PyTorch Distributed and Parallel Training - 3

20230419
박현우

Table of Contents

1. What is RPC-based Distributed Training
2. Remote Reference Protocol
3. Distributed Autograd Engine

1. Using RPC Framework
2. Combining with DDP
3. Pipeline Parallelism

1. What is RPC-based Distributed Training

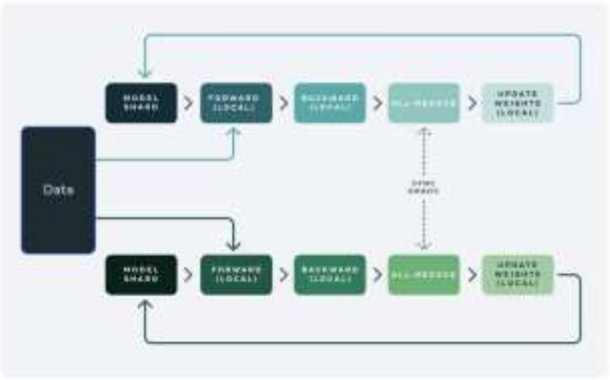
1. Needs for RPC-based Distributed training
2. Advantage of RPC over DDP/FSDP
3. What is torch.distributed.rpc package
4. CUDA RPC

1.1 Needs for RPC-based Distributed Training

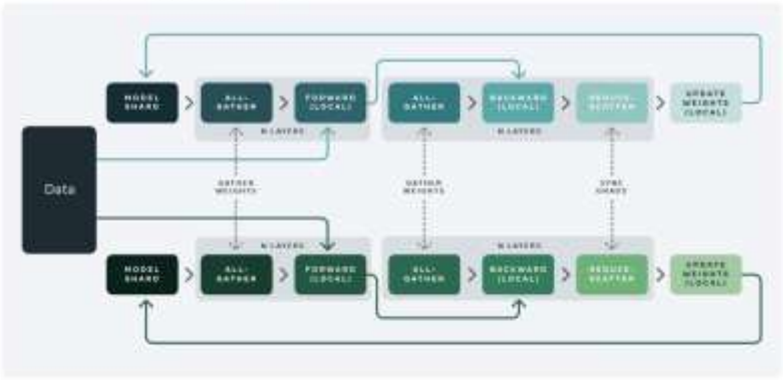
DDP/FSDP support very specific training paradigm:

1. the model is replicated across multiple processes
2. and each process handles a shard of the input data.

Standard data parallel training



Fully sharded data parallel training



1.1 Needs for RPC-based Distributed Training

Most common scenarios for different training paradigms:

1. Reinforcement learning
2. Parameter Server Architecture

1.1 Needs for RPC-based Distributed Training

Reinforcement learning

Expensive to acquire training data from environments while the model itself can be quite small.

Thus, it might be useful

1. to spawn multiple observers running in parallel
2. and share a single agent.

1.1 Needs for RPC-based Distributed Training

Parameter Server Architecture

Implement single, centralized server for storing parameters and process requests from multiple trainers in cluster.

1. a server/set of servers centrally stores parameters, such as large embedding tables,
2. and several trainers query the parameter servers in order to retrieve the most up to date parameters
3. and the parameters are synced and updated with gradients from these trainers.

1.2. Advantage of RPC over DDP/FSDP

Suppose a model has very large embedding layer.

For such large embedding lookup, Data/Model parallelism is not efficient,
as entire embedding (although sharded) must be floating all around the cluster.

In PS arch, communication can be minimized by

1. doing parameter lookup in PS first
2. and then only transferring lookup result.

NOTE : But they are not Mutually exclusive;
can be combined together.

1.3. What is torch.distributed.rpc package

In torch.distributed.rpc package,

1. RPC

   for initializing/managing RPC server

1. RRef (Remote Reference)

   for referencing remote value/components as-is on local process

1. Distributed Autograd

   for autograd engine extended for distributed environment

1. Distributed Optimizer

   for optimizers working on distributed environment

1.4. CUDA RPC

Since v1.8, RPC allows users to configure a per-process global device map using the set_device_map API on CUDA,

specifying how to map local devices to remote devices directly.

This removes unnecessary

1. synchronizations
2. and D2H and H2D copies.

For example, on worker0's device map, "worker1" : {"cuda:0" : "cuda:1"}.

The response of an RPC will use the inverse of the caller device map.

1.4. CUDA RPC

Specifically, PyTorch RPC

1. extracts all Tensors from each request or response into a list
2. and packs everything else into a binary payload.
3. Then, TensorPipe will automatically choose a communication channel for each Tensor based on
    1. Tensor device type
    2. and channel availability

    on both the caller and the callee.

