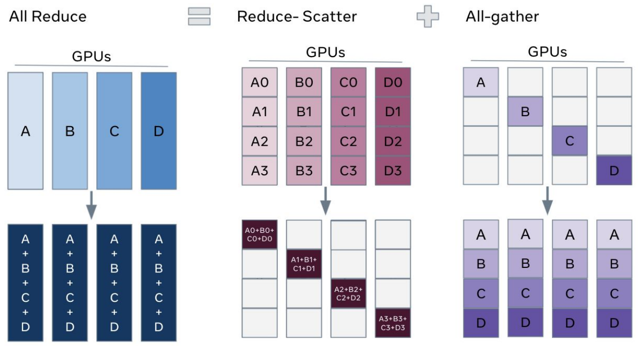
PyTorch Distributed and Parallel Training and FSDP

경희대학교
박현우

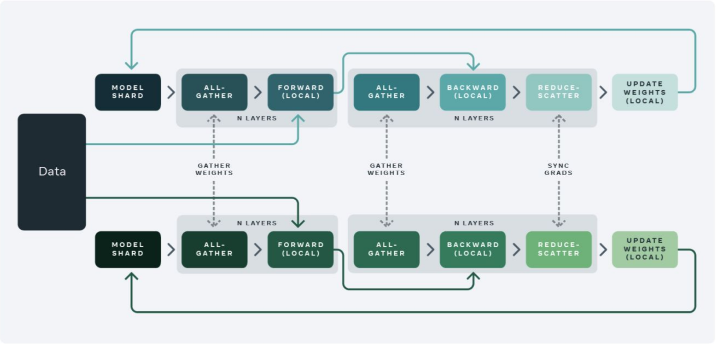
Table of Contents

Pytorch Distributed Training Basic

1. Overview of Pytorch Distributed Training
2. Collective Communications
3. Distributed Data Parallel (DDP)



Fully sharded data parallel training

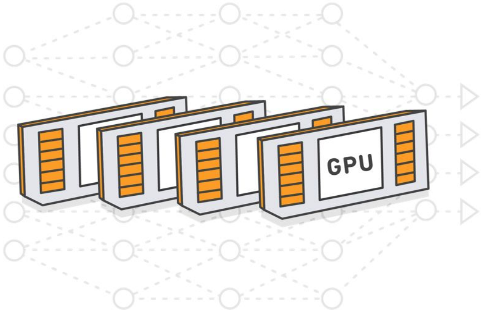


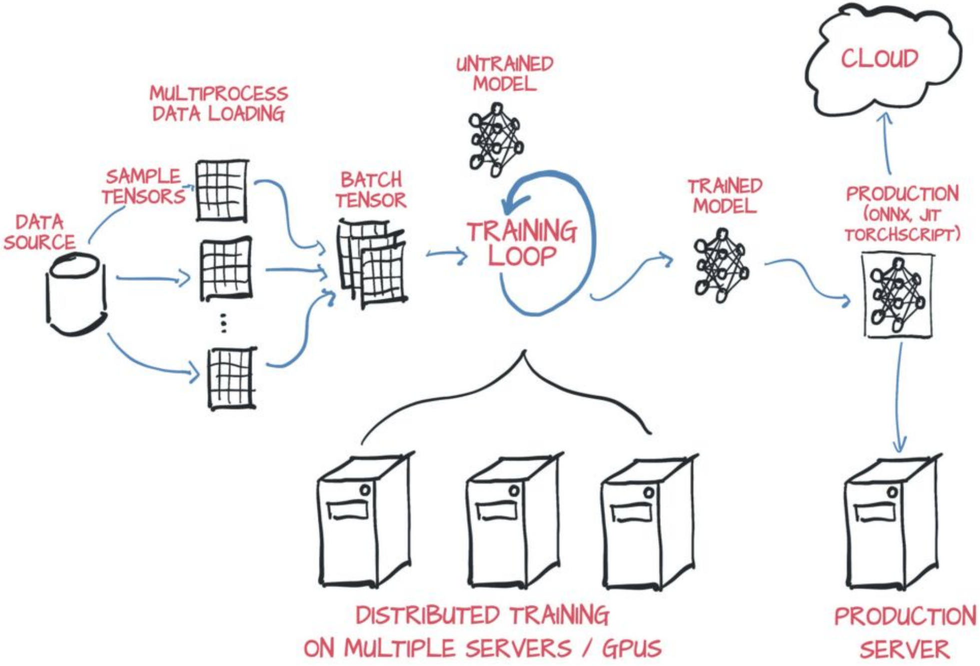
FSDP

1. What is FSDP
2. How FSDP operates

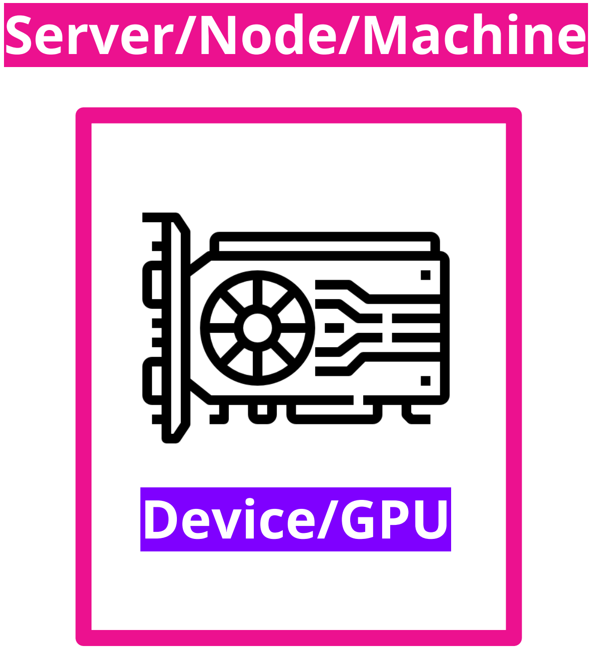
1. Overview of Pytorch Distributed Training

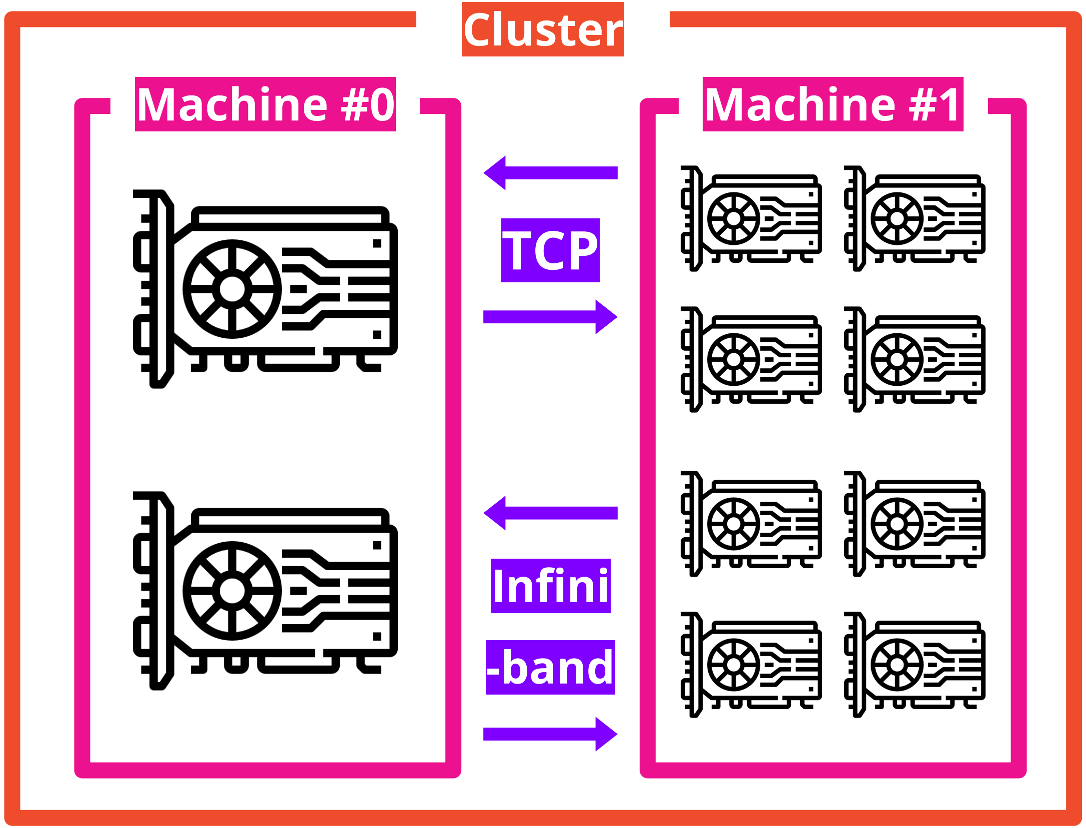
PYTORCH



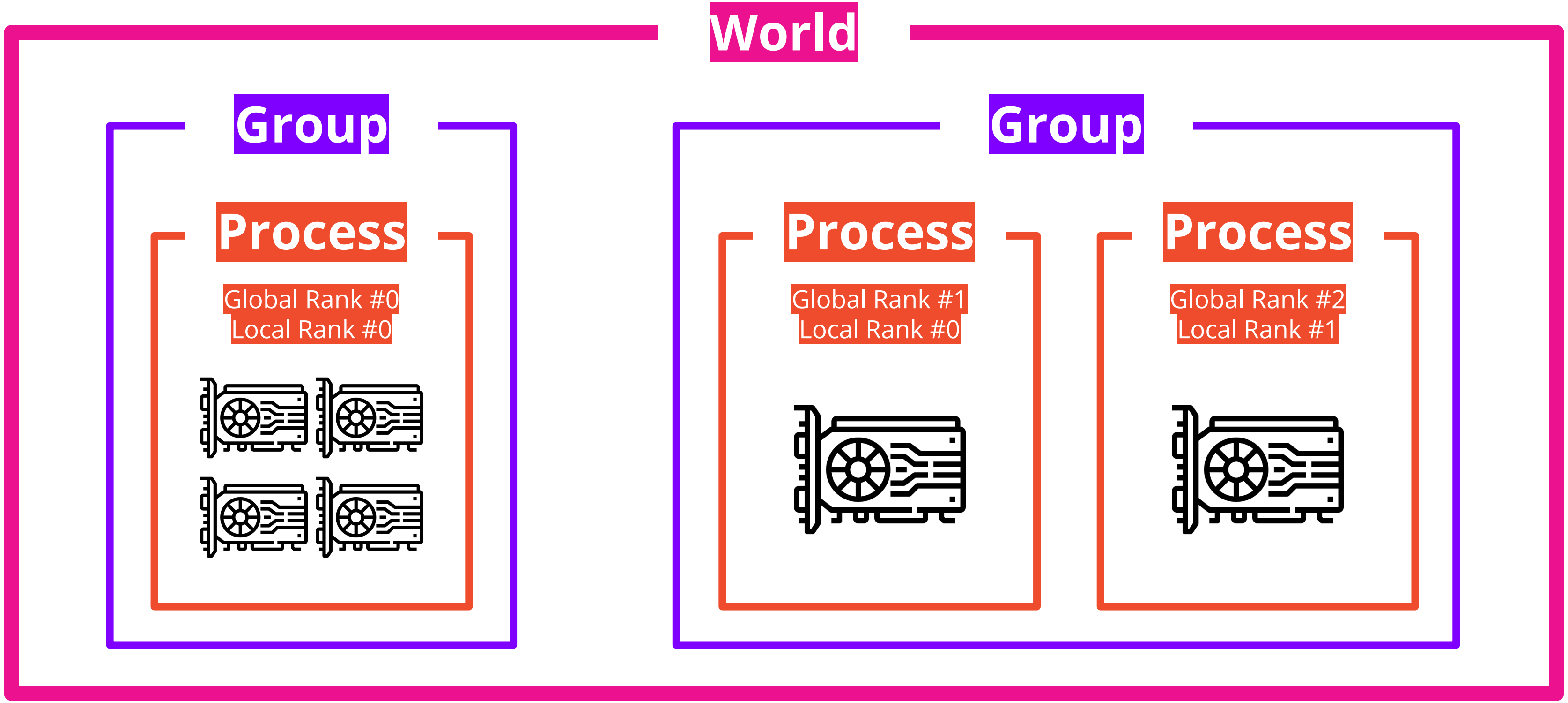


Terminology



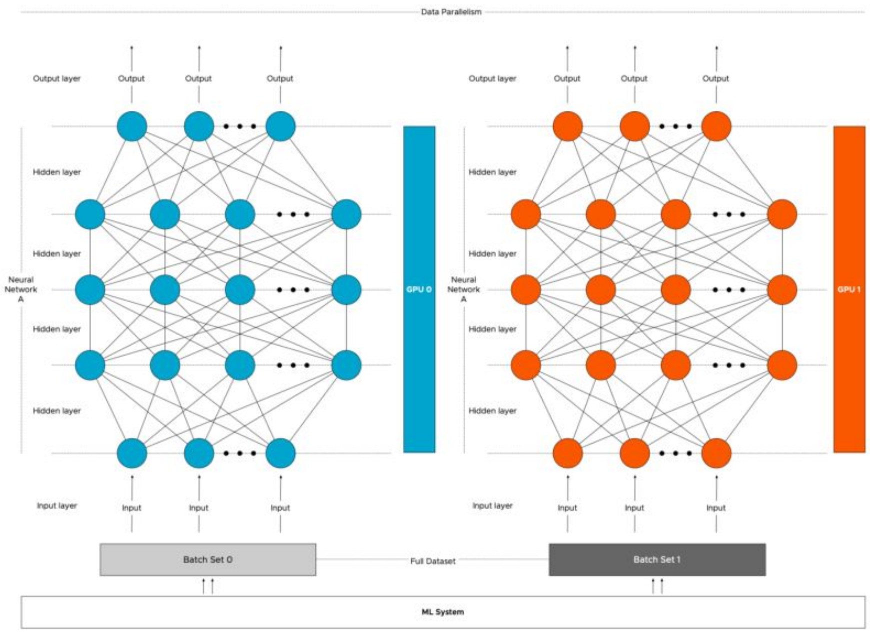


Terminology

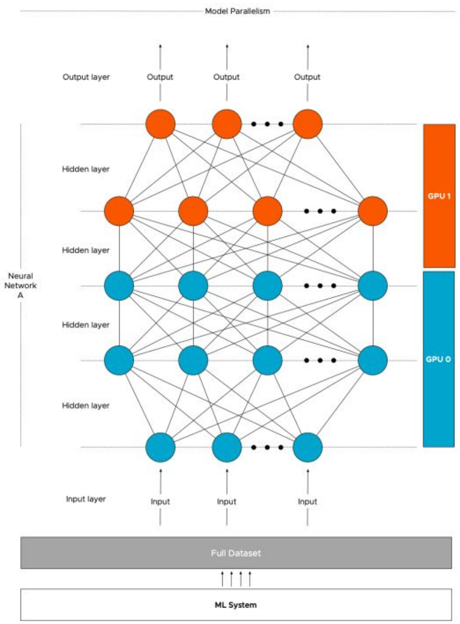


Terminology

Data Parallelism



Model Parallelism



1. torch.distributed

Low Level **High Level**

Components **Modules** **Paradigms**

1.1 Main components of torch.distributed

DDP

Distributed Data-Parallel Training

RPC

Distributed RPC Framework

c10d

Communication Core Library

1.1 Main components of torch.distributed

DDP

Aligned for single-program multiple-data training paradigm.

