            # 首先，将size_op()结果的类型转化为tensor的dtype类型
            horovod_size = tf.cast(size_op() if
int(os.environ.get("HOROVOD_ELASTIC", 0)) else size(),
                                   dtype=tensor.dtype)
            tensor_compressed, ctx = compression.compress(tensor)
            # 定义了一个sum/压缩操作：将某张量和其他所有Horovod进程同名张量求和
            summed_tensor_compressed = _allreduce(tensor_compressed, op=op,

 prescale_factor=prescale_factor,

 postscale_factor=postscale_factor,
                                                  name=name)
            summed_tensor = compression.decompress(summed_tensor_compressed,
ctx)
            if op == Adasum: # 处理其他附加操作
                if 'CPU' not in tensor.device and gpu_available('tensorflow'):
                    if nccl_built():
                        if not is_homogeneous:
                        elif not check_num_rank_power_of_2(int(size() /
local_size())):
                        if rocm_built():
                            horovod_local_size = tf.cast(local_size_op() if
int(os.environ.get("HOROVOD_ELASTIC", 0)) else local_size(),
                                                         dtype=tensor.dtype)
                            new_tensor = summed_tensor / horovod_local_size
                        else:
                            new_tensor = summed_tensor
                    else:
                        new_tensor = summed_tensor
                else:
                    new_tensor = summed_tensor
            else:
                if rocm_built():
                    new_tensor = (summed_tensor / horovod_size) if
average_in_framework else summed_tensor
                else:
                    new_tensor = summed_tensor
        return new_tensor
```

4.6 _allreduce

_allreduce方法 和 allgather方法在 horovod.tensorflow.mpi_ops.py 之中。

HorovodAllreduceOp和HorovodAllgatherOp这两个方法是HVD自定义的与tensorflow相关的OP。_allreduce 和 allgather 分别与之对应。

- _allreduce使用名字“HorovodAllreduce”和HorovodAllreduceOp绑定，由 MPI_LIB.horovod_allreduce 做了中间转换；
- allgather使用名字“HorovodAllgather”和HorovodAllgatherOp绑定，由 MPI_LIB.horovod_allgather 做了中间转换；

结合前面的 _make_cached_allreduce_grads_fn 之中对于名字空间的配置，张量名称大致为：`DistributedAdam_Allreduce/cond_14/HorovodAllreduce_grads_5_0`。

这样就调用到了 MPI 对应操作。

```
def _allreduce(tensor, name=None, op=Sum, prescale_factor=1.0,
postscale_factor=1.0,
               ignore_name_scope=False):
    """An op which reduces an input tensor over all the Horovod processes. The
    default reduction is a sum.

    The reduction operation is keyed by the name of the op. The tensor type and
    shape must be the same on all Horovod processes for a given name. The
reduction
    will not start until all processes are ready to send and receive the tensor.

    Returns:
      A tensor of the same shape and type as `tensor`, summed across all
      processes.
    """
    if name is None and not _executing_eagerly():
        name = 'HorovodAllreduce_%s' % _normalize_name(tensor.name)
    return MPI_LIB.horovod_allreduce(tensor, name=name, reduce_op=op,
                                     prescale_factor=prescale_factor,
                                     postscale_factor=postscale_factor,
                                     ignore_name_scope=ignore_name_scope)

def allgather(tensor, name=None, ignore_name_scope=False):
    """An op which concatenates the input tensor with the same input tensor on
    all other Horovod processes.

    The concatenation is done on the first dimension, so the input tensors on
the
    different processes must have the same rank and shape, except for the first
    dimension, which is allowed to be different.

    Returns:
      A tensor of the same type as `tensor`, concatenated on dimension zero
      across all processes. The shape is identical to the input shape, except
for
      the first dimension, which may be greater and is the sum of all first
      dimensions of the tensors in different Horovod processes.
    """
    if name is None and not _executing_eagerly():
        name = 'HorovodAllgather_%s' % _normalize_name(tensor.name)
    return MPI_LIB.horovod_allgather(tensor, name=name,
```

```
                                        ignore_name_scope=ignore_name_scope)
```

4.7 操作映射

Python世界中，调用 _allreduce 时传递了几个参数，比如tensor和name。其中 op=Sum 最为重要。这个是被 C++ 内部用来确定 reduction具体操作。我们具体梳理下：

4.7.1 C++定义

在 C++中有：

```
enum ReduceOp {
    AVERAGE = 0, // This value should never appear past framework code, as
                 // averaging is taken care of there.
    SUM = 1,
    ADASUM = 2
};

int horovod_reduce_op_sum() {
  return ReduceOp::SUM;
}
```

4.7.2 Python获取配置

在 python 的初始化代码中有：