1.4. CUDA RPC

PyTorch RPC relies on TensorPipe as the communication backend.

TensorPipe abstracts initializing/managing channels for data transportation upon various channel backends.

Existing TensorPipe channels cover

1. NVLink,
2. InfiniBand,
3. SHM (Linux Shared Memory),
4. CMA (Linux Contiguous Memory Allocator),
5. TCP, etc.

2. Remote Reference Protocol

1. What is RRef
2. Assumptions of RRef Protocol
3. RRef Lifetime Guarantees

2.1. What is RRef

RRef is a reference of an object located on either

1. the local
2. or remote worker,

and can be considered as a distributed shared pointer.

Applications can create an RRef by calling remote()

and Every RRef can be uniquely identified by a global RRefId, assigned at the time of creation on the caller of the remote().

2.1. What is RRef

On the owner worker,
there is only one OwnerRRef instance, which contains the real data.

On user workers,
there can be as many UserRRefs as necessary, referencing OwnerRRef
i.e. UserRRef does not hold the data.

An OwnerRRef (and its data) will be deleted when

1. there is no UserRRef instances globally
2. and there are no reference to the OwnerRRef on the owner as well.

2.2. Assumptions of RRef Protocol

RRef protocol is designed with the following assumptions:

1. Transient Network Failures
2. Non-idempotent UDFs
3. Out of Order Message Delivery

2.2. Assumptions of RRef Protocol

Assumption 1 : Transient Network Failures

The RRef design handles transient network failures by retrying messages,

but not handle

1. node crashes
2. or permanent network partitions.

When those incidents occur, the application should

1. take down all workers,
2. revert to the previous checkpoint,
3. and resume training.

2.2. Assumptions of RRef Protocol

Assumption 2: Non-idempotent UDFs

User functions (UDF), which use UserRRefs,

1. are not idempotent
2. and therefore cannot be retried.

However, internal RRef control messages are

1. idempotent
2. and retried upon message failure.

2.2. Assumptions of RRef Protocol

Assumption 3: Out of Order Message Delivery

Protocol doesn’t assume message delivery order between any pair of nodes,

because both sender and receiver are using multiple threads.

Thus, there is no guarantee on which message will be processed first.

2.3. RRef Lifetime

The goal of the protocol is to delete an OwnerRRef at an appropriate time.

i.e.

1. there are no living UserRRef instances
2. and user code is not holding references to the OwnerRRef either.

The tricky part is to determine
if there are any living UserRRef instances.

2.3. RRef Lifetime Guarantees

Two types of guarantees must be ensured for RRef Protocol:

1. The owner will be notified when any UserRRef is deleted (G1).
2. Parent RRef will NOT be deleted until the child RRef is confirmed by the owner (G2).

2.3. RRef Lifetime Guarantees

G1 : The owner will be notified when any UserRRef is deleted.

As messages might come

1. delayed
2. or out-of-order,

we need one more guarantee to make sure
the delete message is not processed too soon.

2.3. RRef Lifetime Guarantees

G2 : Parent RRef will NOT be deleted until the child RRef is confirmed by the owner.

G2 trivially guarantees that no parent UserRRef can be deleted

before the owner knows all of its children UserRRef instances.

2.3. RRef Lifetime Guarantees

G2 : Parent RRef will NOT be deleted until the child RRef is confirmed by the owner.

However, it is possible that
the child UserRRef may be deleted before the owner knows its parent UserRRef.

Nevertheless, this does not cause any problem.

Because,

1. at least one of the child UserRRef’s ancestors will be alive
2. and it will prevent the owner from deleting the OwnerRRef.

3. Distributed Autograd Engine

1. Introduction
2. Distributed Autograd Engine on Forward Pass
3. Distributed Autograd Engine on Backward Pass

3.1. Introduction

```python
import torch
import torch.distributed.rpc as rpc

def my_add(t1, t2):
  return torch.add(t1, t2)

# On worker 0:
t1 = torch.rand((3, 3), requires_grad=True)
t2 = torch.rand((3, 3), requires_grad=True)

# Perform some computation remotely.
t3 = rpc.rpc_sync("worker1", my_add, args=(t1, t2))

# Perform some computation locally based on remote result.
t4 = torch.rand((3, 3), requires_grad=True)
t5 = torch.mul(t3, t4)

# Compute some loss.
loss = t5.sum()
```

For example, we can have

1. two nodes
2. and a very simple model (computation graph),

   which is

   1. partitioned across two nodes
   2. and implemented using torch.distributed.rpc:

