Collective in Ray

Motivation

See this RFC for a full description of why we need collectives in Ray.

Status

Collectives Support Matrix

See below the support matrix for collective calls with different backends. NCCL-based collective are in place (merged to Ray master) whereas GLOO support is mostly working in progress.

- ✘ : do not have plan to support them.
- WIP: in implementation, or to be merged
- ✓ : in Ray master

Backend	**GLOO**		**NCCL**	
Device	**CPU**	**GPU**	**CPU**	**GPU**
send	✓	✘	✘	✓
recv	✓	✘	✘	✓
broadcast	✓	✘	✘	✓
all_reduce	✓	✘	✘	✓
reduce	✓	✘	✘	✓
all_gather	✓	✘	✘	✓

gather	WIP	✘	✘	✘
scatter	WIP	✘	✘	✘
reduce_scatter	✓	✘	✘	✓
all_to_all	✘	✘	✘	WIP
barrier	✓	✘	✘	✓

Supported Tensor Types

- `torch.Tensor`
- `numpy.ndarray`
- `cupy.ndarray`

Importing and Code References

To use these APIs, users are expected to import the collective package in their actor/task or driver code via:

```
import ray.util.collective as col
```

Code references:

- python/ray/util/collective/collective.py
- python/ray/util/collective/types.py
- python/ray/util/collective/collective_group/basic_collective_group.py
- python/ray/util/collective/collective_group/nccl_collective_group.py
- python/ray/util/collective/examples

APIs

This section describes the user APIs of the collective functions under `ray.util.collective`.

Initialization

Collective functions operate on *collective groups*. A collective group contains a number of processes (in Ray, Ray-managed actors or tasks) that will enter the collective function calls. Before making collective calls, we expect users to declare a set of actors/tasks, *statically*, as a collective group. We currently provide three APIs for collective group initialization.

```
def init_collective_group(world_size: int,
                          rank: int,
                          backend=types.Backend.NCCL,
                          group_name: str = "default"):
    """Initialize a collective group inside an actor/task process.

    Args:
        world_size (int): the total number of processes in the group.
        rank (int): the rank of the current process.
        backend: the CCL backend to use, NCCL or GLOO.
        group_name (str): the name of the collective group.

    Returns:
        None
    """
```

This above API is supposed to be called inside an actor/task code. Every actor/task that enters the collective call shall make the above call at least once in order to rendezvous with each other.

Alternatively, we provide a declarative API that enables declaring collective groups in driver programs (i.e., out of the collective process).

```
def declare_collective_group(actors,
                             world_size: int,
                             ranks: List[int],
                             backend=types.Backend.NCCL,
                             group_name: str = "default"):
    """Declare a list of actors as a collective group.

    Note: This function should be called in a driver process.

    Args:
        actors (list): a list of actors to be set in a collective group.
        world_size (int): the total number of processes in the group.
        ranks (List[int]): the rank of each actor.
```

```
        backend: the CCL backend to use, NCCL or GLOO.
        group_name (str): the name of the collective group.

    Returns:
        None
    """
```

Note that for the same set of actors/task processes, multiple collective groups can be constructed, with *group_name (str)* as their unique identifier. This enables to specify complex communication patterns between different (sub)set of processes.

Example Code

```
@ray.remote(num_gpus=1)
class Worker:
   def __init__(self):
       self.send = cp.ones((4, ), dtype=cp.float32)
       self.recv = cp.zeros((4, ), dtype=cp.float32)

   def setup(self, world_size, rank):
       collective.init_collective_group(world_size, rank, "nccl", "default")
       return True

   def compute(self):
       collective.allreduce(self.send, "default")
       return self.send

   def destroy(self):
       collective.destroy_group()

# imperative
num_workers = 2
workers = []
init_rets = []
for i in range(num_workers):
   w = Worker.remote()
   workers.append(w)
   init_rets.append(w.setup.remote(num_workers, i))
_ = ray.get(init_rets)
results = ray.get([w.compute.remote() for w in workers])


# declarative
```

```
for i in range(num_workers):
   w = Worker.remote()
   workers.append(w)
_options = {
   "group_name": "177",
   "world_size": 2,
   "ranks": [0, 1],
   "backend": "nccl"
}
collective.declare_collective_group(workers, **_options)
results = ray.get([w.compute.remote() for w in workers])
```