Specifically:

1. Take care of gradient communication to keep model replicas synchronized
2. Overlap communication with the gradient computations to speed up training.

1.1 Main components of torch.distributed

RPC

Aligned for supporting general training structures beyond data-parallel training.

For example,

1. Distributed Pipeline Parallelism
2. Parameter Server Paradigm
3. Combinations of DDP with other training paradigms

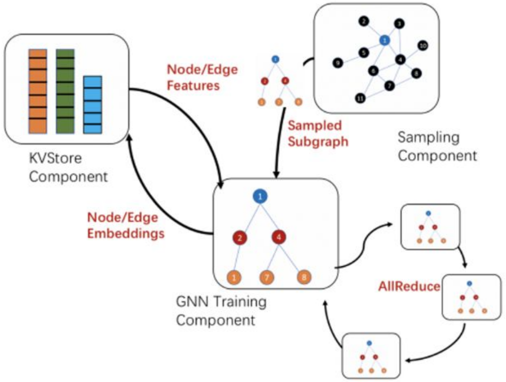


Fig. 2: DistDGL's logical components.

1.1 Main components of torch.distributed

RPC

Specifically:

1. Help manage remote object lifetime (RRef)
2. Extend the autograd engine to Distributed Autograd Engine.

Internally, RPC Backend of PyTorch relies on TensorPipe,

which is an implementation of communications like NVLink or TCP, specially for PyTorch tensor.

1.1 Main components of torch.distributed

c10d

Aligned for sending/receiving tensors across processes within a group.

c10d offers both

1. Collective Communication APIs

   (e.g., all_reduce and all_gather)

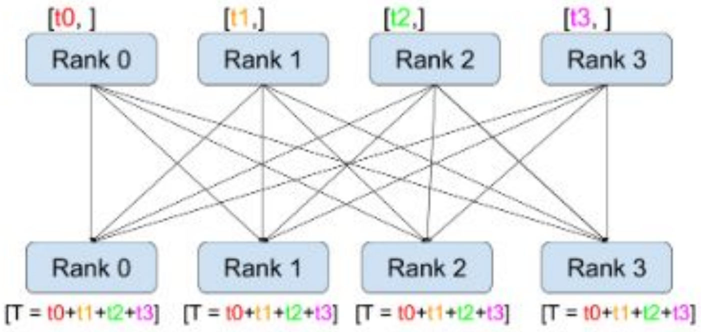
2. P2P communication APIs

   (e.g., send and isend).

1.1 Main components of torch.distributed

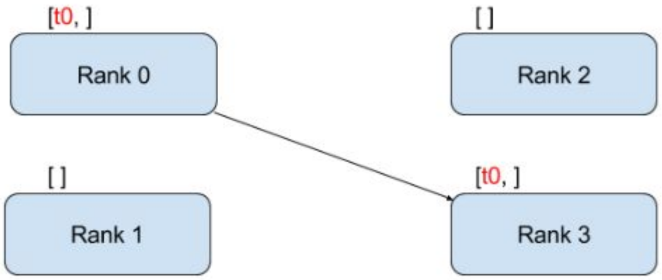
c10d

Collective Communication



All-Reduce

P2P Communication



1.2 Basic Modules for Data-Parallel training

<table>
<tr><th></th><th>Single Machine</th><th>Multiple Machine</th></tr>
<tr><td>Single GPU</td><td>Vanilla PyTorch</td><td rowspan="2">DDP + torchrun
(torch.distributed.elastic)</td></tr>
<tr><td>Multiple GPU</td><td>DataParallel (DP)
DistributedDataParallel (DDP)</td></tr>
</table>

1.2 Basic Modules for Data-Parallel training

Vanilla PyTorch

For simplest,
small-sized model.

DataParallel (DP)

For speed up training with
minimal code changes.

1.2 Basic Modules for Data-Parallel training

DistributedDataParallel (DDP)

For faster speed than DP,
but with more code.

DDP + torchrun (torch.distributed.elastic)

For scaling across cluster of machines; find-grained error handling and dynamic allocation/drop of machines during training.

1.2 Basic Modules for Data-Parallel training

In case of DataParallel (DP)

• WARNING

It is recommended to use `DistributedDataParallel`, instead of this class, to do multi-GPU training, even if there is only a single node. See: Use nn.parallel.DistributedDataParallel instead of multiprocessing or nn.DataParallel and Distributed Data Parallel.

1.3 Main Paradigms of Distributed/Parallel training

Data Parallel Trainings	FSDP	RPC-based Trainings
DataParallel Distributed DataParallel	FSDP	Distributed Pipeline Parallelism Parameter Server Architecture

1.3 Main Paradigms of Distributed/Parallel training

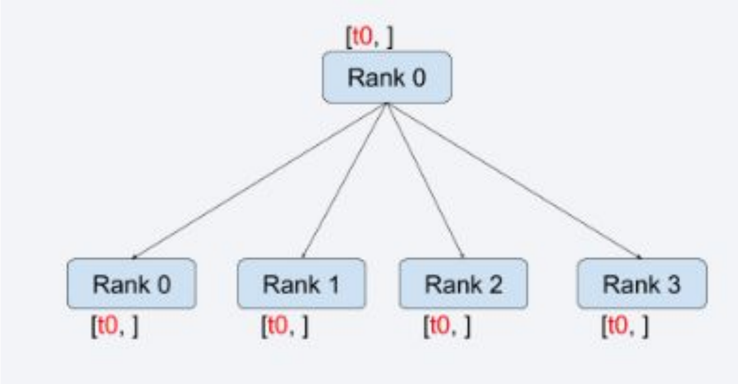
Simplicity

Scalability
Flexibility
Customizability

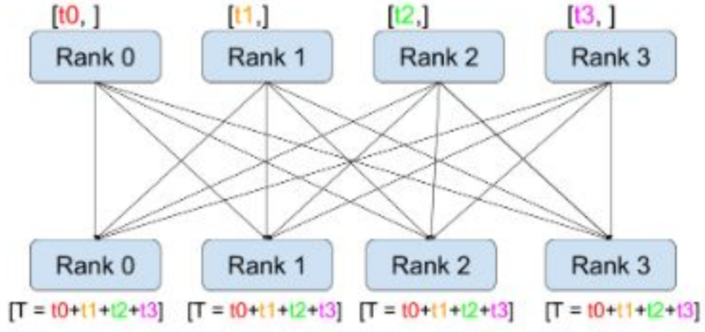
DataParallel	Distributed DataParallel	FSDP	RPC-based
Limited to Single Machine	Can extend to Multi Machines	Can extend to large Cluster	Fully Customizable

2. Collective Communications

Collective Communications is to communicate across multiple processes in a cluster.



Broadcast



All-Reduce

2. Collective Communications

Standard Communications

- Scatter
- Broadcast
- All-Reduce
- Gather
- Reduce
- All-Gather

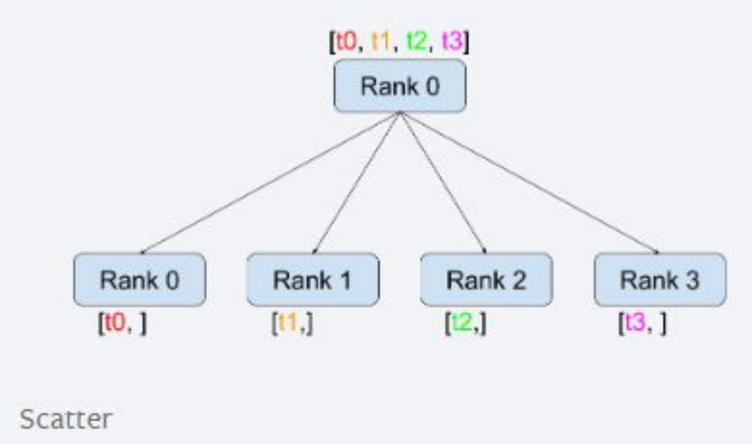
Special Communications

- Reduce-Scatter
- All-to-All
- Barrier

2. Collective Communications

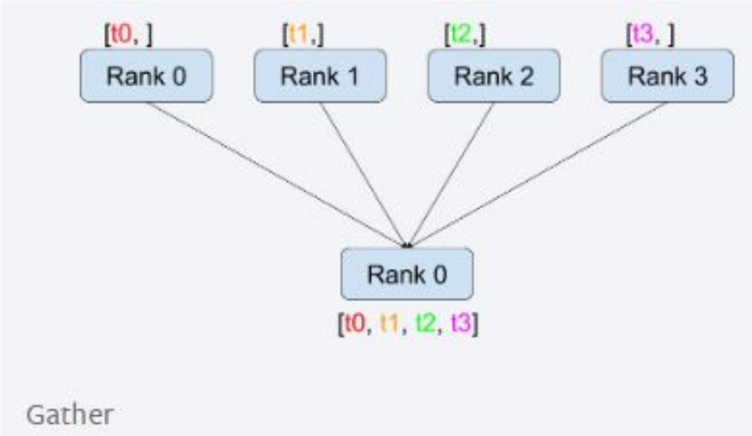
Scatter

Copies the *i*-th tensor to the *i*-th process.



Gather

Copies tensor from all processes in dst.



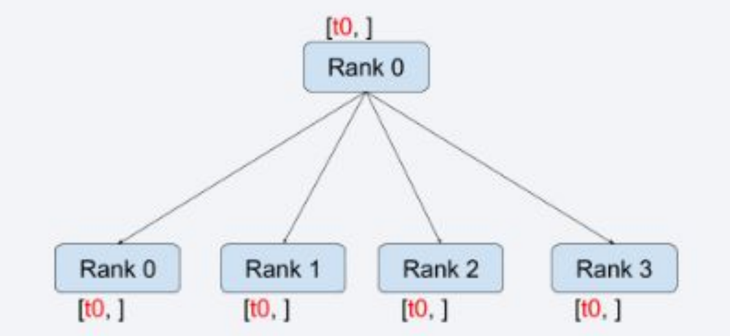
2. Collective Communications

Broadcast

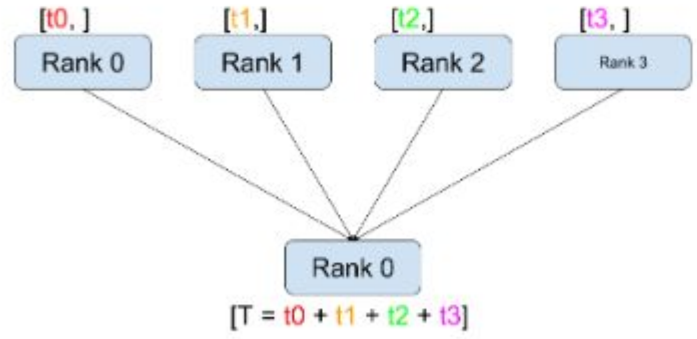
Copies tensor from source to all other processes.

Reduce

1. Applies operation to every tensor
2. Stores the result in destination.



Broadcast

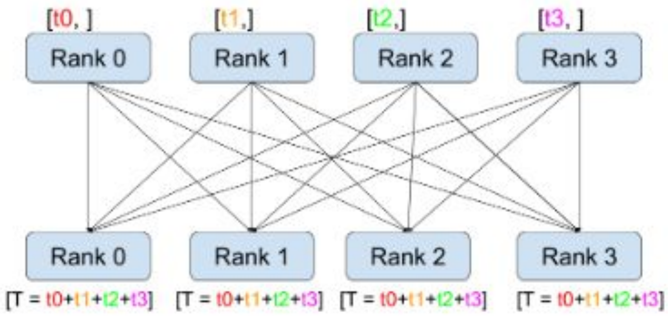


Reduce

2. Collective Communications

All-Reduce

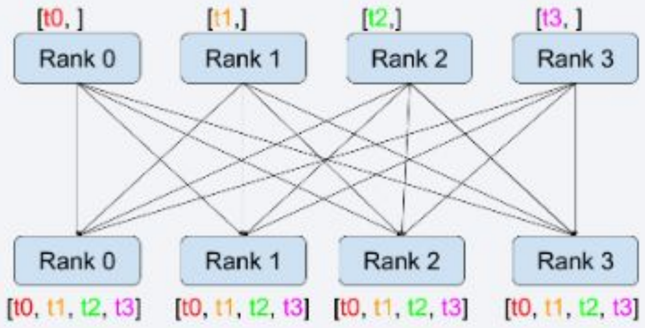
Same as Reduce, but the result is stored in all processes.



All-Reduce

All-Gather

Simply, All-Reduce without op.

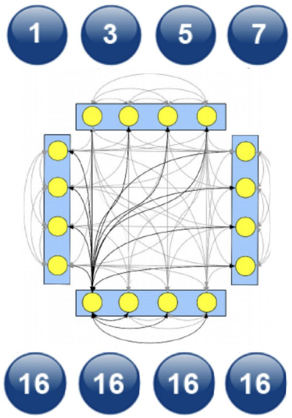


All-Gather

2.1. All-Reduce Implementation

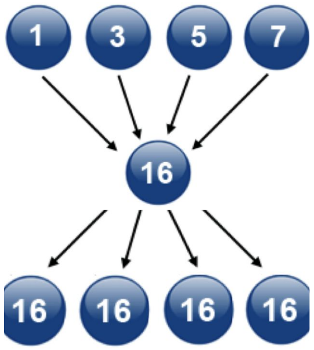
Approach #1

All processes are individually communicating to each other.



Approach #2

Aggregate on Master process and propagate (Reduce + Broadcast)



2.1. All-Reduce Implementation

Approach #1

Have O(N^2) communications

Approach #2

Impose severe load on master process

2.1. All-Reduce Implementation

As such, All-Reduce/All-Gather communications are Extremely expensive.

Thus, in order to efficiently distribute trainings,

Need to Decompose these operations and Distribute them in parallel.

3. Distributed Data Parallel (DDP)

DDP implements both Data and Model Parallelism across multiple machines.