3.1. Introduction

The main motivation behind distributed autograd is to

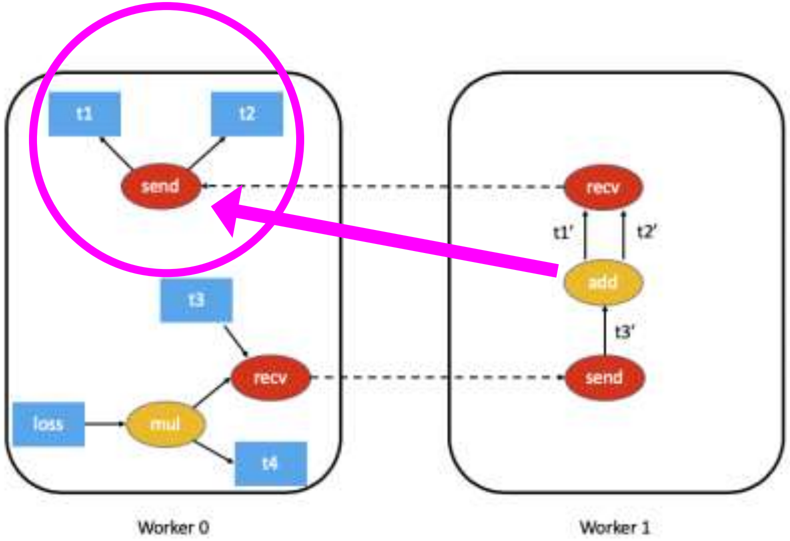
1. enable running a backward pass on such distributed models with the loss that we've computed
2. and record appropriate gradients for all tensors that require gradients.

3.2. Distributed Autograd Engine on Forward Pass

For distributed autograd,
we need to keep track of all RPCs during the forward pass for later backward pass.

To do so, we attach send and recv functions to the autograd graph
when we perform an RPC.

3.2. Distributed Autograd Engine on Forward Pass



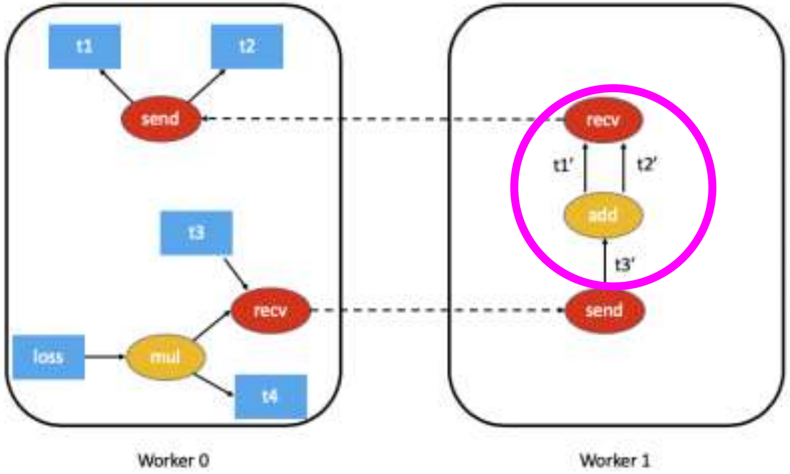
▲ t5.sum() excluded for simplicity

Procedures during forward pass - 1

The send function is attached to the source of the RPC i.e. callee of the function (Worker 0)

Also, output edges point to the autograd function for the input tensors of the RPC.

3.2. Distributed Autograd Engine on Forward Pass



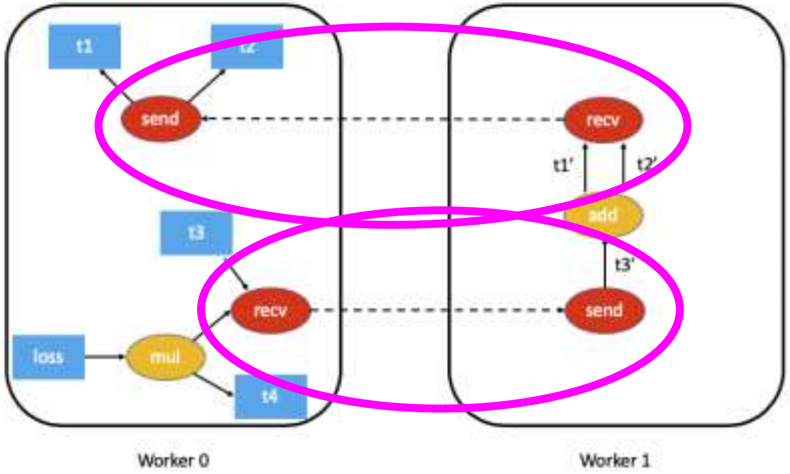
Procedures during forward pass - 2

The recv function is attached to the destination of the RPC

and its inputs are retrieved from operators executed on the destination using the input tensors.

i.e. t1' and t2' for add ops.