Basic APIs

A set of basic APIs to query backend availability, group existence, group world size, or process ranks, or destroy groups, are provided. See below their signatures.

```
def nccl_available():
    """Check if nccl backend is available."""

def gloo_available():
    """Check if gloo backend is available."""


def is_group_initialized(group_name):
   """Check if the group is initialized in this process by the group name."""


def destroy_collective_group(group_name: str = "default") -> None:
   """Destroy a collective group given its group name."""


def get_rank(group_name: str = "default") -> int:
   """Return the rank of this process in the given group.

   Args:
       group_name (str): the name of the group to query

   Returns:
       the rank of this process in the named group,
       -1 if the group does not exist or the process does
```

```
        not belong to the group.
    """

def get_world_size(group_name: str = "default") -> int:
    """Return the size of the collective group with the given name.

    Args:
        group_name: the name of the group to query

    Returns:
        The world size of the collective group, -1 if the group does
            not exist or the process does not belong to the group.
    """
```

Collective Functions

Check the matrix for the current status of supported collective calls and backend.

We currently provide two classes of APIs to make collective calls: imperative collective APIs, and declarative collective APIs.

Imperative collective APIs

Below are some example signatures of the collective functions:

```
def allreduce(tensor, group_name: str = "default", op=types.ReduceOp.SUM):
    """Collective allreduce the tensor across the group.

    Args:
        tensor: the tensor to be all-reduced on this process.
        group_name (str): the collective group name to perform allreduce.
        op: the reduce operation.

    Returns:
        None
    """

def broadcast(tensor, src_rank: int = 0, group_name: str = "default"):
    """Broadcast the tensor from a source process to all others.
```

```
    Args:
        tensor: the tensor to be broadcasted (src) or received (dst).
        src_rank (int): the rank of the source process.
        group_name (str): the collective group name to perform broadcast.

    Returns:
        None
    """


def allgather(tensor_list: list, tensor, group_name: str = "default"):
    """Allgather tensors from each process of the group into a list.

    Args:
        tensor_list (list): the results, stored as a list of tensors.
        tensor: the tensor (to be gathered) in the current process
        group_name (str): the name of the collective group.

    Returns:
        None
    """
```

The imperative APIs exhibit the following behaviours:

- All the collective APIs are synchronous blocking calls。
- Since each API only specifies a part of the collective communication, the API is expected to be called by each participating process of the (pre-declared) collective group. Upon all the processes have made the call and rendezvous with each other, the collective communication happens and proceeds.
- The APIs are imperative --- they need to be used inside the collective process (actor/task) code.

[WIP] Declarative Collective APIs

There is an ongoing effort on developing a set of *declarative* collective APIs, in addition to the imperative one. See the RFC-202012-ObjectRef-compatible-collectives for a full description of the motivation and design.

Below are some example signatures of the declarative collective APIs. ***Note that these APIs are under development, have not been merged into Ray master, and are still subject to changes.***

```
def allreduce_refs(tensor_refs: list,
                   group_name: str = "default",
                   op=types.ReduceOp.SUM):
    """Collective allreduce the tensors across the group.

    This APIs takes a list of ObjectRefs as input instead of the tensors.

    Args:
        tensor_refs: the ObjectRefs to the list of tensors; Each ref
corresponds   to a tensor on a collective participant process.
        group_name (str): the collective group name to perform allreduce.
        op: The reduce operation.

    Returns:
        None
    """

def reduce_refs(tensor_refs,
                dst_rank: int = 0,
                group_name: str = "default",
                op=types.ReduceOp.SUM):
    """Reduce the tensors across the group to the destination rank.

    Args:
        tensor_refs: the ObjectRefs to the list of tensors; each ref
corresponds to a tensor on a collective participant process.
        dst_rank (int): the rank of the destination process.
        group_name (str): the collective group name to perform reduce.
        op: The reduce operation.

    Returns:
        None
    """


def allgather_refs(tensor_list_refs: list,
                   tensor_refs,
                   group_name : str = "default"):
    """Allgather tensors from each process of the group into a list.

    Args:
        tensor_list_refs (list): a list of ObjectRefs, each ObjectRef refers
```

```
to a list of tensors
           on a collective process.
       tensor_refs (list): a list of ObjectRefs, each ObjectRef refers to a
tensor on a collective
           process.
       group_name (str): the name of the collective group.

   Returns:
       None
   """
```