DDP can be run on in both

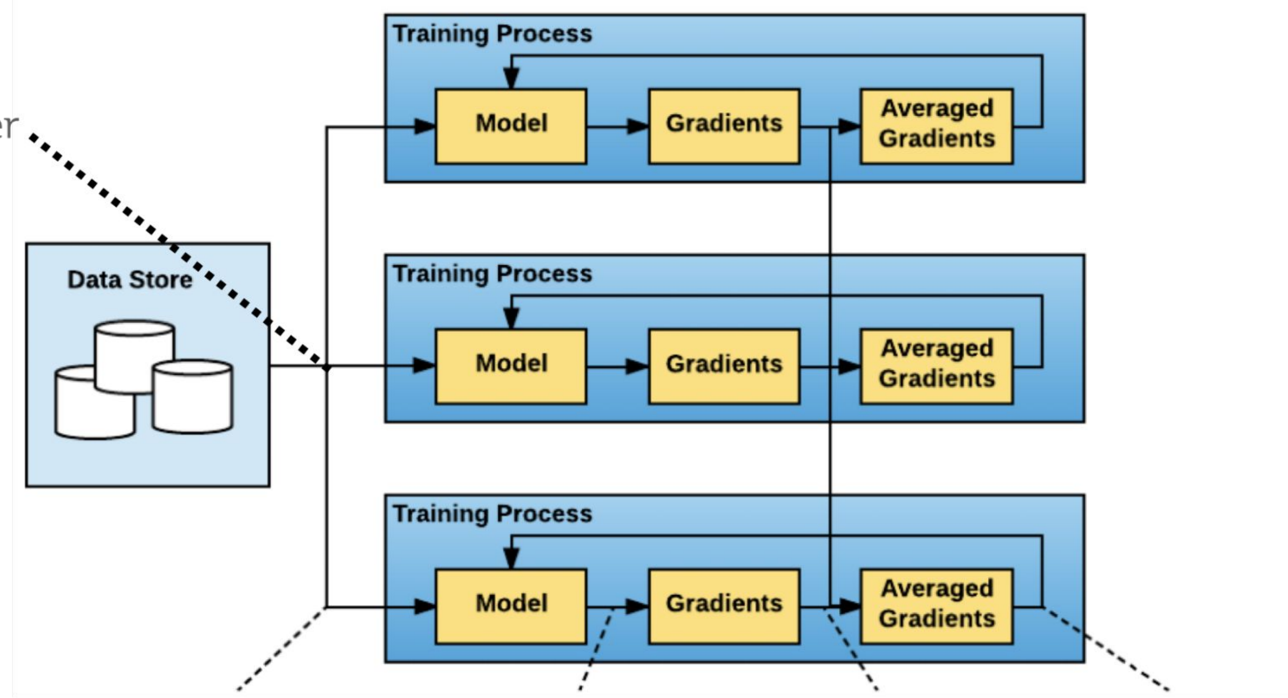
1. Single Machine - Multiple Devices
2. Multiple Machines - Multiple Devices

(in which for latter, torchrun is required.)

Also, DDP allows for Heterogeneous cluster (#).

3.1 Procedure of DDP

1. 데이터 Scatter



2. Forward 수행 3. Backward 수행 4. Gradient All-reduce 5. 파라미터 업데이트

3.1 Procedure of DDP

Specifically:

1. DDP registers an autograd hook for each parameter (in model.parameters())
2. The hook will fire when the corresponding gradient is computed in the backward pass.
3. Then DDP uses that signal to trigger gradient synchronization across processes.
4. Each process updates its own model with given (reduced) gradient.

3.2 Caveats of DDP

Skewed Processing Speed problem

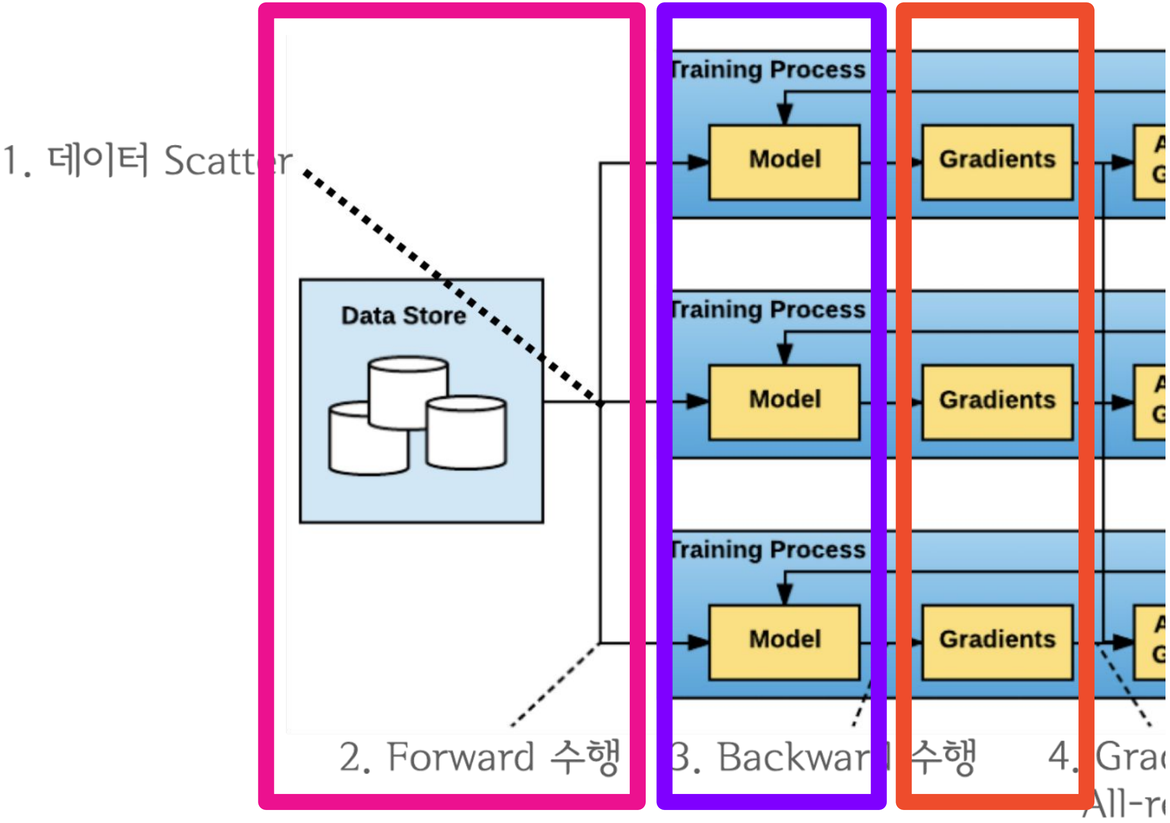
Inefficient Communication

3.2.1 Skewed processing speed problem

There are 3 distributed synchronization points in DDP:

1. **Constructor**
2. **Forward pass**
3. **Backward pass**

3.2.1 Skewed processing speed problem



Constructor

Forward Pass

Backward Pass

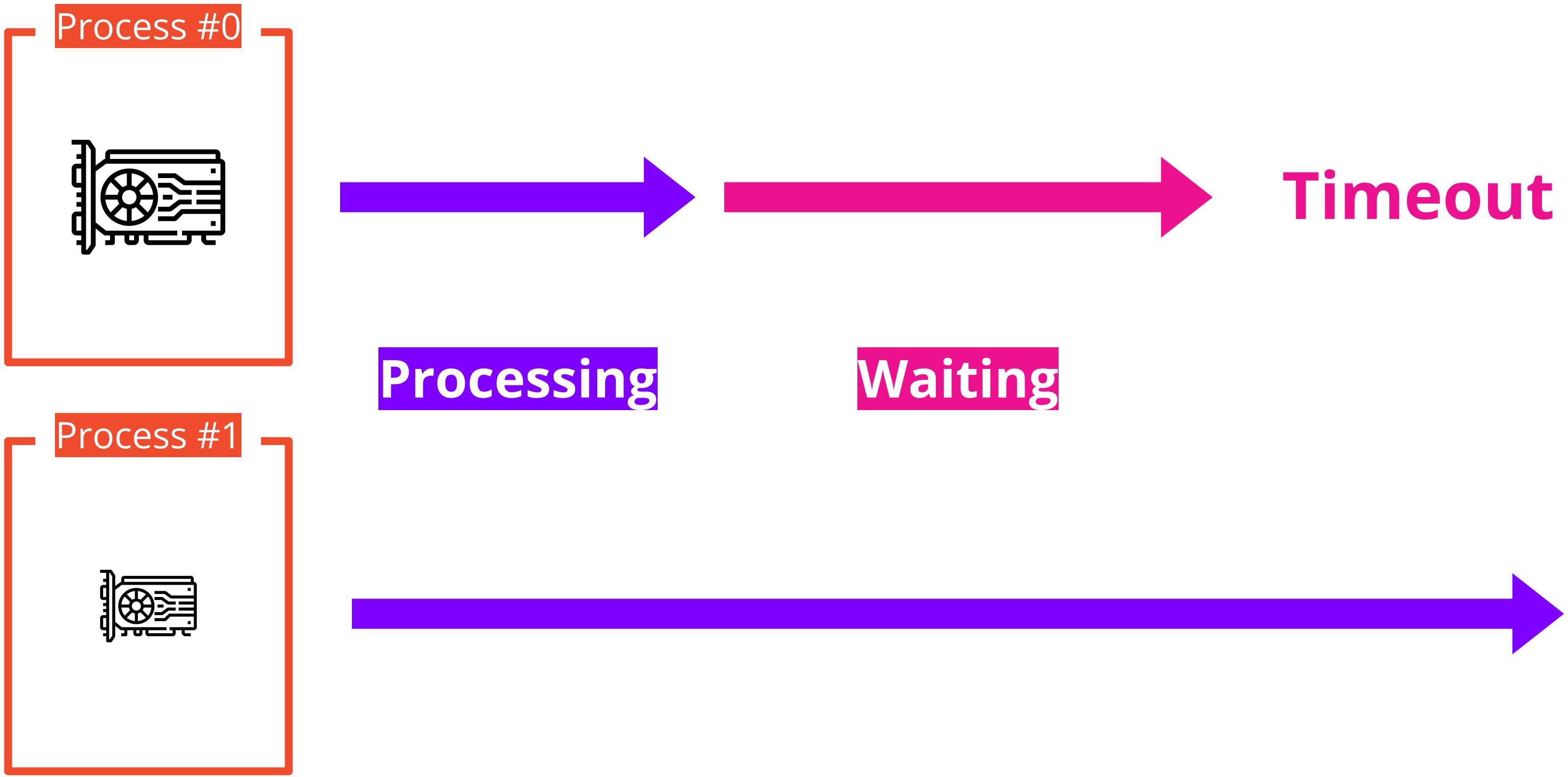
3.2.1 Skewed processing speed problem

In ideal, each process would

1. launch the same number of synchronizations
2. and reach these synchronization points in the same order.

However, in practice,
Desynchronization can be occurred.

3.2.1 Skewed processing speed problem



3.2.1 Skewed processing speed problem

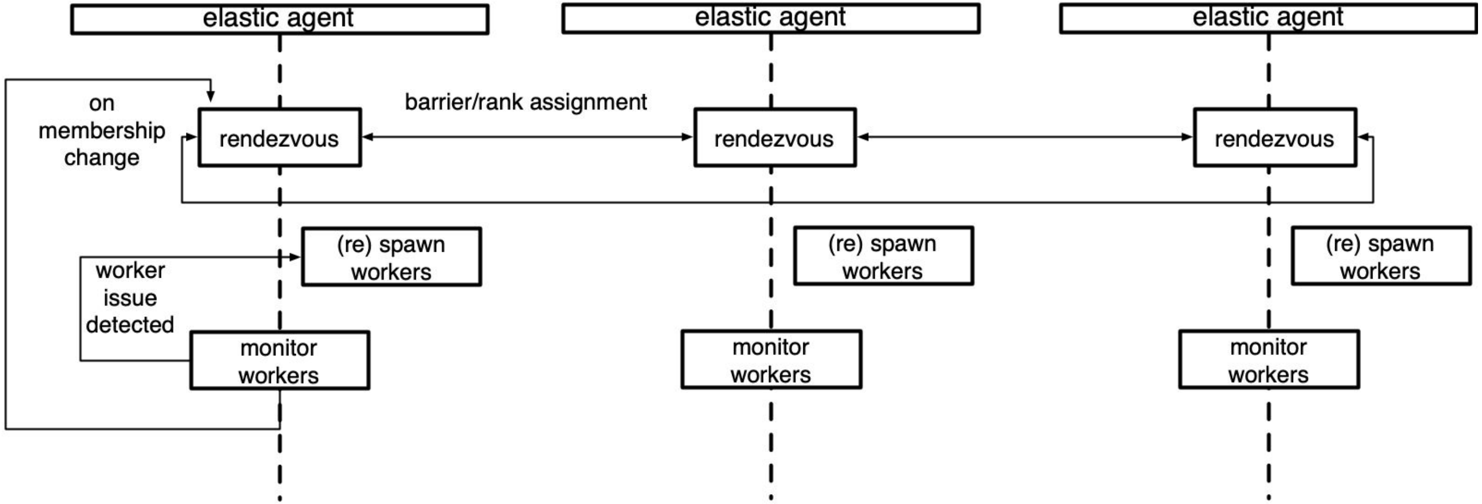
Such skewed processing speeds can also be occurred by

1. **Network delays**
2. **Resource contentions**
3. **Unpredictable workload spikes**

3.2.1 Skewed processing speed problem

DDP doesn't provide delicate, flexible synchronization policy across processes.

Unlike torch.distributed.elastic, DDP cannot recover from such timeout by itself.



3.2.1 Skewed processing speed problem

Thus, it is fully responsible for developers to deal with such synchronization.

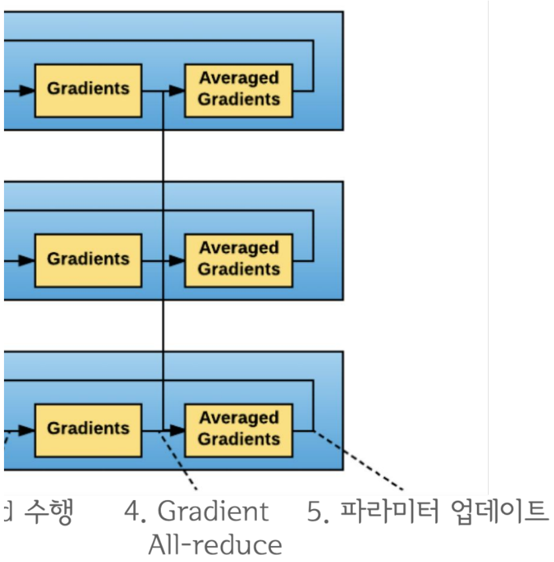
• WARNING

This module assumes all parameters are registered in the model of each distributed processes are in the same order. The module itself will conduct gradient `allreduce` following the reverse order of the registered parameters of the model. In other words, it is users' responsibility to ensure that each distributed process has the exact same model and thus the exact same parameter registration order.

3.2.2 Inefficient Communication

DDP depends on All-Reduce to synchronize Updated gradients,

which are very expensive operation.