3.2. Distributed Autograd Engine on Forward Pass



Procedures during forward pass - 3

Each send-recv pair is assigned a globally unique autograd_message_id to uniquely identify the pair.

This helps lookup of the corresponding function during backward pass.

This is done using torch.distributed.rpc.RRef.to_here().

3.3. Distributed Autograd Engine on Backward Pass

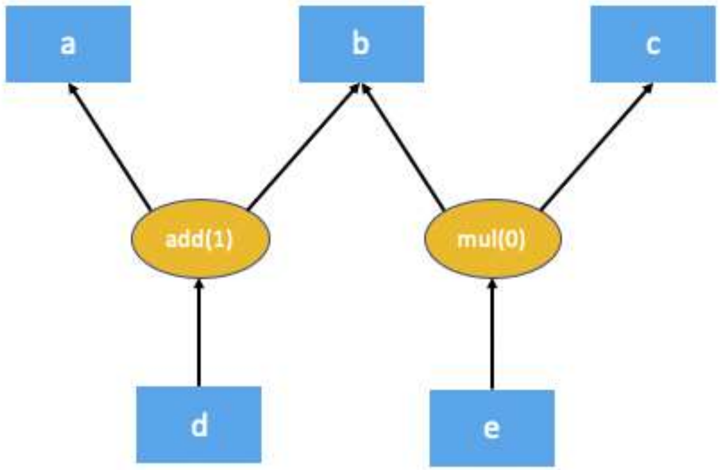
During backward pass, Distributed Autograd Engine goes through two steps:

1. Computing dependencies in the autograd graph,
   to help the engine know when a node in the graph is ready for backward execution.

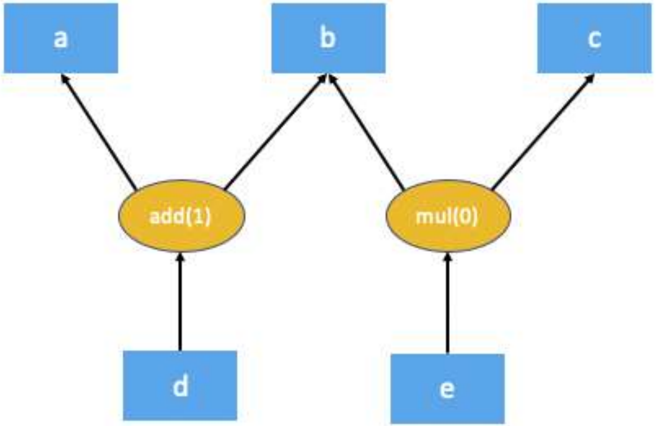
2. Applying FAST mode algorithm,
   to compute gradients.

3.3.1. Computing dependencies

```
import torch
a = torch.rand((3, 3), requires_grad=True)
b = torch.rand((3, 3), requires_grad=True)
c = torch.rand((3, 3), requires_grad=True)
d = a + b
e = b * c
d.sum.().backward()
```



3.3.1. Computing dependencies



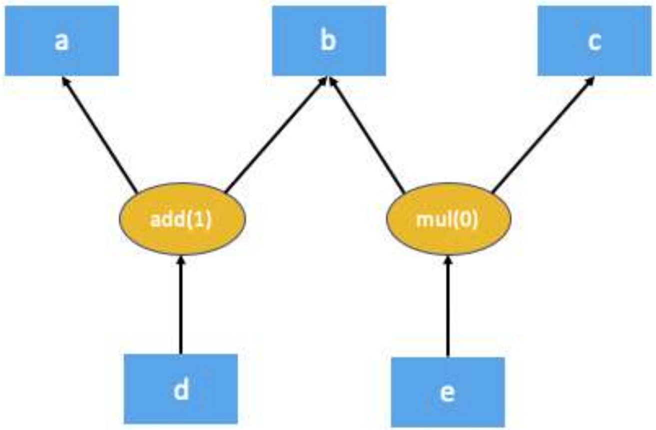
The numbers in brackets denote the number of dependencies

According to the graph,

1. the add node needs 1 input (= b)
2. and the mul node doesn't need any inputs;

i.e. the mul doesn't need to be executed for autograd to compute d.sum().backward().