The declarative collective APIs exhibits the following behaviors and patterns:

- Different from the imperative ones, the declarative collective APIs specify the entire collective communication via a single API call, by passing ObjectRefs of tensors owned by all other participating processes. These collective-compatible ObjectRefs are realized by a simplified Python Object Store (POS, see RFC-202012-ObjectRef-compatible-collectives for details), with limited functionality compared to a normal Ray ObjectRef.
- Hence, the declarative API is supposed to be called in a driver program, instead of inside the actor/task code.

Example Code

```
@ray.remote
class Worker:
    def __init__(self):
        self.buffer = cupy.ones((10,), dtype=cupy.float32)

    def get_buffer(self)
        Return self.buffer
```

```
# Create two actors and create a collective group
A = Worker.remote()
B = Worker.remote()
col.declare_collective_group([A, B], options={rank=[0, 1], ...})

# Specify a collective allreduce "completely" instead of "partially" on
each actor
col.allreduce_refs([A.get_buffer.remote(), B.get_buffer.remote()])
```

Point-to-point Communication Functions

`Ray.util.collective` also supports P2P `send/recv` functions between processes or GPU devices. See below the signatures of `send` and `recv` APIs; both imperative and declarative versions of the APIs are provided. The declarative set of `send/recv` calls are still working in progress.

```python
def send(tensor, dst_rank: int, group_name: str = "default"):
    """Send a tensor to a remote process synchronously.

    Args:
        tensor: the tensor to send.
        dst_rank (int): the rank of the destination process.
        group_name (str): the name of the collective group.

    Returns:
        None
    """


def recv(tensor, src_rank: int, group_name: str = "default"):
    """Receive a tensor from a remote process synchronously.

    Args:
        tensor: the received tensor.
        src_rank (int): the rank of the source process.
        group_name (str): the name of the collective group.

    Returns:
        None
    """

def send_ref(src_tensor_ref, dst_tensor_ref, group_name: str = "default"):
    """Send a tensor to a destination process synchronously.

    Args:
        src_tensor_ref: the ObjectRef of the source tensor.
        dst_tensor_ref: the ObjectRef of the destination tensor.
        group_name (str): the name of the collective group.

    Returns:
        None
```

```
    """

def recv_ref(dst_tensor_ref, src_tensor_ref, group_name: str = "default"):
    """Send a tensor to a destination process synchronously.

    Args:
        dst_tensor_ref: the ObjectRef of the destination tensor.
        src_tensor_ref: the ObjectRef of the source tensor.
        group_name (str): the name of the collective group.

    Returns:
        None
    """
```

The `send/recv` exhibit the same behaviour with the collective functions.

- Imperative P2P functions are synchronous blocking calls -- a pair of `send` and `recv` must be called together on paired processes in order to specify the entire communication, and must successfully rendezvous with each other to proceed.
- Declarative P2P functions specify the P2P communication using a single API call, either via `send` or `recv`. The ObjectRefs passed through `send/recv` calls are backed by POS, hence have all the limitations that POS imposes.