3.3. Model Parallelism

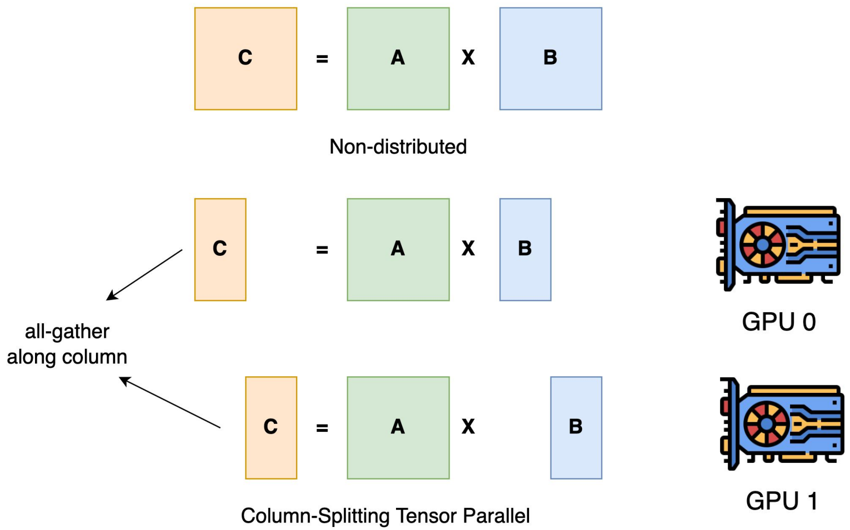
Types of Model Parallelism

Intra-layer model parallelism or Tensor Parallelism

Inter-layer model parallelism or Pipeline Parallelism

3.3. Model Parallelism in DDP

Intra-layer model parallelism

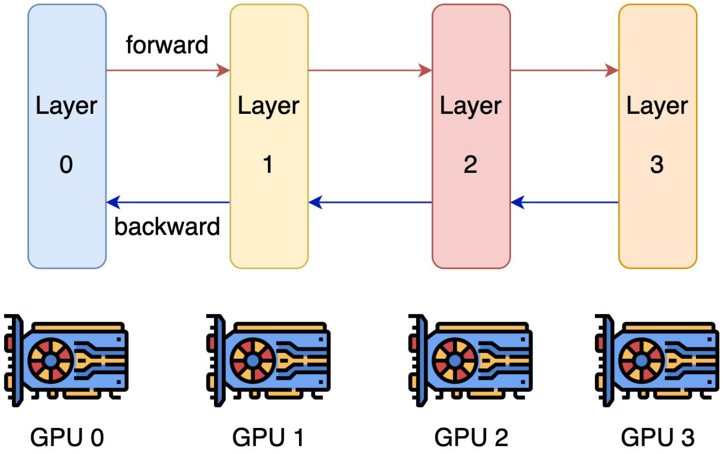


Vertically splitting the model

No Dependency among processes

3.3. Model Parallelism in DDP

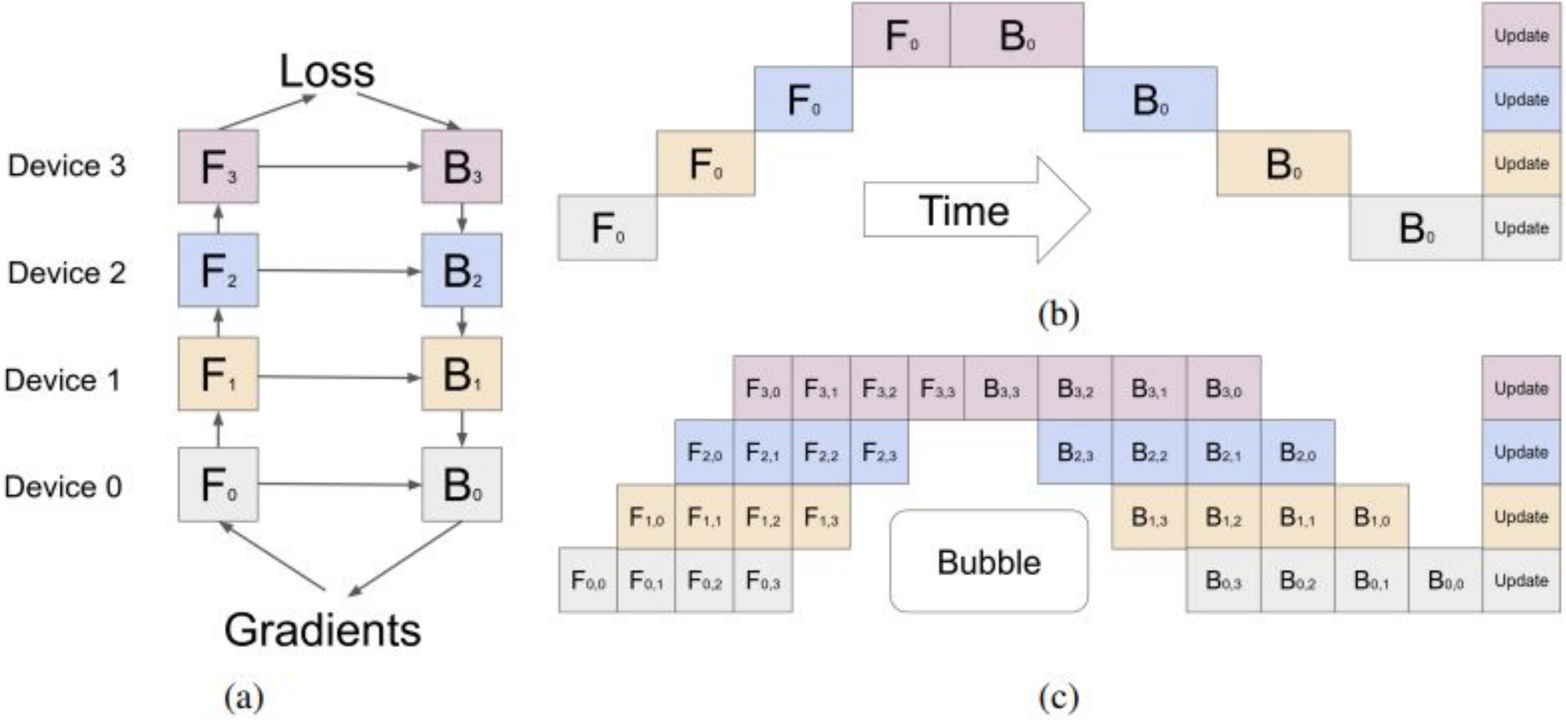
Inter-layer model parallelism



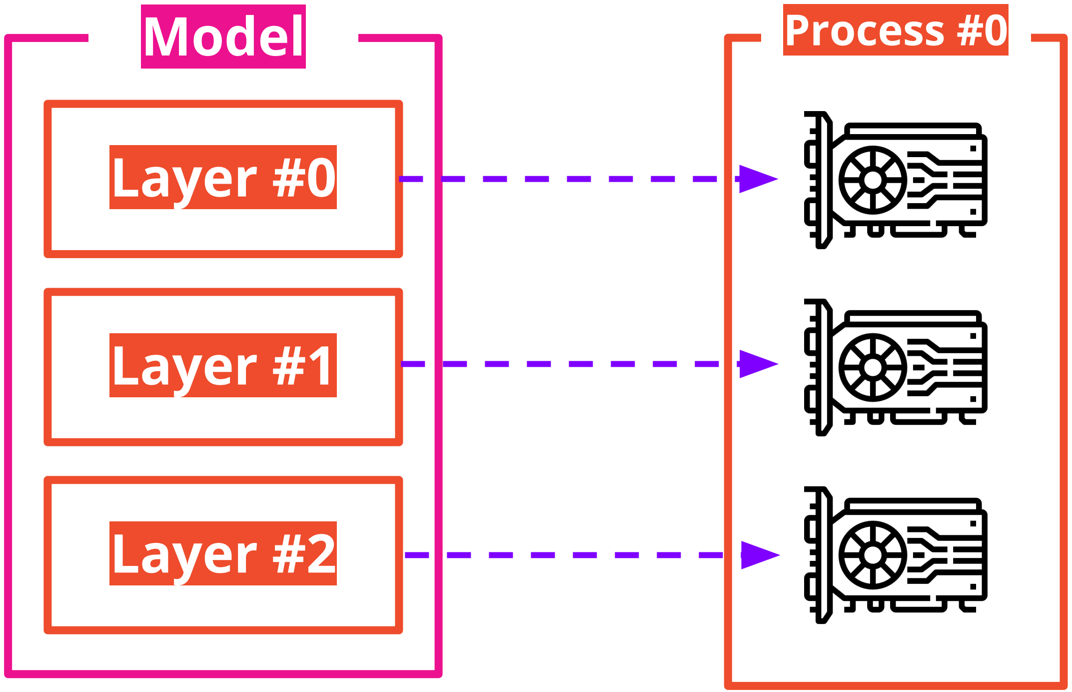
Horizontally splitting the model

Have Dependency among processes

3.3. Model Parallelism in DDP

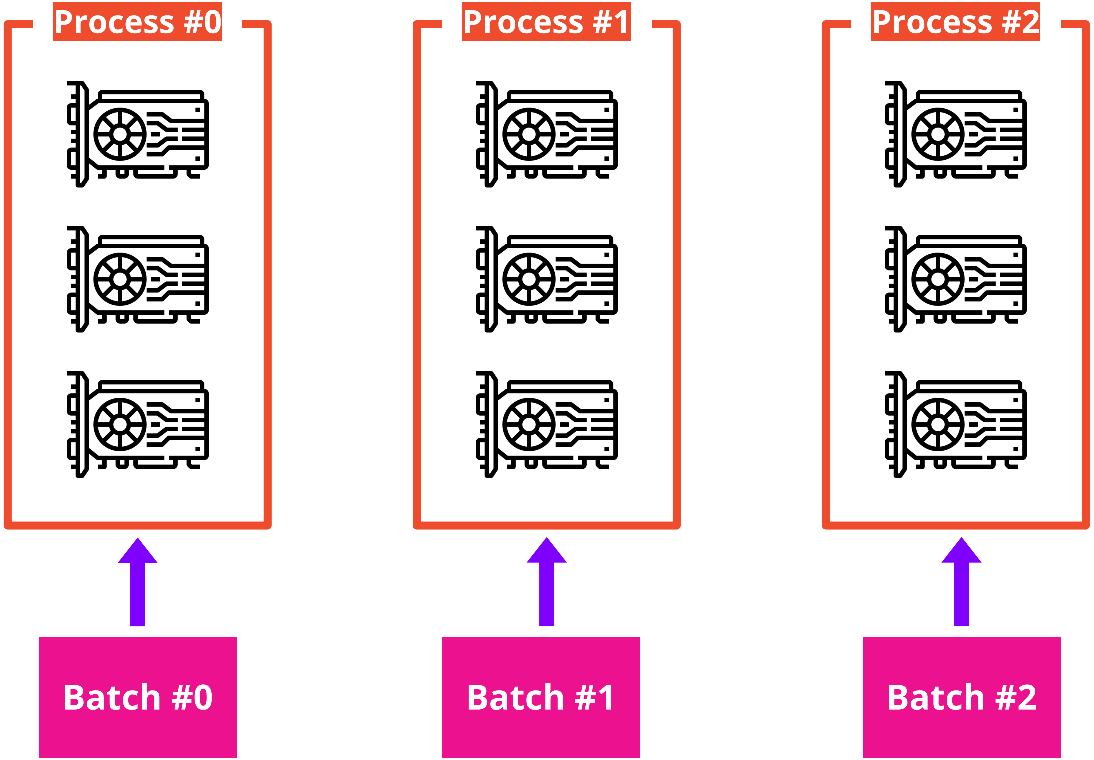


3.3. Model Parallelism in DDP



Inter-layer model parallelism

3.3. Model Parallelism in DDP



3.3. Model Parallelism in DDP

• NOTE

`Pipe` only supports intra-node pipelining currently, but will be expanded to support inter-node pipelining in the future. The forward function returns an `RRef` to allow for inter-node pipelining in the future, where the output might be on a remote host. For intra-node pipelinining you can use `local_value()` to retrieve the output locally.

• NOTE

You can wrap a `Pipe` model with `torch.nn.parallel.DistributedDataParallel` only when the checkpoint parameter of `Pipe` is `'never'`.

3.3. Model Parallelism in DDP

Youhe-Jiang commented on Jun 30, 2022 • edited

As shown in the figure below, pytorch does not support inter-node pipelining. I want to know if fairscale supports it? Thanks for any replies!

```
.. note::
    :class:`Pipe` only supports intra-node pipelining currently, but
    will be expanded to support inter-node pipelining in the future.
    The forward function returns an :class:`~torch.distributed.rpc.RRef`
    to allow for inter-node pipelining in the future, where the output
    might be on a remote host. For intra-node pipelinining you can use
    :meth:`~torch.distributed.rpc.RRef.local_value` to retrieve the
    output locally.
```

min-xu-ai commented on Jul 1, 2022 Contributor

There is some experimental support for multiprocess pipe. Unfortunately, we don't have anyone who is familiar with it at the moment.

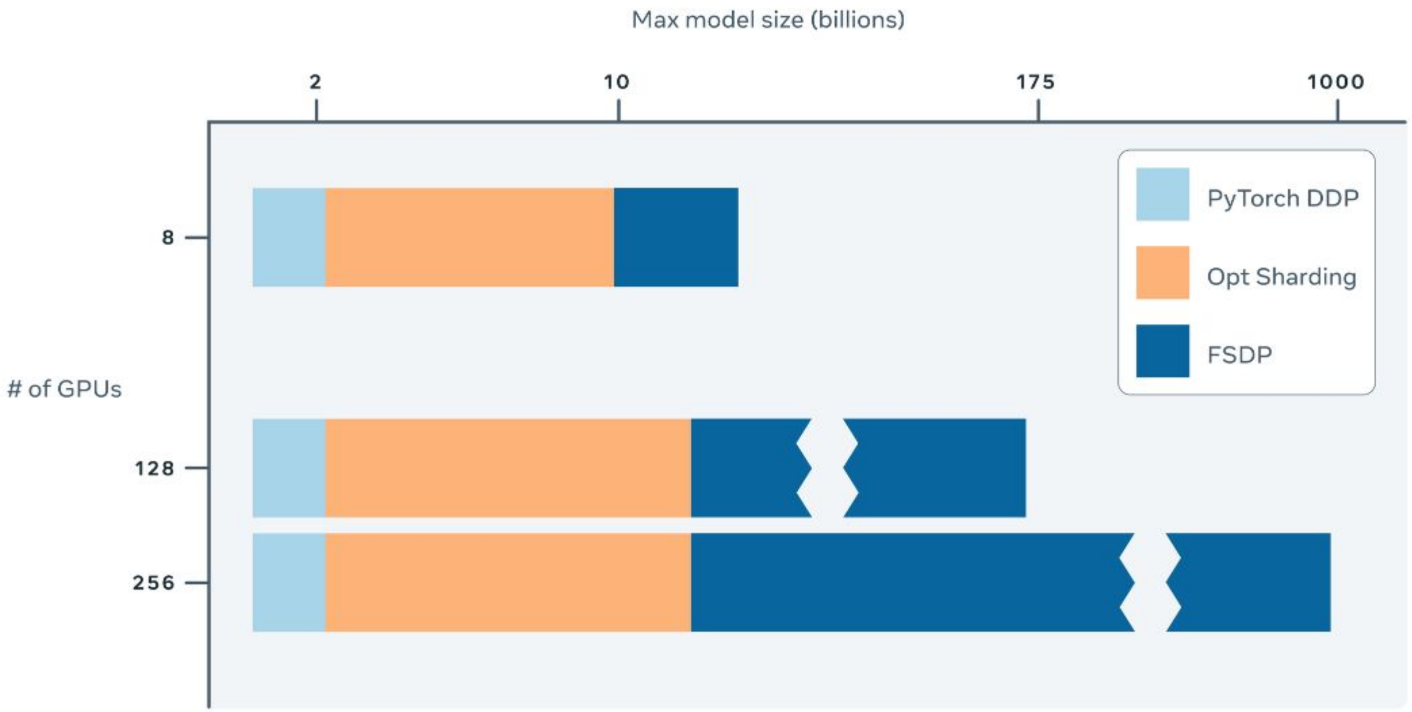
4. FSDP

Fully Sharded Data Parallel FSDP is Data and Model parallel training algorithm.