3.3.1. Computing dependencies



The problem here is that certain nodes in the autograd graph might not be executed in the backward pass;

i.e. Due to overhead from distributed nature,

for example, while d is depending on e,
e might not be executed at the time d requires e

i.e. while traversing the graph

3.3.2. FAST mode algorithm

Much of such overhead can be avoided by assuming
every send and recv function are valid as part of the backward pass

c.f. most applications don’t perform RPCs that aren’t used.

This simplifies the distributed autograd algorithm and is much more efficient,
at the cost that few limitations.

This algorithm is called the FAST mode algorithm.

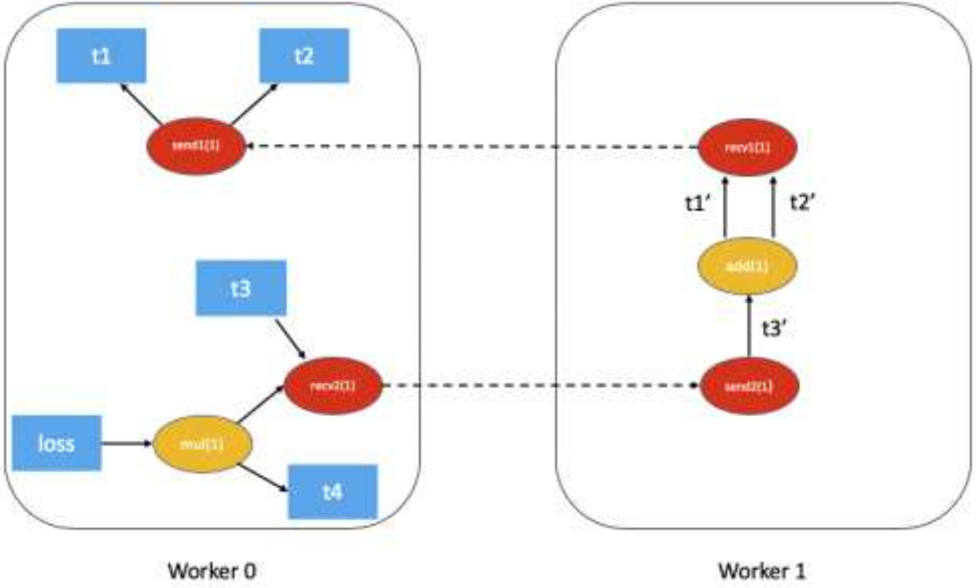
3.3.2. FAST mode algorithm

The key assumption is that
each send function has a dependency of 1 when we run a backward pass.

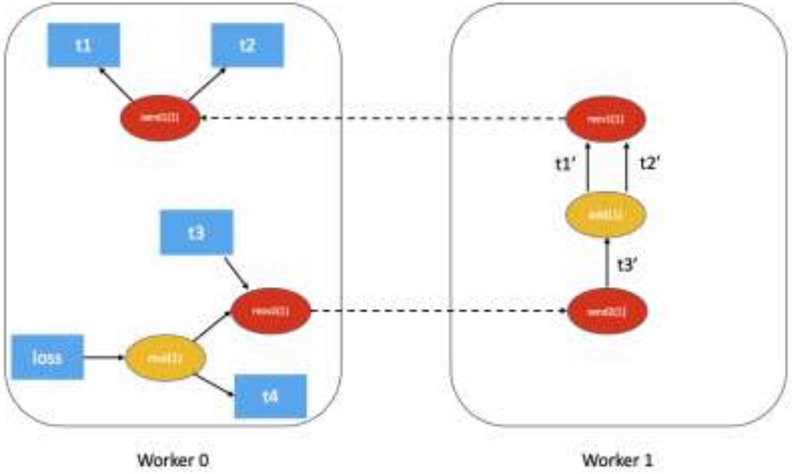
i.e. assume that we’ll receive a gradient over RPC from another node.

3.3.2. FAST mode algorithm

```python
def my_add(t1, t2):
  return torch.add(t1, t2)

# On worker 0:

# Setup the autograd context. Computations that take
# part in the distributed backward pass must be within
# the distributed autograd context manager.
with dist_autograd.context() as context_id:
  t1 = torch.rand((3, 3), requires_grad=True)
  t2 = torch.rand((3, 3), requires_grad=True)

  # Perform some computation remotely.
  t3 = rpc.rpc_sync("worker1", my_add, args=(t1, t2))

  # Perform some computation locally based on remote result.
  t4 = torch.rand((3, 3), requires_grad=True)
  t5 = torch.mul(t3, t4)

  # Compute some loss.
  loss = t5.sum()

  # Run the backward pass.
  dist_autograd.backward(context_id, [loss])

  # Retrieve the gradients from the context.
  dist_autograd.get_gradients(context_id)
```