```
class HorovodBasics(object):
    """Wrapper class for the basic Horovod API."""

    def __init__(self, pkg_path, *args):
        full_path = util.get_extension_full_path(pkg_path, *args)
        self.MPI_LIB_CTYPES = ctypes.CDLL(full_path, mode=ctypes.RTLD_GLOBAL)

        self.Average = self.MPI_LIB_CTYPES.horovod_reduce_op_average()
        self.Sum = self.MPI_LIB_CTYPES.horovod_reduce_op_sum() # 在这里联系起来
        self.Adasum = self.MPI_LIB_CTYPES.horovod_reduce_op_adasum()
```

这样，在调用 _allreduce 默认参数是 op=Sum，就对应了 C++ 的 ReduceOp::SUM。

4.7.3 建立联系

_allreduce 继续调用：

```
MPI_LIB.horovod_allreduce(tensor, name=name, reduce_op=op
```

MPI_LIB.horovod_allreduce被转换到了C++世界下面代码中

- 首先，通过OP_REQUIRES_OK的配置可以得到reduce_op_；
- 其次，ComputeAsync 之中通过 reduce_op_ 就可以确定具体需要调用那种操作；

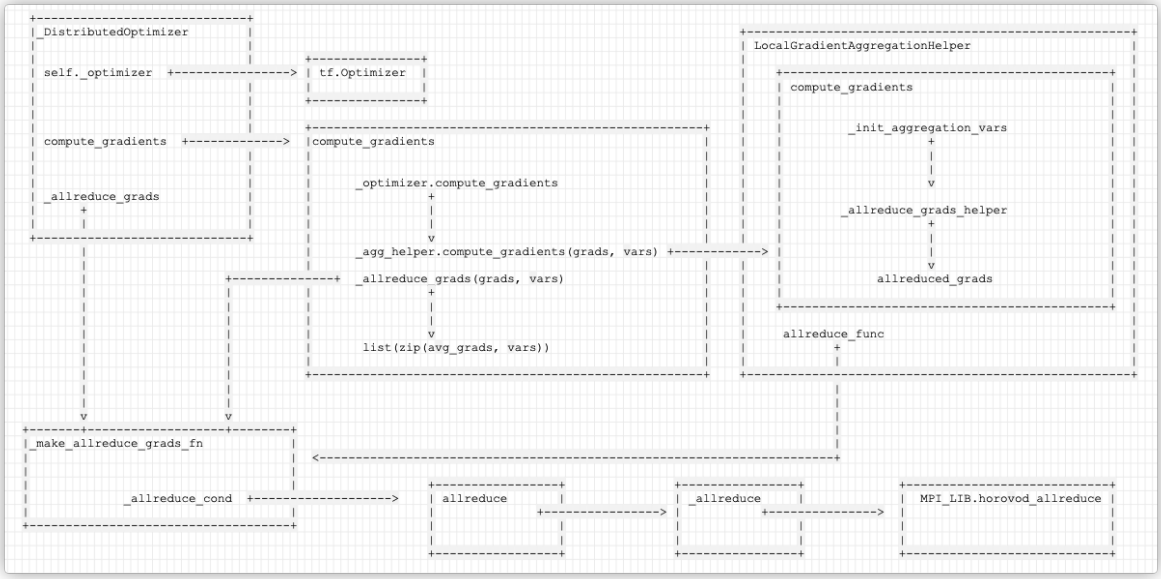
因此，Python和C++世界就进一步联系起来。

```
class HorovodAllreduceOp : public AsyncOpKernel {
public:
  explicit HorovodAllreduceOp(OpKernelConstruction* context)
      : AsyncOpKernel(context) {
```

```
    // 这里会声明，从 context 中得到reduce_op，赋值给reduce_op_
    OP_REQUIRES_OK(context, context->GetAttr("reduce_op", &reduce_op_));
    // 省略无关代码
  }