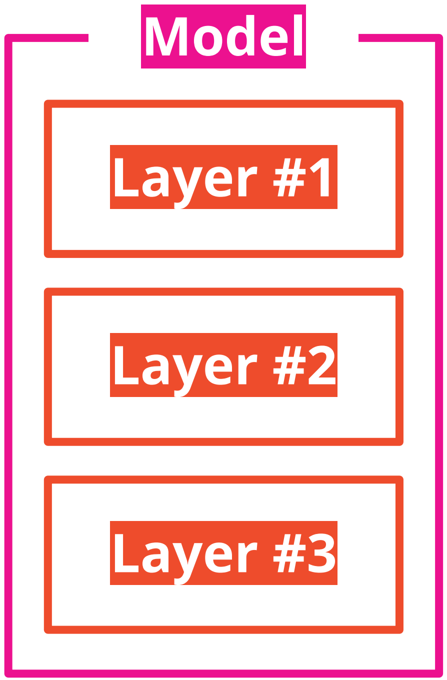
FSDP aims to train very large models on large cluster of machines.

FSDP solves problems in DDP, such as insufficient model parallelism support or inefficient communications.

4. FSDP


Max model size (billions)
2
10
175
1000
# of GPUs
8
128
256
PyTorch DDP
Opt Sharding
FSDP

4.1. Full Parameter Sharding



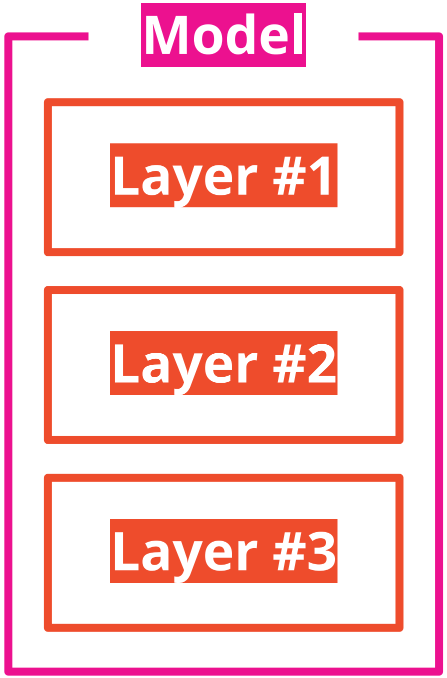
Gradient

Gradient through either forward and backward pass of specific subset of the model (here, layers).

e.g.

1. Forward pass through only Layer #1.
2. Backward pass through Layer #2 (given Gradient of Layer #3)

4.1. Full Parameter Sharding



Optimizer States

Gradients through backward pass of complete model.

i.e. Backward pass through all Layer #1, #2, #3.

= All backward Gradients

4.1. Full Parameter Sharding

Target of Replication

DDP	FSDP
Full Model Weights	Partial Model Weights
Full Optimizer States	Partial Optimizer States
	Partial Gradients

4.1. Full Parameter Sharding

Full Parameter Sharding shards across processes gradients of the model.

The name of Full Parameter Sharding comes from that parameters are “fully sharded” across the cluster.

c.f. Replication of full parameters in DDP

Thus, FSDP can be regarded as native Inter-layer Parallelism or Pipeline Parallelism.

4.2. FSDP optimization strategy

FSDP also ensures that the computation for each microbatch still local to each GPU worker, even with Full Parameter Sharding. (#)

FSDP does so by sharing parameters in every forward/backward compute;

i.e. instead of sharing each microbatch.

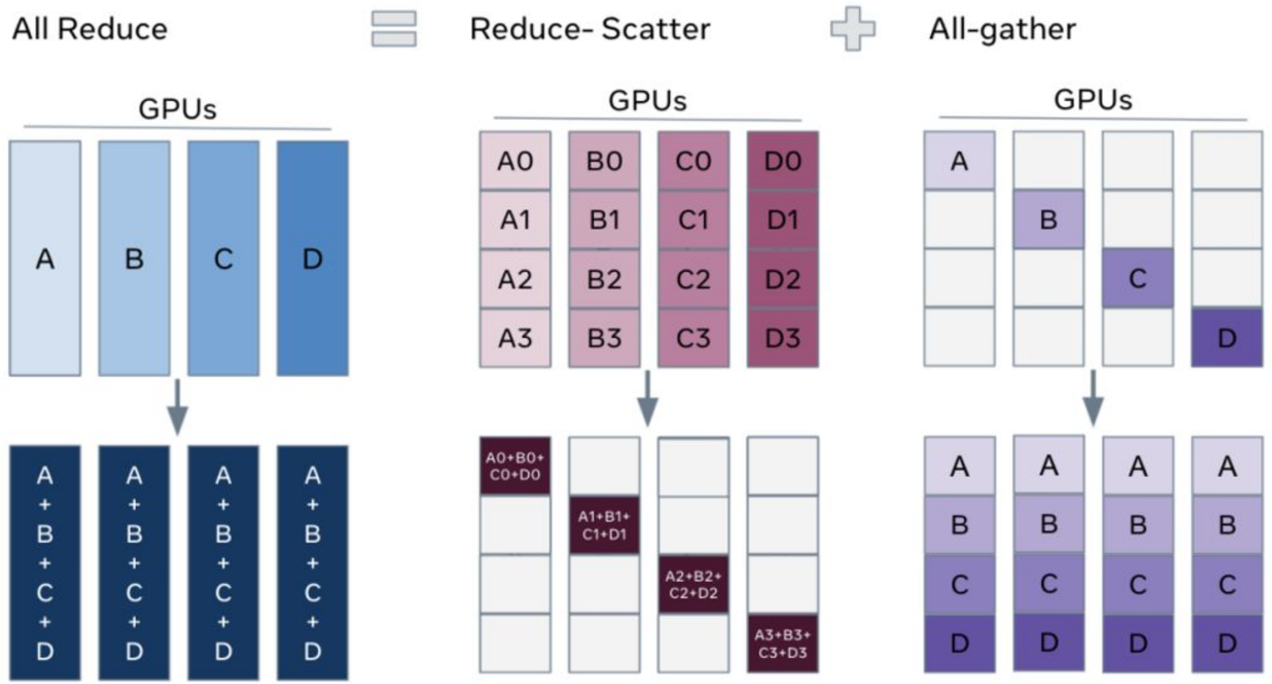
4.2. FSDP optimization strategy

Because FSDP synchronizes for Gradient, FSDP involves larger communication volume compared to DDP.

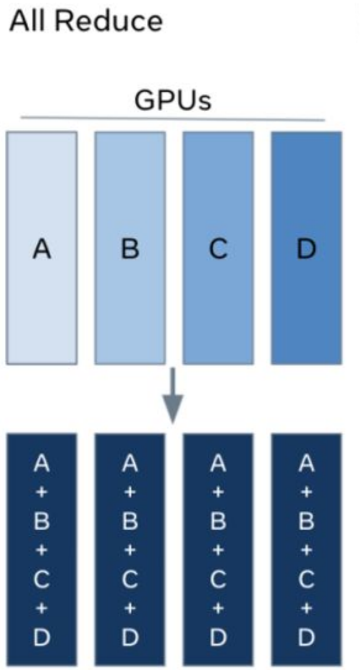
FSDP reduces such increased communication overhead by

Decomposing communication/computation and Overlapping them in training

5. How FSDP operates



5.1. Decomposition of All-Reduce



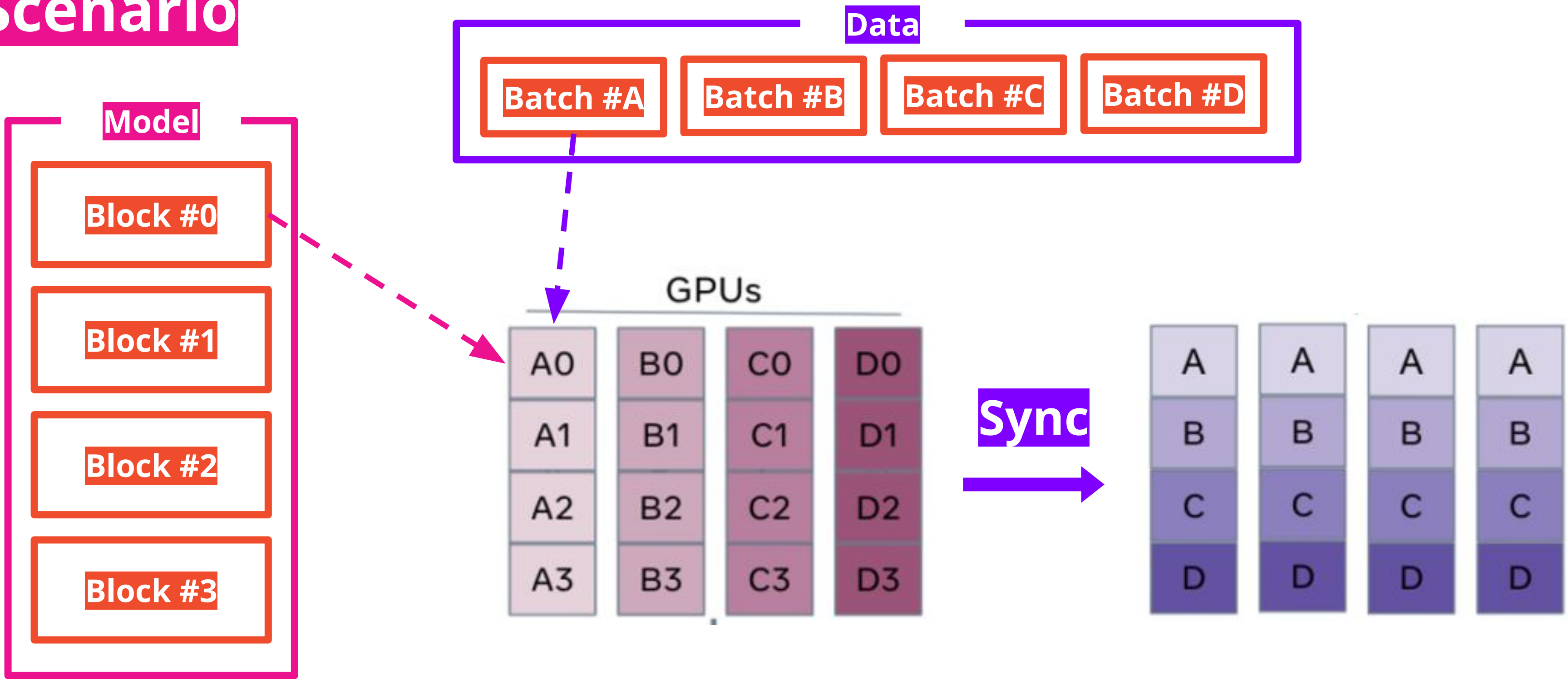
In standard DDP,

optimizer states with respect to each batches are sync-ed across all devices via All-Reduce.

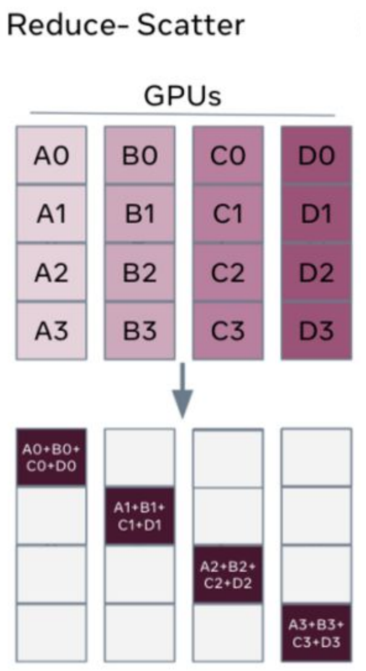
In FSDP,
such optimizer states are decomposed into multiple Gradients.

5.1. Decomposition of All-Reduce

Scenario



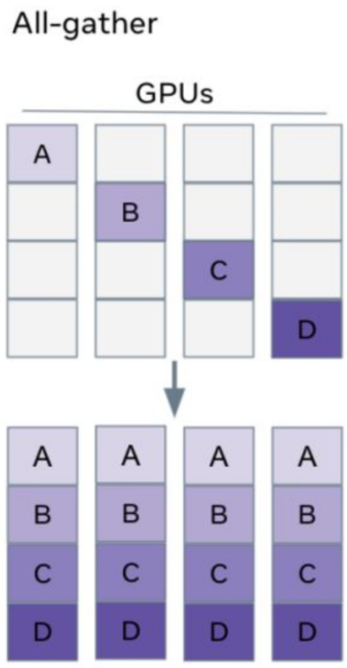
5.1. Decomposition of All-Reduce



During the Reduce-Scatter phase,

the gradients are aggregated in equal blocks among ranks based on their rank index.

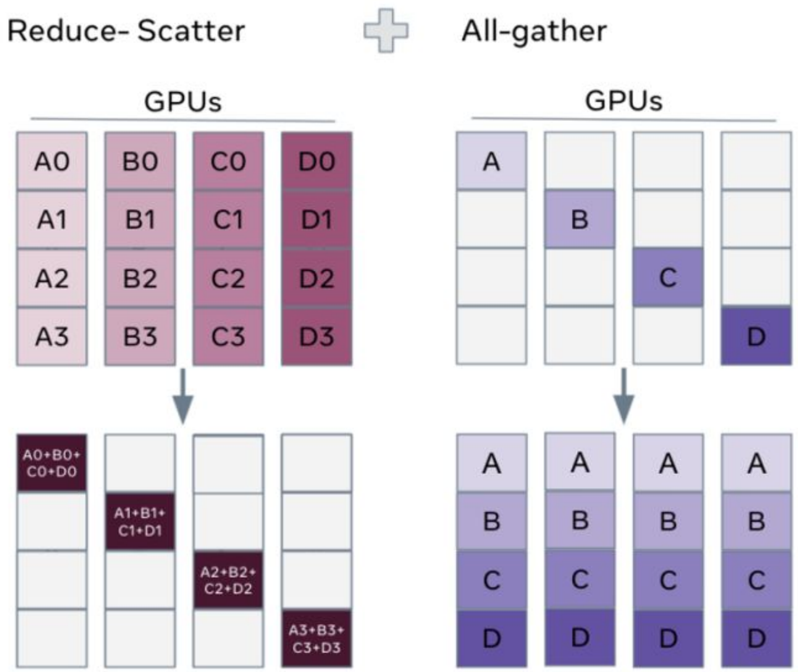
5.1. Decomposition of All-Reduce



During the all-gather phase,

the sharded portion of aggregated gradients on each GPU are propagated to all GPUs.