3.3.2. FAST mode algorithm



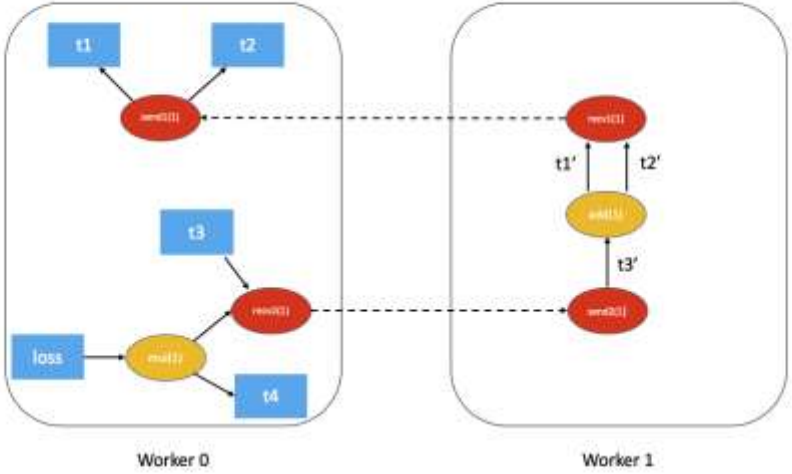
FAST mode algorithm - 1

Starting from the worker with the roots for the backward pass
- all roots MUST be local -

lookup all the send functions for the current distributed autograd context.

= On Woker 0, loss and send1

3.3.2. FAST mode algorithm

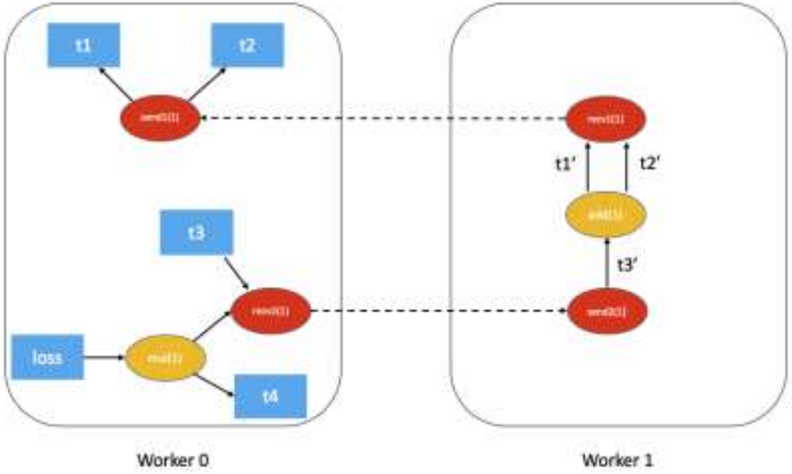


FAST mode algorithm - 2

Compute dependencies locally, starting from

1. the provided roots
2. and all the send functions

3.3.2. FAST mode algorithm

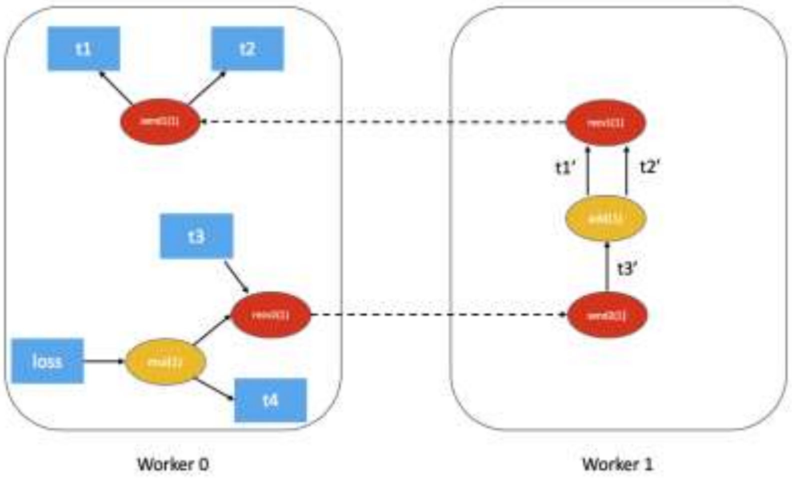


FAST mode algorithm - 2

=

1. first execute the mul function,
2. accumulate its output in the autograd context as the gradient for t4.