Example Code

```
@ray.remote
class Worker:
    def __init__(self):
        self.buffer = cupy.ones((10,), dtype=cupy.float32)

    def get_buffer(self)
        return self.buffer

    def do_send(self, target_rank=0):
        # this call is blocking
        col.send(target_rank)

    def do_recv(self, src_rank=0):
        # this call is blocking
        col.recv(src_rank)

    def do_allreduce(self):
        # this call is blocking as well
```

```
        col.allreduce(self.buffer)
        return self.buffer


# Create two actors
A = Worker.remote()
B = Worker.remote()

# Put A and B in a collective group using existing APIs
col.declare_collective_group([A, B], options={rank=[0, 1], ...})

# let A to send a message to B, the only way to trigger and complete this
send/recv is by:
# Note in the code below, a send/recv has to be specified once at each
worker
ray.get([a.do_send.remote(target_rank=1), b.do_recv.remote(src_rank=0)])

# An anti-pattern: the following code will hang, because it does
instantiate the recv side call
ray.get([a.do_send.remote(target_rank=1)])

# Declarative send/recv: specify a send/recv via refs
A.recv_ref(B.get_buffer.remote())
```

Single-GPU and Multi-GPU Collective and P2P Functions

In many cluster setups, a machine usually has more than 1 GPUs, effectively leveraging the GPU-GPU bandwidth can significantly improve communication performance.

`ray.util.collective` supports multi-GPU collective calls, in which case, a process (actor/tasks) manages more than 1 GPUs (e.g., `ray.remote(num_gpus=4)`). Using multiGPU collective functions are normally more performance-advantageous than spawning the number of processes equal to the number of GPUs.

Some example signature of multi-GPU collective and P2P functions are below:

```
def allreduce_multigpu(tensor_list: list,
                       group_name: str = "default",
                       op=types.ReduceOp.SUM):
    """Collective allreduce a list of tensors across the group.
```

```
    Args:
        tensor_list (List[tensor]): list of tensors to be allreduced,
            each on a GPU.
        group_name (str): the collective group name to perform allreduce.

    Returns:
        None
    """


def broadcast_multigpu(tensor_list,
                       src_rank: int = 0,
                       src_tensor: int = 0,
                       group_name: str = "default"):
    """Broadcast the tensor from a source GPU to all other GPUs.

    Args:
        tensor_list: the tensors to broadcast (src) or receive (dst).
        src_rank (int): the rank of the source process.
        src_tensor (int): the index of the source GPU on the source process.
        group_name (str): the collective group name to perform broadcast.

    Returns:
        None
    """


def reducescatter(tensor,
                  tensor_list: list,
                  group_name: str = "default",
                  op=types.ReduceOp.SUM):
    """Reducescatter a list of tensors across the group.

    Reduce the list of the tensors across each process in the group, then
    scatter the reduced list of tensors -- one tensor for each process.

    Args:
        tensor: the resulted tensor on this process.
        tensor_list (list): The list of tensors to be reduced and scattered.
        group_name (str): the name of the collective group.
        op: The reduce operation.

    Returns:
```

```
        None
    """


def send_multigpu(tensor,
                  dst_rank: int,
                  dst_gpu_index: int,
                  group_name: str = "default"):
    """Send a tensor to a remote GPU synchronously.

    The function assume each process owns >1 GPUs, and the sender
    process and receiver process has equal number of GPUs.

    Args:
        tensor: the tensor to send, located on a GPU.
        dst_rank (int): the rank of the destination process.
        dst_gpu_index (int): the destination gpu index.
        group_name (str): the name of the collective group.

    Returns:
        None
    """
```

All multi-GPU APIs are with the following assumptions:

- Only NCCL backend is (will be) supported.
- Collective processes that make multi-GPU collective or P2P calls need to own the same number of GPU devices.
- The input to multiGPU collective functions are normally a list of tensors, eash located on a different GPU device.