  void ComputeAsync(OpKernelContext* context, DoneCallback done) override {
    OP_REQUIRES_OK_ASYNC(context, ConvertStatus(common::CheckInitialized()),
                         done);
    // 省略无关代码
    // 这里会依据 reduce_op_，来确认C++内部调用何种操作
    horovod::common::ReduceOp reduce_op = static_cast<horovod::common::ReduceOp>
(reduce_op_);
    // 省略无关代码
  }
```

4.8 拓展流程

我们拓展目前流程图如下：



0x05 Tensorflow 2.x

5.1 Horovod 实施

对于 TF2.x，每行代码顺序执行，不需要构建图，也取消了control_dependency。Horovod 通过调用 TensorFlow 2.0 API 可以很直接地获取梯度。所以 Horovod 梯度更新部分的实现并不是基于计算图的实现，而是使用 `hvd.DistributedGradientTape`。

Worker 在训练时候做如下操作：

- 使用 DistributedGradientTape 封装 TF 官方的 Tape，配置 allreduce函数。
- 读取一组训练数据。
- 在本地模型调用前向传播函数计算损失。
- 给定损失之后，worker 利用 TensorFlow eager execution 的 GradientTape 机制，调用基类函数得到梯度。
- 各个Worker 会调用 Allreduce 来同步梯度。
- 各个Worker 会依据最新梯度相应更新模型。

5.2 示例代码

首先，我们给出示例代码如下，下面省略部分非关键代码，具体可以参见注释：

```
# Horovod: initialize Horovod.
hvd.init() # 初始化HVD

# Horovod: pin GPU to be used to process local rank (one GPU per process)
# 配置GPU
gpus = tf.config.experimental.list_physical_devices('GPU')
for gpu in gpus:
    tf.config.experimental.set_memory_growth(gpu, True)
if gpus:
    tf.config.experimental.set_visible_devices(gpus[hvd.local_rank()], 'GPU')

# 加载数据
(mnist_images, mnist_labels), _ = \
    tf.keras.datasets.mnist.load_data(path='mnist-%d.npz' % hvd.rank())

# 把数据进行特征切片
dataset = tf.data.Dataset.from_tensor_slices(
    (tf.cast(mnist_images[..., tf.newaxis] / 255.0, tf.float32),
             tf.cast(mnist_labels, tf.int64))
)
# 打乱数据，分批加载
dataset = dataset.repeat().shuffle(10000).batch(128)

mnist_model = tf.keras.Sequential([
    tf.keras.layers.Conv2D(32, [3, 3], activation='relu'),
    tf.keras.layers.Conv2D(64, [3, 3], activation='relu'),
    tf.keras.layers.MaxPooling2D(pool_size=(2, 2)),
    tf.keras.layers.Dropout(0.25),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(128, activation='relu'),
    tf.keras.layers.Dropout(0.5),
    tf.keras.layers.Dense(10, activation='softmax')
])
# 损失函数
loss = tf.losses.SparseCategoricalCrossentropy()

# Horovod: adjust learning rate based on number of GPUs.
opt = tf.optimizers.Adam(0.001 * hvd.size())

@tf.function
def training_step(images, labels, first_batch):
    with tf.GradientTape() as tape:
        probs = mnist_model(images, training=True)
        loss_value = loss(labels, probs)

    # Horovod: add Horovod Distributed GradientTape.
    # 调用 DistributedGradientTape，配置allreduce函数
    tape = hvd.DistributedGradientTape(tape)

    # 显式得到梯度，其内部经过一系列操作后，会调用horovod的allreduce操作，最终是
MPI_LIB.horovod_allreduce函数
    grads = tape.gradient(loss_value, mnist_model.trainable_variables)
    # 应用梯度，更新权重
```

```
    opt.apply_gradients(zip(grads, mnist_model.trainable_variables))

    # Horovod: broadcast initial variable states from rank 0 to all other
processes.
    # This is necessary to ensure consistent initialization of all workers when
    # training is started with random weights or restored from a checkpoint.
    #
    # Note: broadcast should be done after the first gradient step to ensure
optimizer
    # initialization.
    # 广播变量
    if first_batch:
        hvd.broadcast_variables(mnist_model.variables, root_rank=0)
        hvd.broadcast_variables(opt.variables(), root_rank=0)

    return loss_value


# Horovod: adjust number of steps based on number of GPUs.
for batch, (images, labels) in enumerate(dataset.take(10000 // hvd.size())):
    loss_value = training_step(images, labels, batch == 0)
```

5.3 _DistributedGradientTape

关键类_DistributedGradientTape 定义如下：

```
class _DistributedGradientTape(tf.GradientTape):
    def __init__(self, tape, device_dense, device_sparse, compression,
sparse_as_dense, op,
                 gradient_predivide_factor, groups, persistent=False,
                 watch_accessed_variables=True):
        if hasattr(tape, '_watch_accessed_variables'):
            super(self.__class__, self).__init__(persistent,
watch_accessed_variables)
        else:
            super(self.__class__, self).__init__(persistent)

        # 把TF官方tape保存起来
        self._tape = tape
        # 配置allreduce函数
        self._allreduce_grads = _make_allreduce_grads_fn(
            'DistributedGradientTape', device_dense, device_sparse, compression,
            sparse_as_dense, op, gradient_predivide_factor, groups)