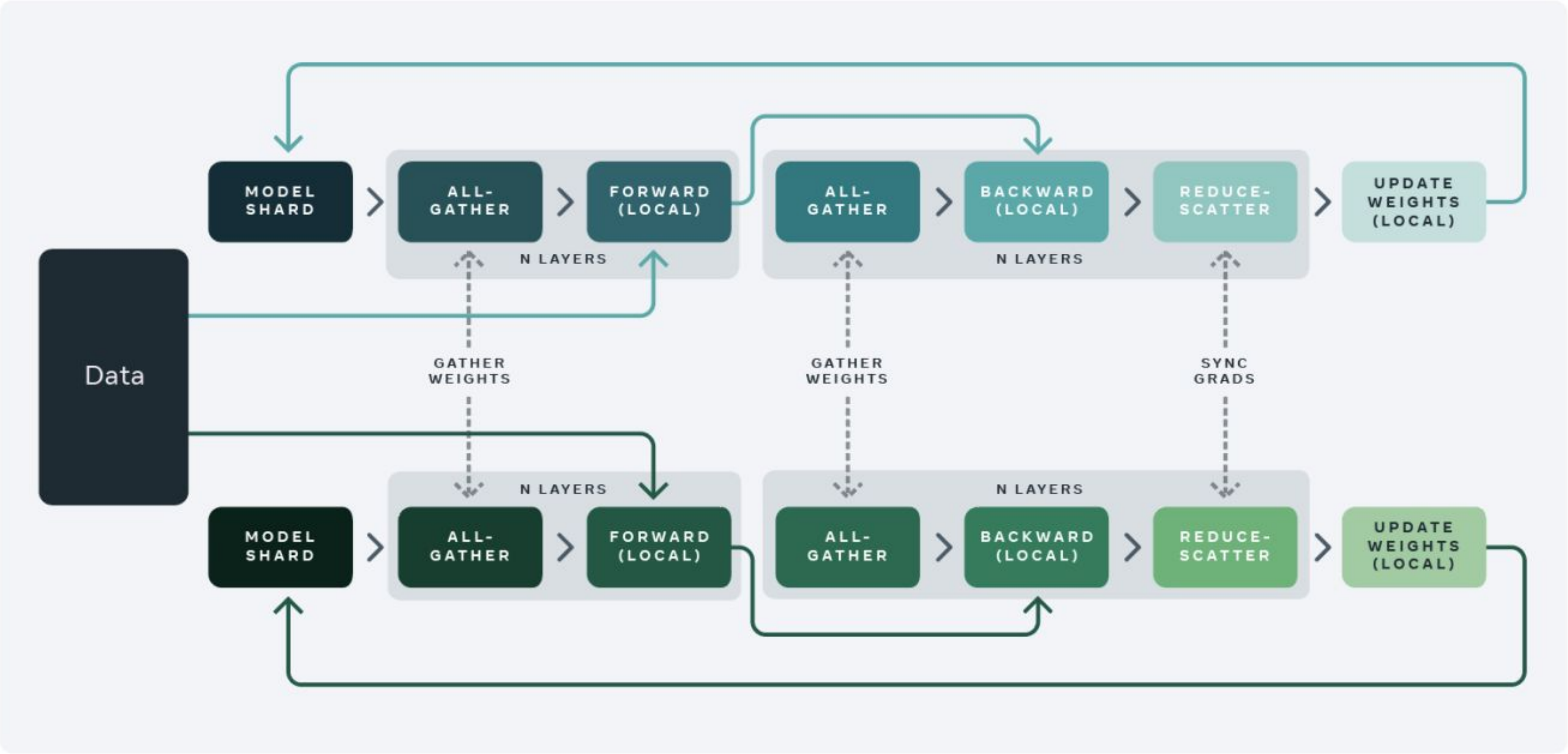
5.1. Decomposition of All-Reduce



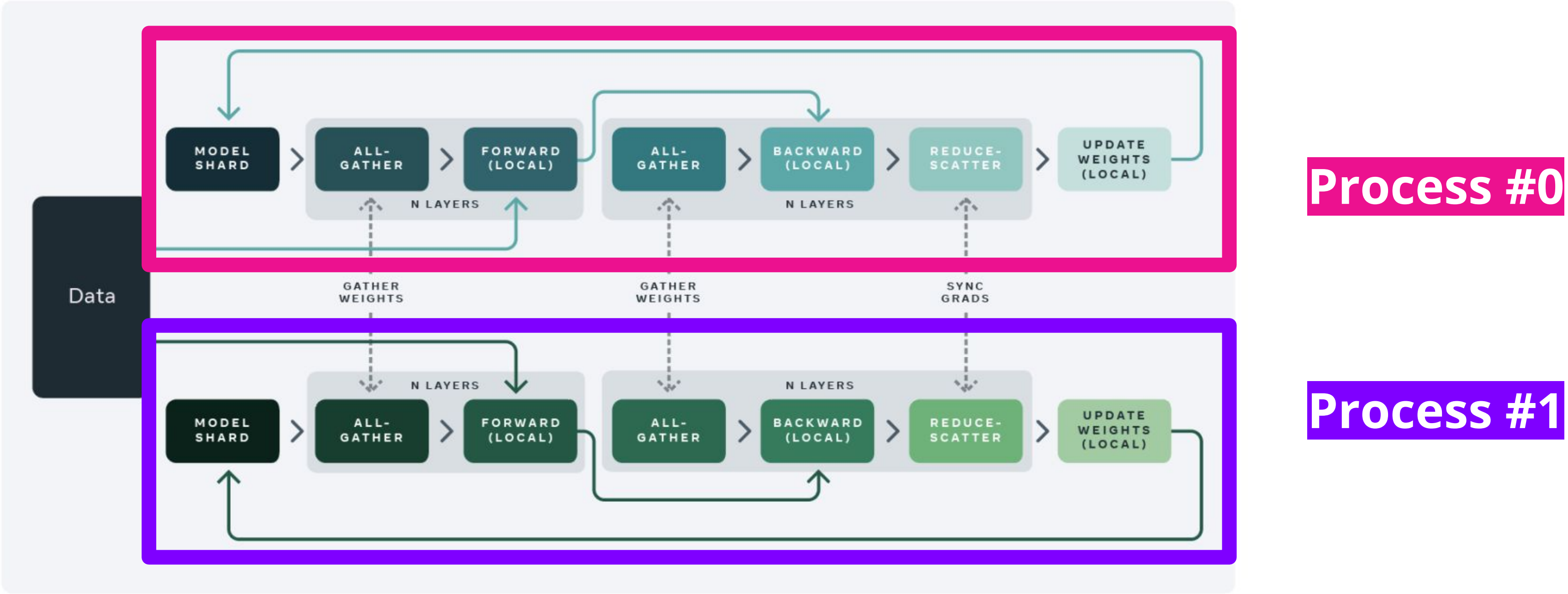
Once All-Reduce is decomposed into multiple Reduce-Scatter + All-Gather,

we can now Rearrange them and Run Asynchronously.

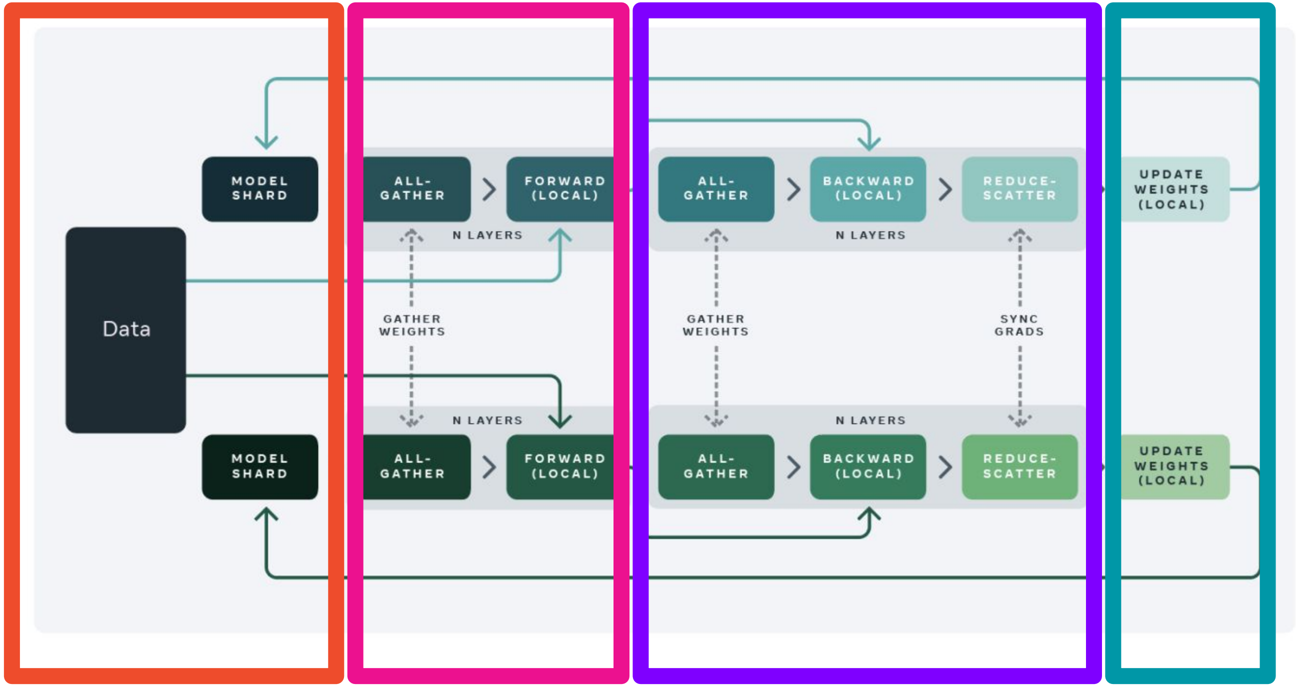
5.2. Structure of FSDP



5.2. Structure of FSDP

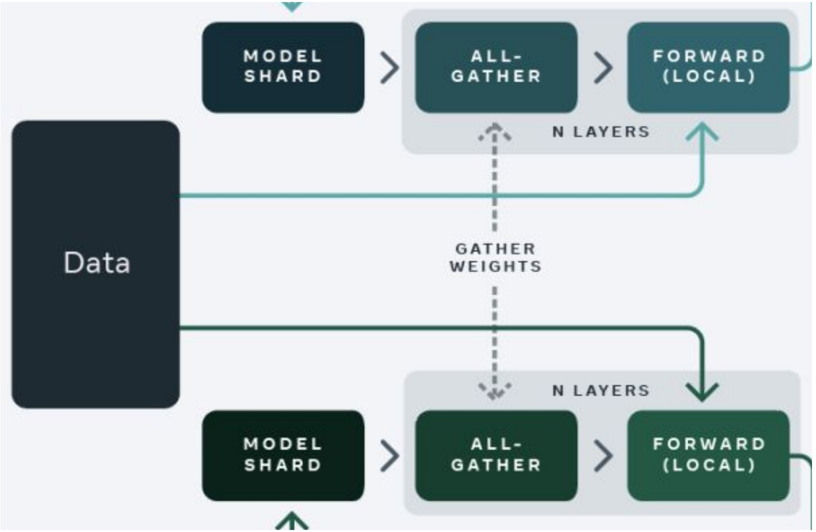


5.2. Structure of FSDP



1. Construct
2. Forward Pass
3. Backward Pass
4. Update

5.2.1. Construct



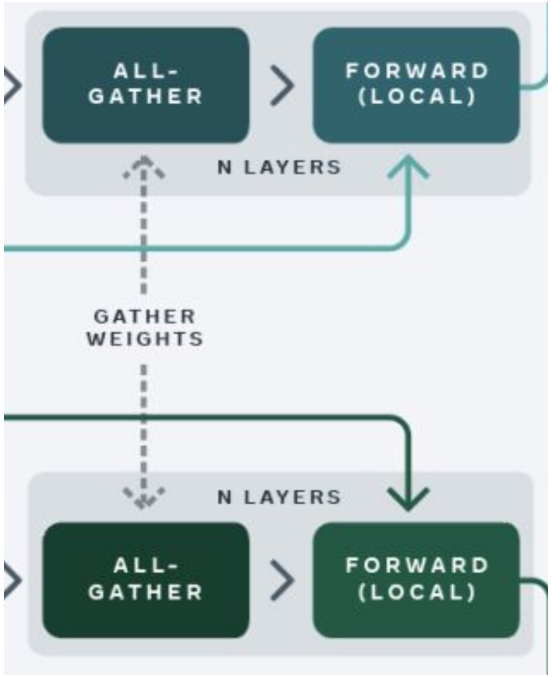
Each process contains one block of the model;

e.g. Process #0 contains **Block #0**.

Dataset is also sharded for each process;

e.g. Process #0 contains **Batch #A**.

5.2.2. Forward Pass



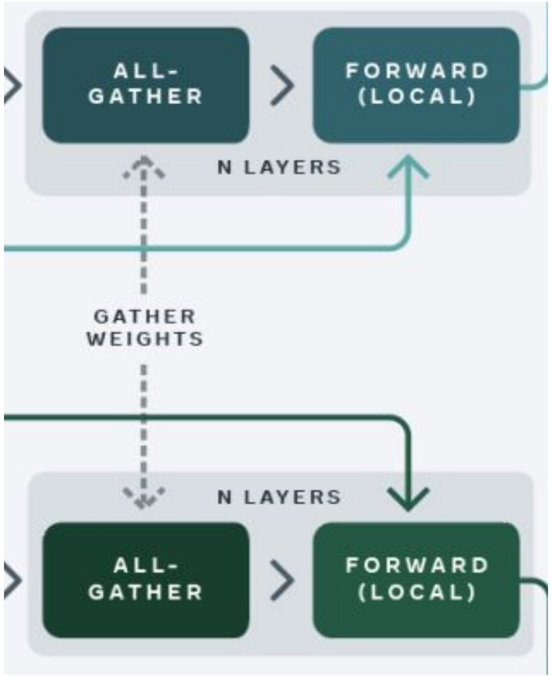
In Forward Pass stage,

each process perform following steps for all layers of the model.

1. **Get layer using All-Gather**
2. **Compute forward pass**
3. **Discard gathered layer**

(Assume there are N layers in each block)

5.2.2. Forward Pass



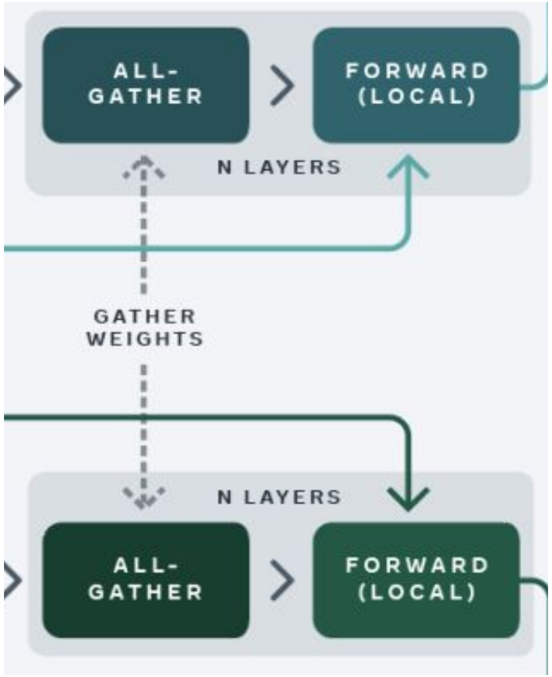
Step 1

For Layer #0 (in **Block #0**),

Process #1 calls All-Gather for Layer #0 as Process #1 doesn't have it.