3.3.2. FAST mode algorithm



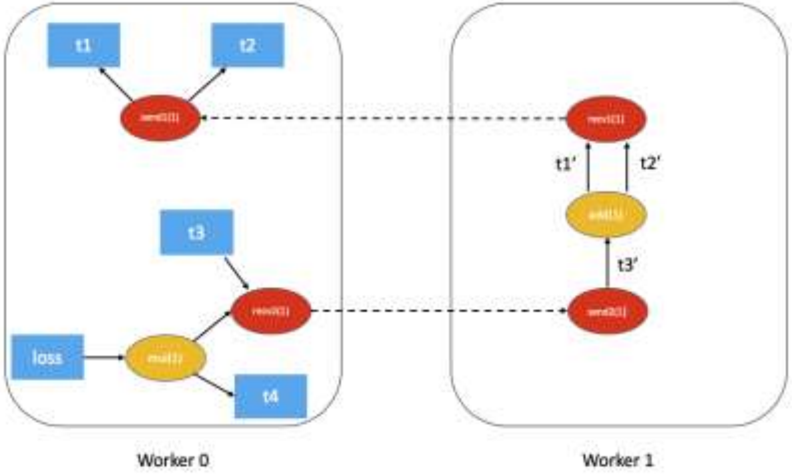
FAST mode algorithm - 3

When the autograd engine executes the recv function,

the recv function sends the input gradients via RPC to the appropriate worker.

= Execute recv2 which sends the gradients to Worker 1.

3.3.2. FAST mode algorithm

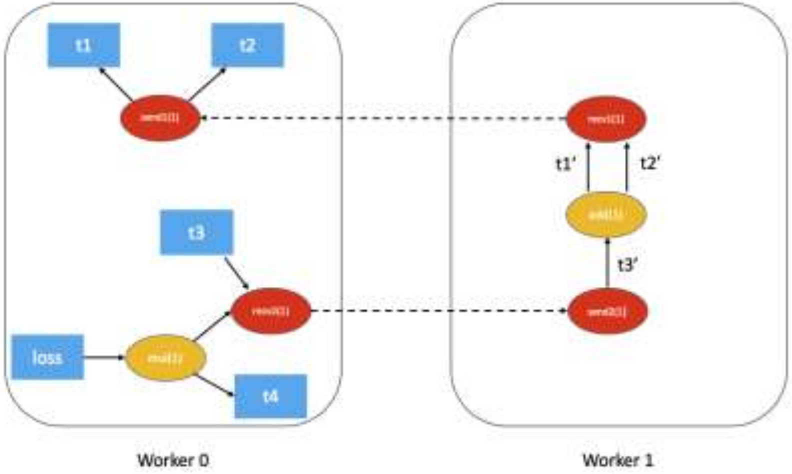


FAST mode algorithm - 4

In 3, the recv function also sends autograd_context_id and autograd_message_id to the remote host.

Using these, lookup the appropriate send function.

3.3.2. FAST mode algorithm

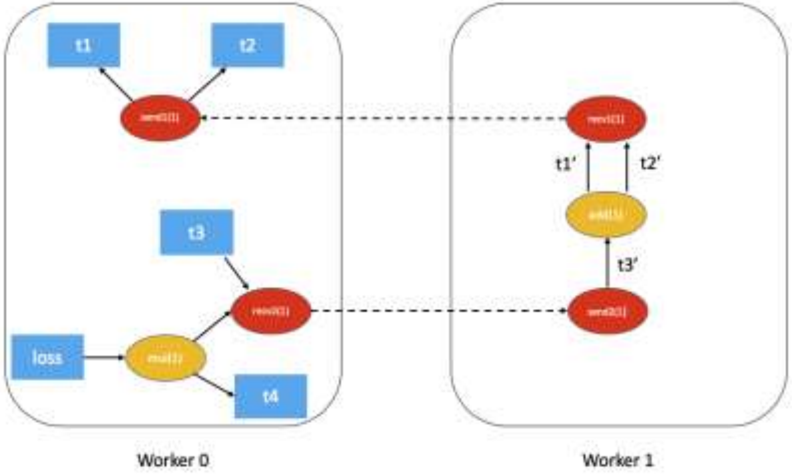


FAST mode algorithm - 5

If this is the first time a worker has received a request for the given autograd_context_id,

it will compute dependencies locally as described in points 1-3 above.