    # 用户显式的调用此函数，其内部使用_make_allreduce_grads_fn进行处理
    def gradient(self, target, sources, output_gradients=None):
        # 调用基类函数获得梯度
        gradients = super(self.__class__, self).gradient(target, sources,
output_gradients)
        return self._allreduce_grads(gradients, sources)
```

_make_allreduce_grads_fn 函数会进行一系列调用，最终调用到 MPI_LIB.horovod_allreduce，具体做如下工作：

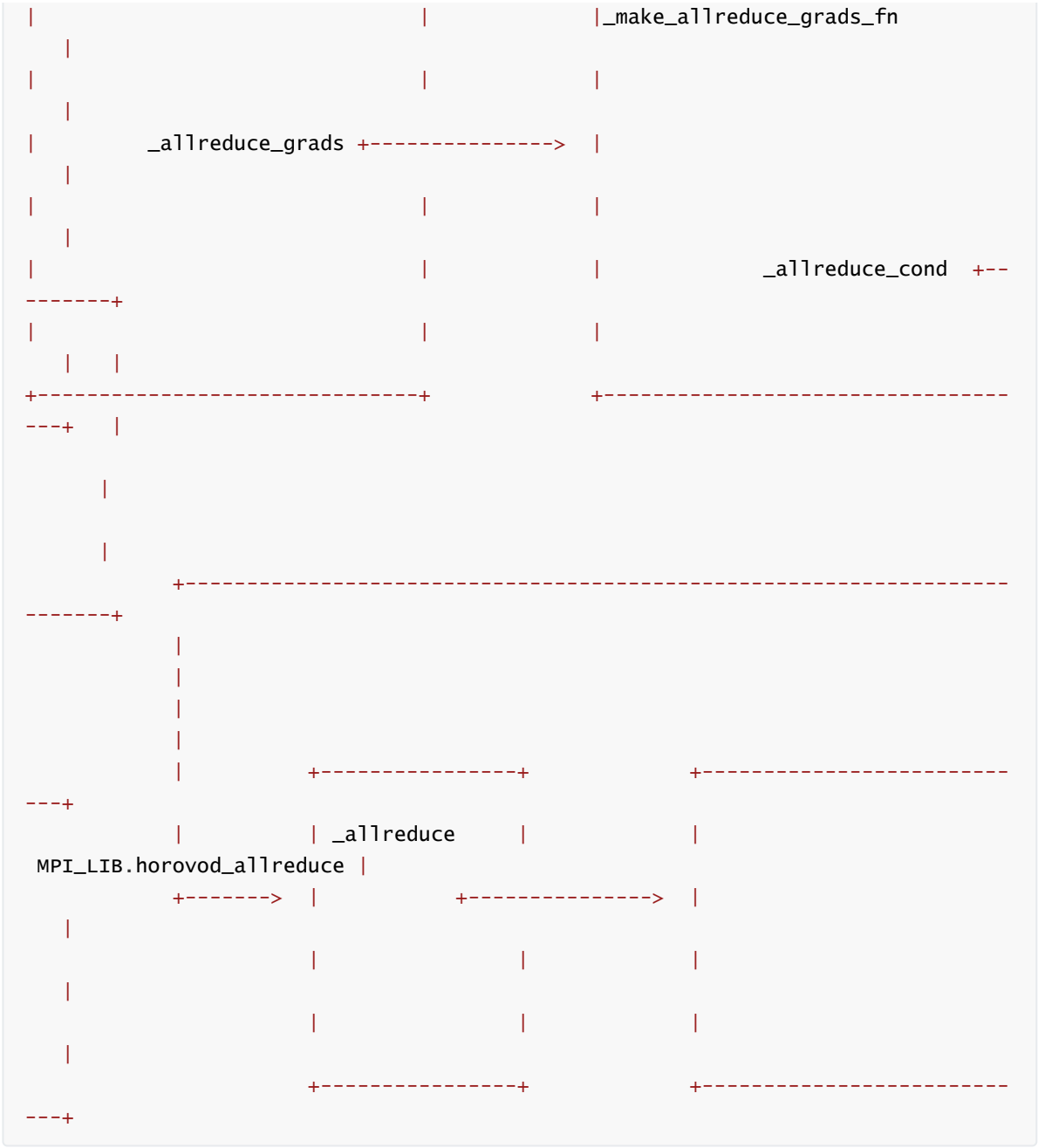
- 修改name scope，加上后缀 `_Allreduce`；
- 如果配置，则进行压缩；

- 依据op类型，调用allreduce 或者 直接返回tensor；
- DistributedGradientTape 的 name scope 被改写成了 DistributedGradientTape_Allreduce，名字被加上了 HorovodAllreduce_ 的前缀。
- 调用MPI_LIB.horovod_allreduce函数；