Example Code

```
import ray.util.collective as collective
from cupy.cuda import Device


@ray.remote(num_gpus=2)
class Worker:
    def __init__(self):
        with Device(0):
```

```python
            self.send1 = cp.ones((4, ), dtype=cp.float32)
        with Device(1):
            self.send2 = cp.ones((4, ), dtype=cp.float32) * 2
        with Device(0):
            self.recv1 = cp.ones((4, ), dtype=cp.float32)
        with Device(1):
            self.recv2 = cp.ones((4, ), dtype=cp.float32) * 2

    def setup(self, world_size, rank):
        collective.init_collective_group(world_size, rank, "nccl", "177")
        return True

    def allreduce_call(self):
        collective.allreduce_multigpu([self.send1, self.send2], "177")
        return [self.send1, self.send2], self.send1.device,
self.send2.device

    def p2p_call(self):
        if self.rank == 0:
           collective.send_multigpu(self.send1 * 2, 1, 1, "8")
        else:
           collective.recv_multigpu(self.recv2, 0, 0, "8")
        return self.recv2

# Note that the world size is 2 but there are 4 GPUs.
num_workers = 2
workers = []
init_rets = []
for i in range(num_workers):
    w = Worker.remote()
    workers.append(w)
    init_rets.append(w.setup.remote(num_workers, i))
a = ray.get(init_rets)
results = ray.get([w.allreduce_call.remote() for w in workers])
results = ray.get([w.p2p_call.remote() for w in workers])
```

Microbenchmarks

NCCL Microbenchmarks

- Benchmark #1: Single node, each node has two Titan X GPUs
  - Spawning two actors, each actor is allocated with 1 GPU; Communication between two actors
- Benchmark #2: single node, each node has two P40 GPUs; communication between two GPUs
  - Spawning two actors, each actor is allocated with 1 GPU; Communication between two actors
- Benchmark #3: A cluster with 16 nodes, each node has 1 GPU
  - Spawning up to 16 actors, each actor is allocated with 1 GPU; communication between 16 actors.
- Benchmark #4: send/recv round-trip experiments on single AWS p3.8x, AWS g4dn.12, and p2.8 instance
  - 2 actors, each allocated with num_gpus=1, round trip.

GLOO Microbenchmarks

- A cluster with 16 nodes
  - Spawning an actor process on each node (num_cpus=1); communicating between 16 nodes.

Real Use case: Scaling Up Spacy Pipeline

Case Description

Spacy-ray is an extension that allows distributed training Spacy-based ML pipeline using Ray. Spacy-ray implements a *sharded parameter server* by communicating gradients using the `ray.get()` and `ray.set()` RPC calls. The slowness is mainly caused by (based on the microbenchmark results):

- RPC is less optimized when using them to assemble PS-related operations.
- `ray.get()` and `ray.set()` causes substantial memory movement overhead when the object is stored on GPU memory instead of RAM.
- `ray.get()` and `ray.set()` via Ray object store causes substantial overhead when serializing and deserializing objects.

Solution

We simply modify the communication method by replacing the `ray.get()` and `ray.set()` with `ray.util.collective.allreduce` APIs, and keep the overall structure unchanged.

Key change to the original spacy-ray is the way that workers increment gradients (Below codes are simplified for better readability, please refer to githubs for actual codes):

In spacy-ray, the worker first updates relevant status, then it updates the gradients if the parameter is stored locally, or invokes the `inc_grad()` method remotely on the parameter server shard that holds the parameter.

```
def inc_grad(self, key, value):
    self._grad_counts[key] += 1
    if key not in self._owned_keys:
        peer = self.peers[key]
        peer.inc_grad.remote(key, self._version[key], value)
    else:
        if self._grads.get(key) is None:
            self._grads[key] = value.copy()
        else:
            self._grads[key] += value
```

In our implementation, all workers will invoke `inc_grad()` at the same time, and call allreduce with operation SUM to communicate the gradients:

```
def inc_grad(self, key, value):
    self._grad_counts[key] += 1
    grad = value.copy()
    collective.allreduce(grad, "default")
    if self._grads.get(key) is None:
        self._grads[key] = value.copy()
    else:
        self._grads[key] += value
```

Implementation Details

We keep the same CLI as a spacy-ray. The user can modify the config file for training hyperparameter and project.yml for cluster/pipeline settings. After specifying the desired setup, and launching the ray cluster, the user can launch the distributed jobs using:

```
spacy project run ray-train
```

Side note: the original Spacy-ray depends on Ray v0.8; we migrate from Ray v0.8 to Ray v1.2 as well in order to use `ray.util.collective`.