(Process #0 can skip this, as Process #0 holds it on itself.)

5.2.2. Forward Pass

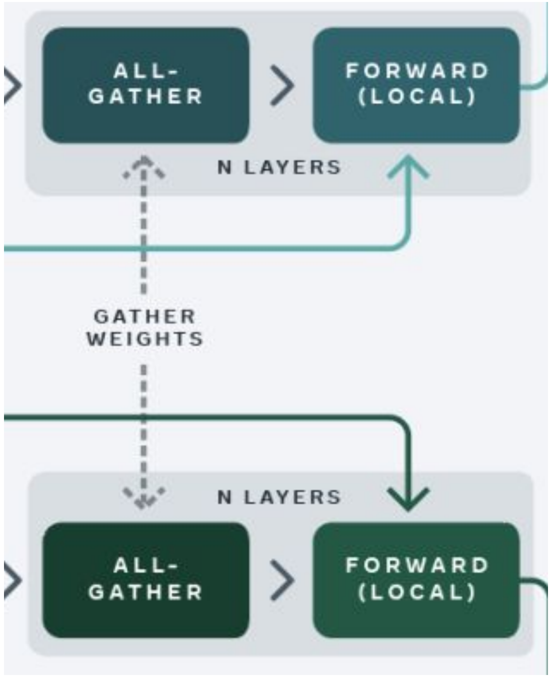


Step 2

All Processes compute for Layer #0 on their own dataset.

e.g. Process #0 compute on **Batch #A**

5.2.2. Forward Pass

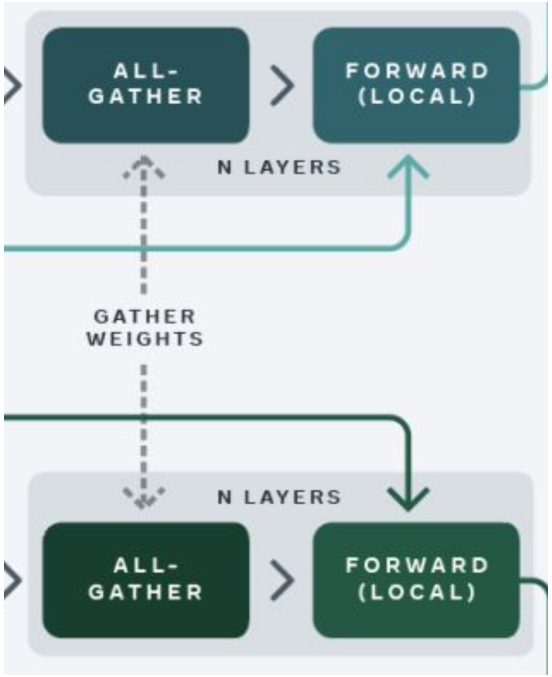


Step 3

Process #1 discards gathered layer, Layer #0.

This is to maximize memory efficiency.

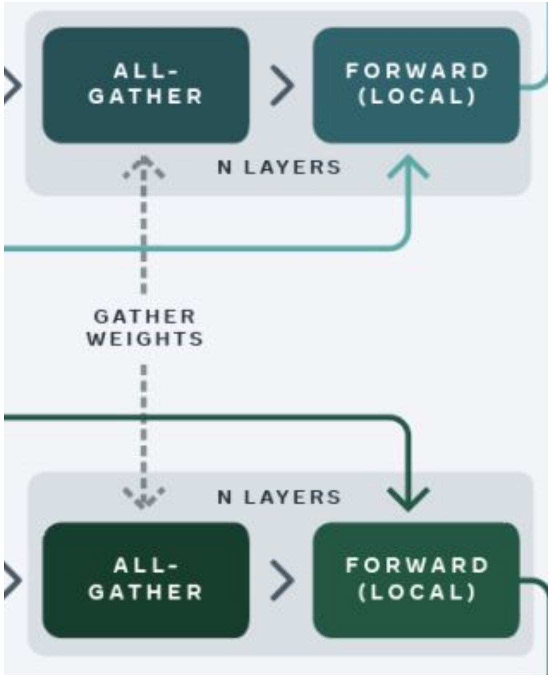
5.2.2. Forward Pass



Pseudo Code

```
FSDP forward pass:
    for layer_i in layers:
        all-gather full weights for layer_i
        forward pass for layer_i
        discard full weights for layer_i
```

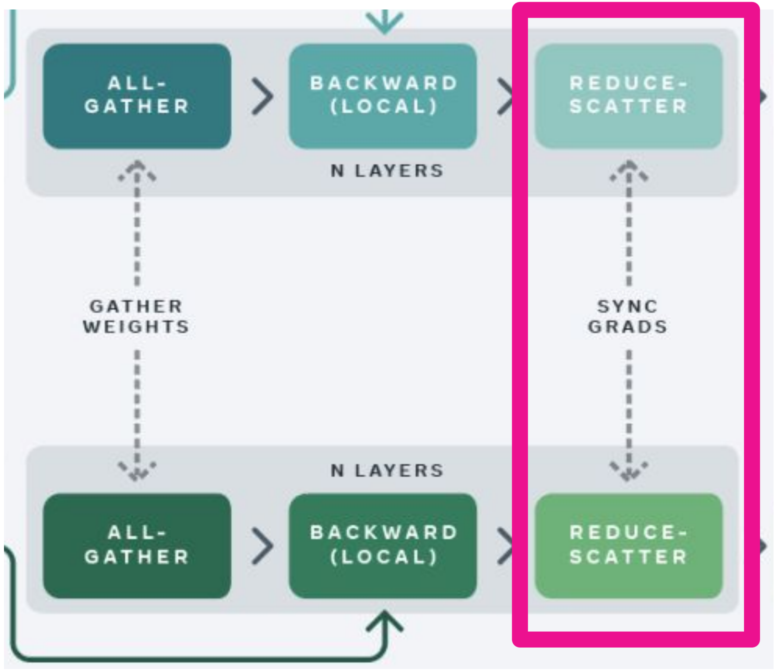
5.2.2. Forward Pass



At the end of forward pass,

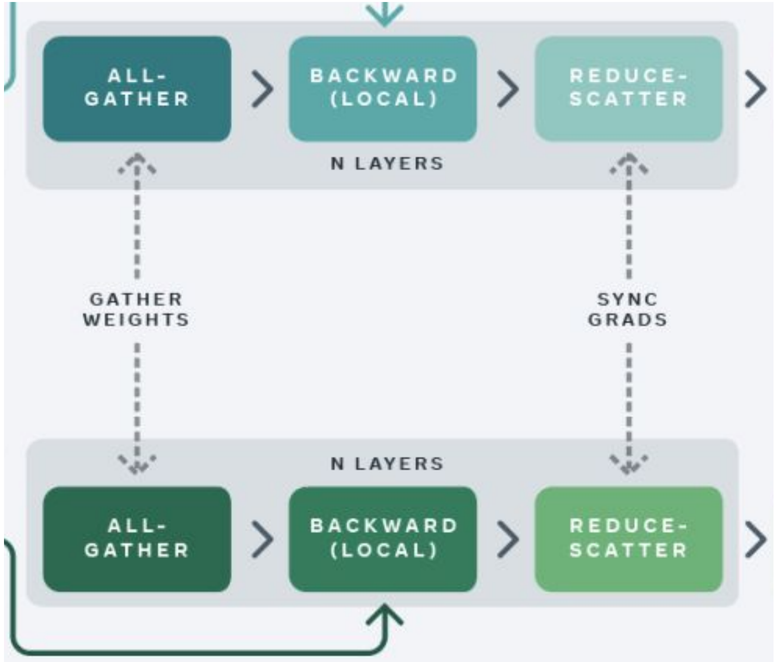
each process contains outputs of full model with respect to their own dataset.

5.2.3. Backward Pass



Computation of backward pass is the same as forward pass,
except Gradient Synchronization.

5.2.3. Backward Pass

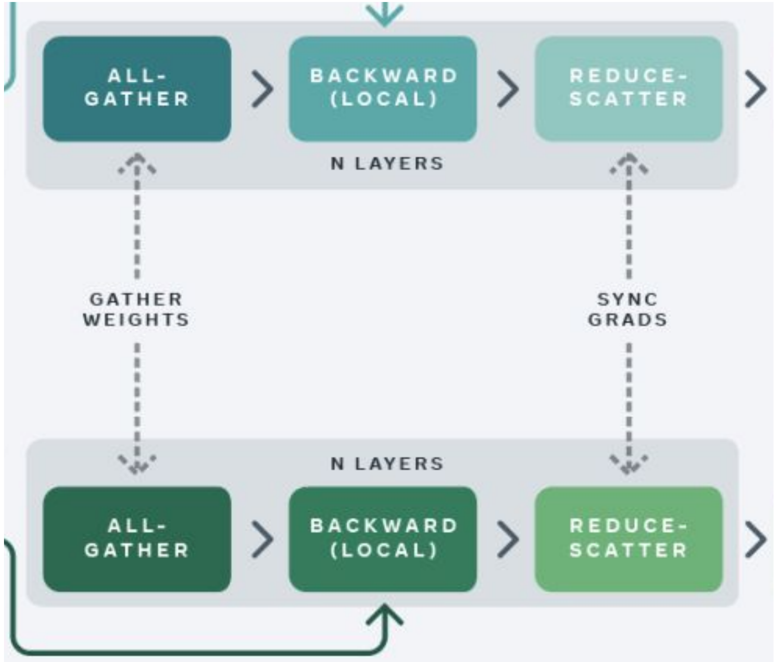


Gradient Synchronization

After backward pass for each layer,

the local gradients are synchronized across the processes via reduce-scatter.

5.2.3. Backward Pass



Pseudo Code

```
FSDP backward pass:
    for layer_i in layers:
        all-gather full weights for layer_i
        backward pass for layer_i
        discard full weights for layer_i
        reduce-scatter gradients for layer_i
```

5.2.4. Update

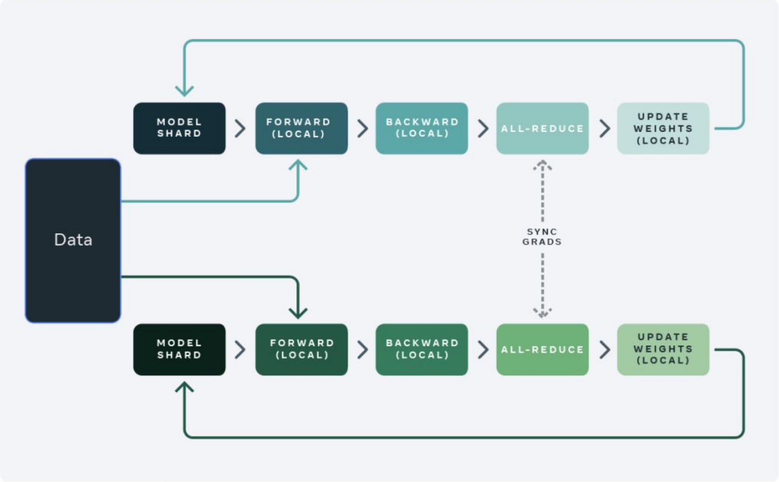
Weight Update

Given reduced gradients from other processes,

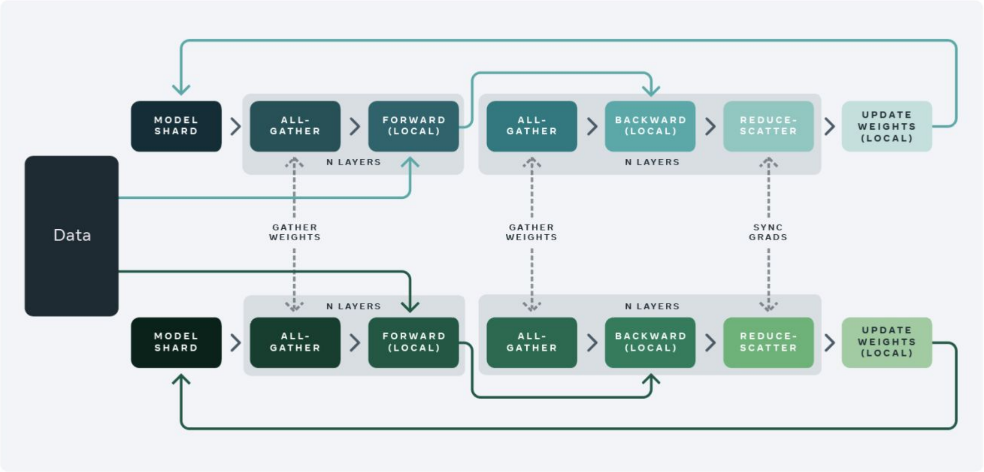
each process **updates its local weight.**