```
@_cache
def _make_allreduce_grads_fn(name, device_dense, device_sparse,
                             compression, sparse_as_dense, op):
    def allreduce_grads(grads):
        with tf.name_scope(name + "_Allreduce"): # 修改name scope，加上后缀
            if sparse_as_dense:
                grads = [tf.convert_to_tensor(grad) # 压缩
                         if grad is not None and isinstance(grad,
tf.IndexedSlices)
                         else grad for grad in grads]

            return [_allreduce_cond(grad,
                                    device_dense=device_dense,
                                    device_sparse=device_sparse,
                                    compression=compression,
                                    op=op)
                    if grad is not None else grad
                    for grad in grads]

def _allreduce_cond(tensor, *args, **kwargs):
    def allreduce_fn():
        return allreduce(tensor, *args, **kwargs)

    def id_fn():
        return tensor

    return tf.cond(size_op() > 1, allreduce_fn, id_fn) # 不用的调用方法

def _allreduce(tensor, name=None, op=Sum):
    """An op which reduces an input tensor over all the Horovod processes. The
    default reduction is a sum.

    The reduction operation is keyed by the name of the op. The tensor type and
    shape must be the same on all Horovod processes for a given name. The
reduction
    will not start until all processes are ready to send and receive the tensor.

    Returns:
      A tensor of the same shape and type as `tensor`, summed across all
      processes.
    """
    if name is None and not _executing_eagerly():
        name = 'HorovodAllreduce_%s' % _normalize_name(tensor.name)
    # # 调用HorovodAllreduceOp
    return MPI_LIB.horovod_allreduce(tensor, name=name, reduce_op=op)
```

逻辑如下：

```
+------------------------------+
| _DistributedGradientTape     |            +---------------------------------
---+
```



0x06 HorovodAllreduceOp

MPI_LIB.horovod_allreduce 调用的就是 HorovodAllreduceOp。MPI_LIB.horovod_allreduce 是 python 函数，HorovodAllreduceOp 是C++代码，这里 TF 做了一个适配和转换，让我们可以从 python 函数直接调用到 C++ 函数。

HorovodAllreduceOp 继承了AsyncOpKernel，是一种TF Async OP，而且被 REGISTER_KERNEL_BUILDER 注册到 TF，因此就可以嵌入到 TF 流程之中。

TF 会调用到 HorovodAllreduceOp 所覆盖的ComputeAsync方法，在ComputeAsync内部会把 张量的Allreduce操作加入Horovod后台队列，从而把 TF OP 和 Horovod OP 联系起来。

总结一下，HorovodAllreduceOp 继承了TF AsyncOpKernel，因此可以嵌入到 TF 流程，同时用组合方式与 Horovod 后台线程联系起来。

```
class HorovodAllreduceOp : public AsyncOpKernel { //派生了，所以可以嵌入到 TF流程之中
public:
  explicit HorovodAllreduceOp(OpKernelConstruction* context)
      : AsyncOpKernel(context) {
    OP_REQUIRES_OK(context, context->GetAttr("reduce_op", &reduce_op_));
```

```
    OP_REQUIRES_OK(context, context->GetAttr("prescale_factor",
&prescale_factor_));
    OP_REQUIRES_OK(context, context->GetAttr("postscale_factor",
&postscale_factor_));
    OP_REQUIRES_OK(context, context->GetAttr("ignore_name_scope",
&ignore_name_scope_));
  }

  void ComputeAsync(OpKernelContext* context, DoneCallback done) override {
    OP_REQUIRES_OK_ASYNC(context, ConvertStatus(common::CheckInitialized()),
                         done);
    ... // 省略一些变量验证，初始化代码

    // 将张量的Allreduce操作OP加入队列
    auto enqueue_result = EnqueueTensorAllreduce(
        hvd_context, hvd_tensor, hvd_output, ready_event, node_name, device,
        [context, done](const common::Status& status) {
          context->SetStatus(ConvertStatus(status));
          done();
        }, reduce_op, (double) prescale_factor_, (double) postscale_factor_);
    OP_REQUIRES_OK_ASYNC(context, ConvertStatus(enqueue_result), done);
  }

private:
  int reduce_op_;
  // Using float since TF does not support double OP attributes
  float prescale_factor_;
  float postscale_factor_;
  bool ignore_name_scope_;
};
```

从下文开始我们看看Horovod on Spark。