3.3.2. FAST mode algorithm

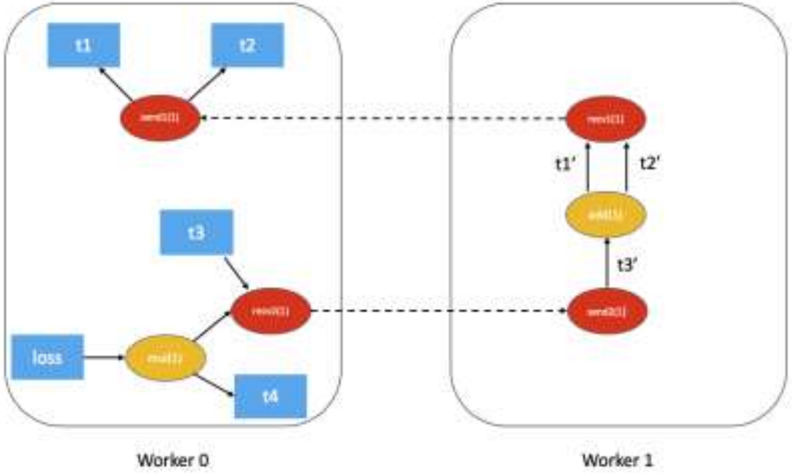


FAST mode algorithm - 5

= Since this is the first time Worker 1 has heard about this backward pass,

1. it starts dependency computation
2. and marks the dependencies
   for send2, add and recv1 appropriately.

3.3.2. FAST mode algorithm

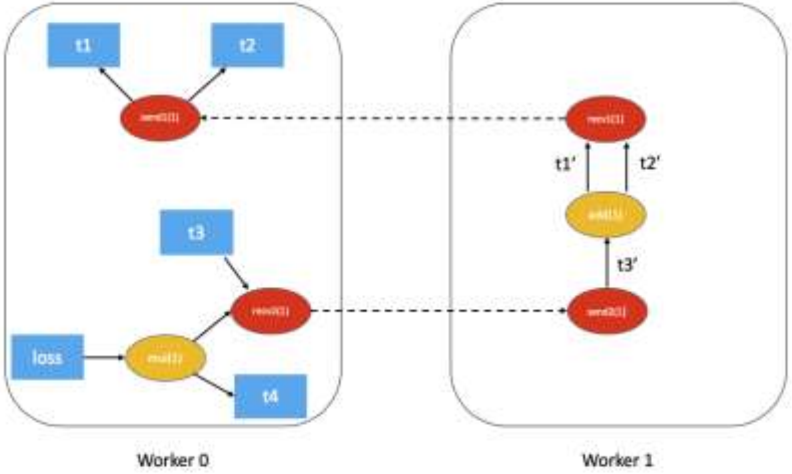


FAST mode algorithm - 6

The send function retrieved from 5 is then enqueued for execution on the local autograd engine for that worker.

= Enqueue send2 on the local autograd engine of Worker 1,
which in turn executes add and recv1.

3.3.2. FAST mode algorithm



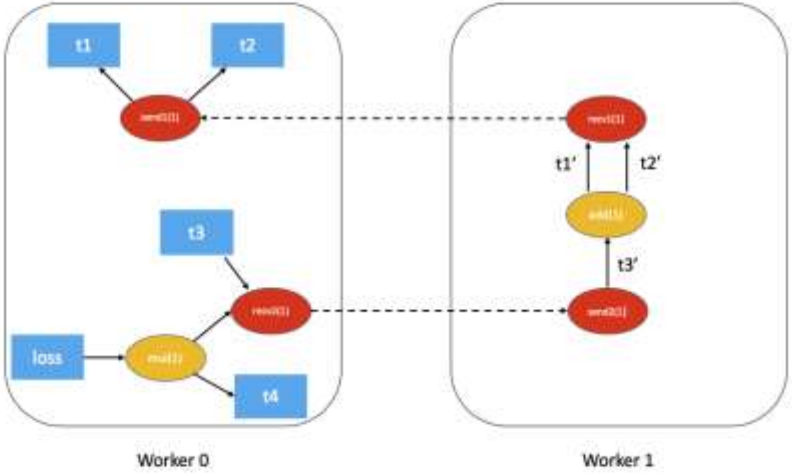
FAST mode algorithm - 6

= When recv1 is executed, it sends the gradients over to Worker 0.

Since Worker 0 has already computed dependencies for this backward pass,

1. it just enqueues
2. and executes send1 locally.

3.3.2. FAST mode algorithm



FAST mode algorithm - 7

Finally, instead of accumulating the gradients on the .grad field of the Tensor,

we accumulate the gradients separately per Distributed Autograd Context.

= Finally, gradients are accumulated for t1, t2 and t4 in the Distributed Autograd Context.

3.3.2. FAST mode algorithm

In the general case, it might not be necessary that
every send and recv function is valid as part of the backward pass.

To address this, SMART mode algorithm is proposed.

But currently, only the FAST mode algorithm is implemented.