Code Reference

- spacy_ray_nccl/proxies.py

Results

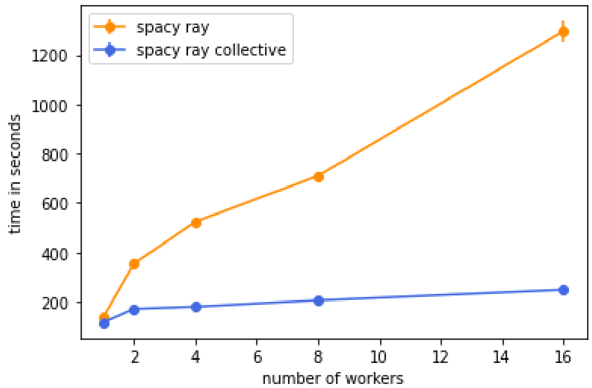
The runtime comparison for 1000 updates using spacy pipeline = [**"tok2vec", "ner"**] is shown below. Computation and communication happen sequentially all on the **default CUDA stream**. Mean and standard deviation are obtained by three trials (unit: second)

# workers	Spacy-ray	spacy-ray-nccl-allreduce	speedup
1 worker	137.5 ± 2.1	116.7 ± 2.51	**1.18x**
2 workers	354.1 ± 16.8	171.1 ± 1.11	**2.07x**
4 workers	523.9 ± 10.4	179.6 ± 2.91	**2.92x**
8 workers	710.1 ± 3.0	205.8 ± 1.20	**3.45x**
16 workers	1296.1 ± 42.1	248.3 ± 3.63	**5.22x**

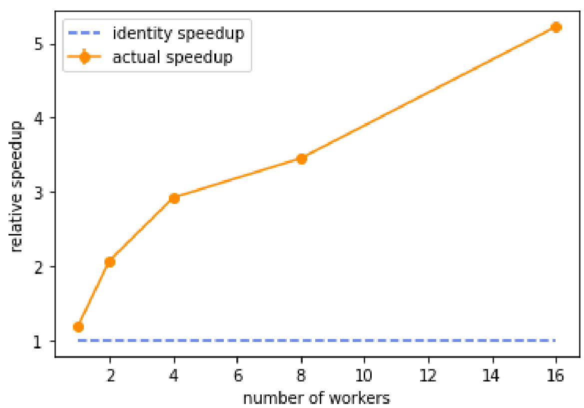
*Absolute scalability (system throughput compared to 1 worker. For n workers, it is computed by (n * runtime of 1 worker) / (runtime of n workers)):*

Comparison	Spacy-ray	spacy-ray-nccl
2 workers	0.77	1.36
4 workers	1.05	2.60
8 workers	1.55	4.54
16 workers	1.70	7.52

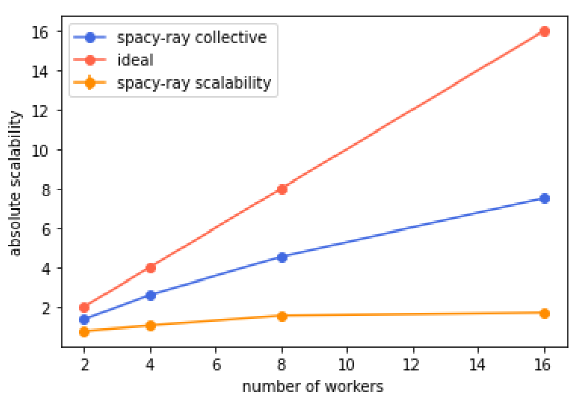
Runtime Comparison



Speedup Comparison



Scalability Comparison



Summary of Results

The experiments on 1, 2, 4, 8, 16 nodes (each with 1 GPU) show that our newly added collective functionality enables more scalability. One the 16 nodes setting, we speed up training by **5.22x speedup** over the original Spacy-ray, **using a single GPU NULL stream for both computation and communication**. The code has been merged into Ray master.

A WIP experimental multi-stream version that allocates separate streams for NCCL kernels (so *the computation and communication is overlapped*) is under development, and preliminary results show around **6.82x speedup over spacy-ray**.