5.3. Comparison to DDP

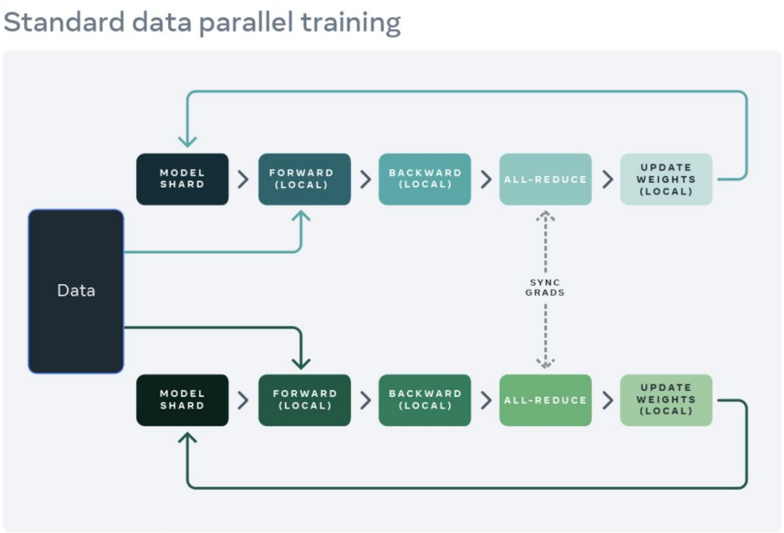
Standard data parallel training

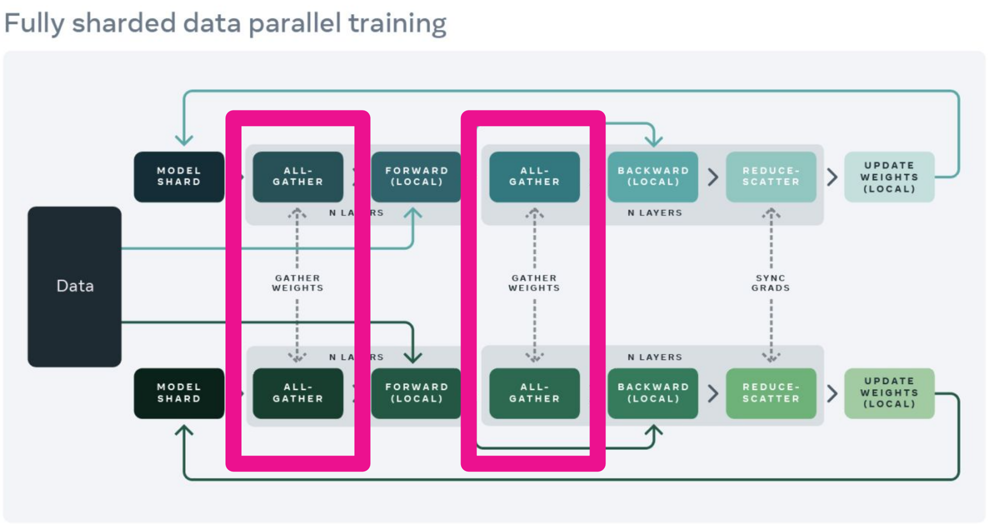


Fully sharded data parallel training



5.3. Comparison to DDP

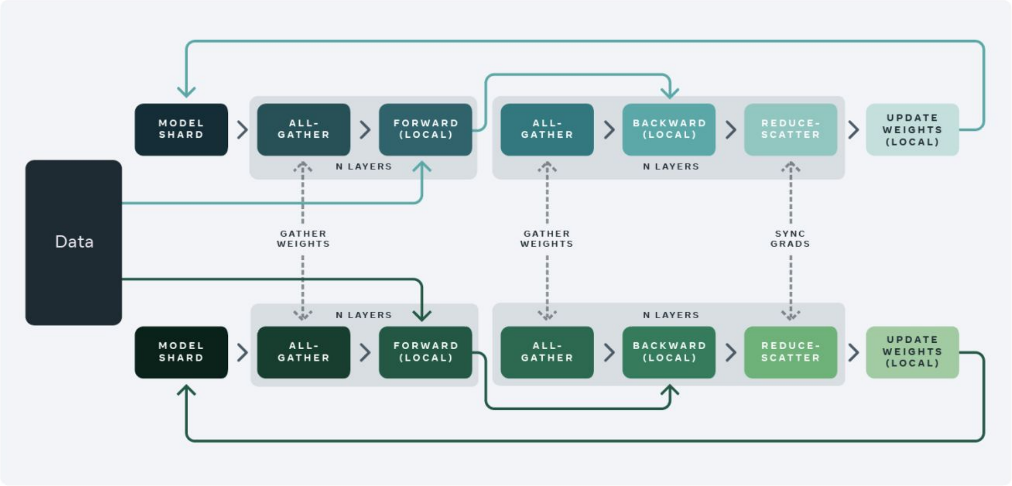




FSDP does have more communications than DDP.

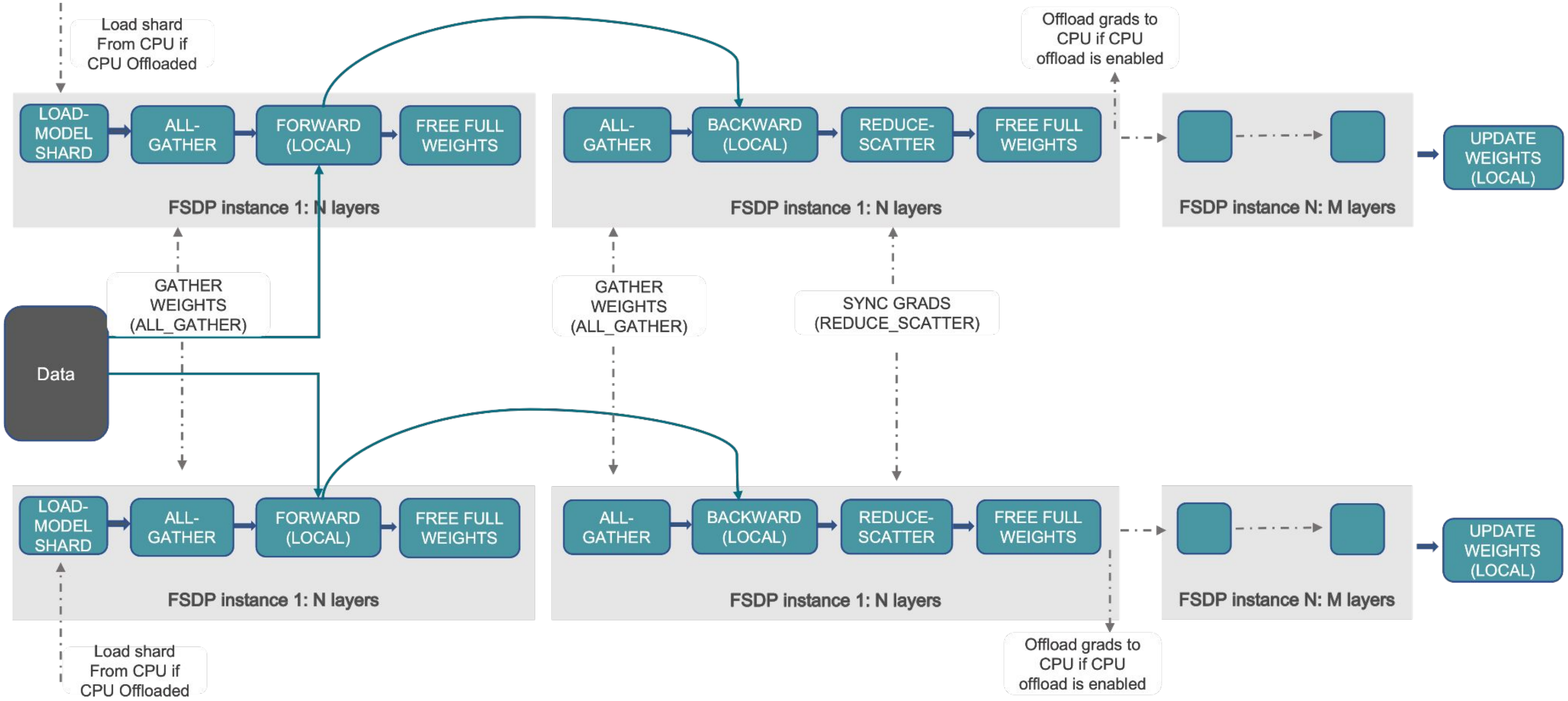
5.3. Comparison to DDP

Fully sharded data parallel training

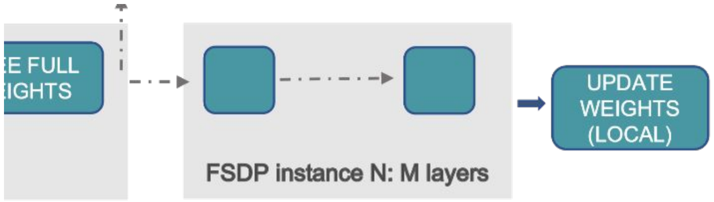


However, FSDP can overlap communication with computation.

5.4. Full Architecture of FSDP



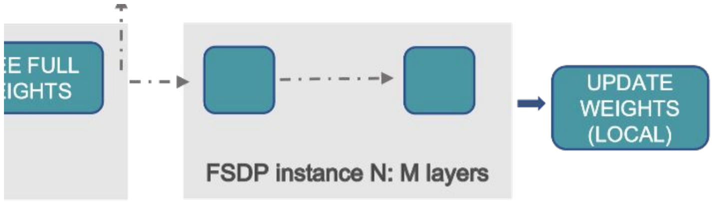
5.4. Full Architecture of FSDP



Although previous example has 1:N layers per FSDP instance,

FSDP can also map N:M layers.

5.4. Full Architecture of FSDP



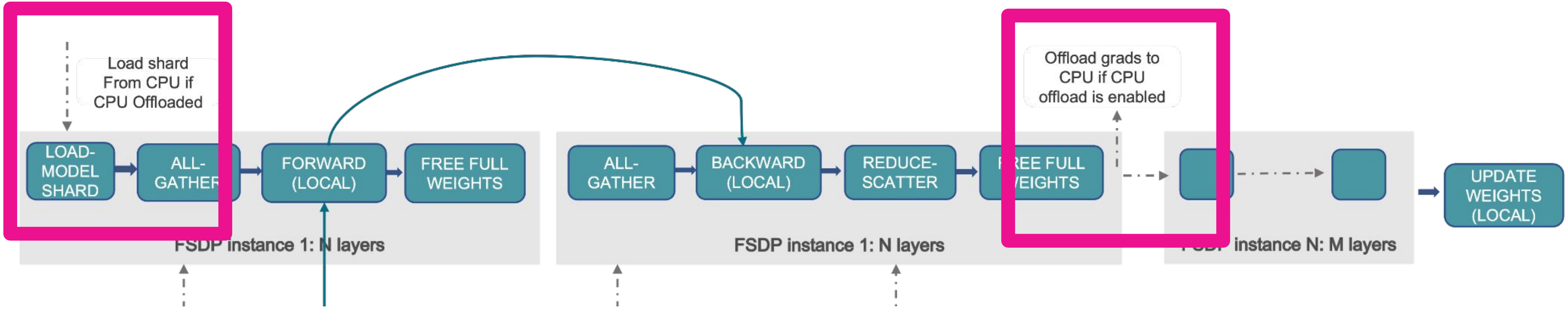
In this case,

although some instances contain redundant layers, (= waste of memory),
they are cooperating in communication
(= share communication load).

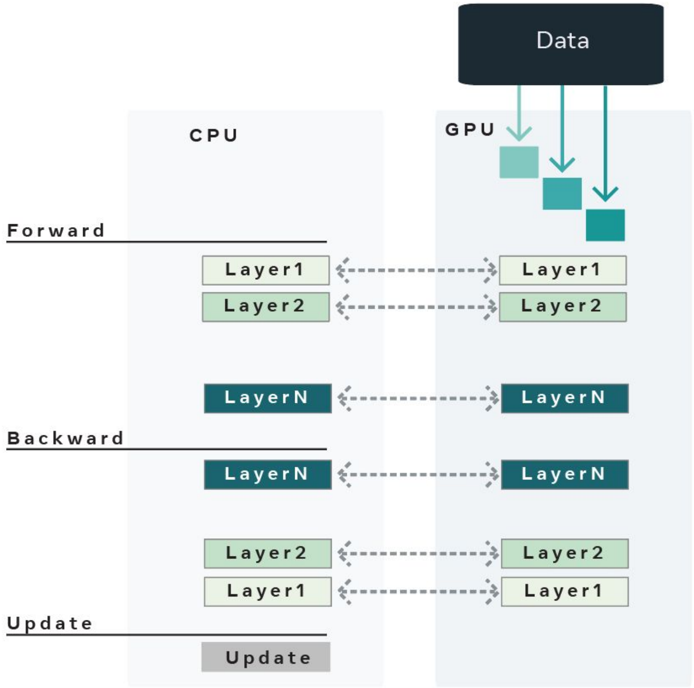
5.4. Full Architecture of FSDP

FSDP also supports CPU-offload (Zero-Offload)

in cases where even sharded parameters are too large to fit in each device.



5.4. Full Architecture of FSDP



CPU Offload for shard loading

CPU memory temporally contains model shard and sends copy to device.

5.4. Full Architecture of FSDP

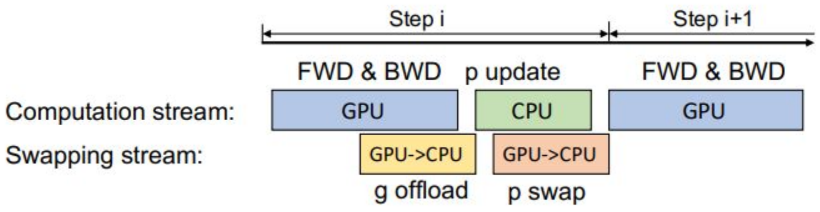


Figure 3: ZeRO-Offload training process on a single GPU.

CPU Offload for Optimizer

CPU is responsible for updating the parameters and holding onto the optimizer state.

5.5. Performance Analysis

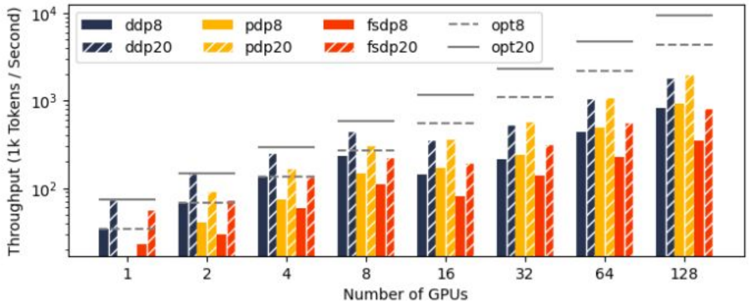


Figure 3. GPTSmall (125M) Throughput vs Number of GPUs

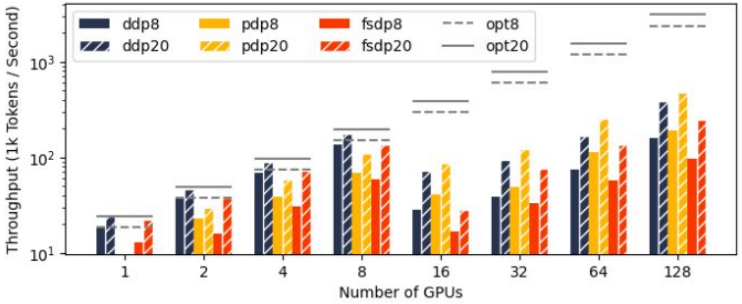


Figure 4. GPTLarge (760M) Throughput vs Number of GPUs

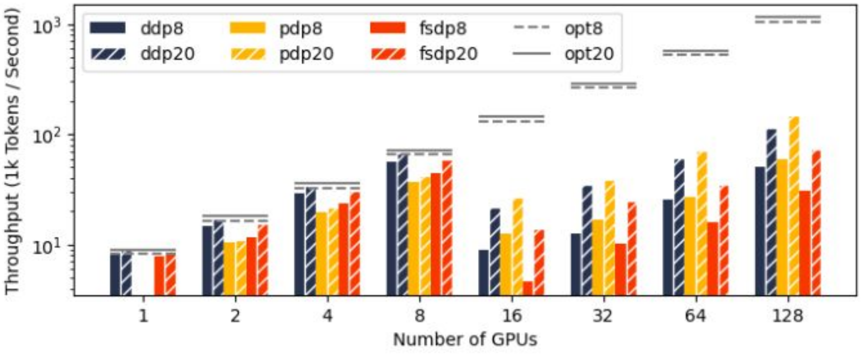


Figure 5. GPT2.7B Throughput vs Number of GPUs

For small, medium sized models, FSDP suffers severely from Communication Overhead.

5.5. Performance Analysis

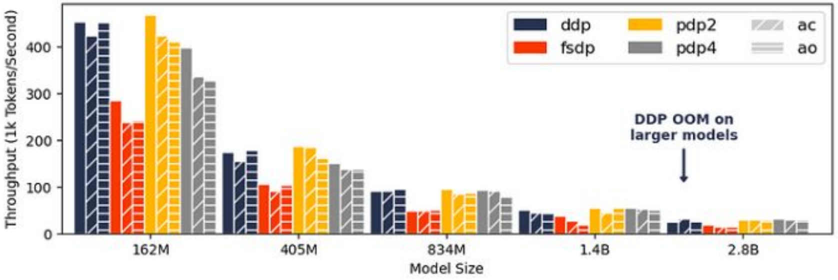


Figure 10. Throughput vs Model Size (162M — 2.8B)

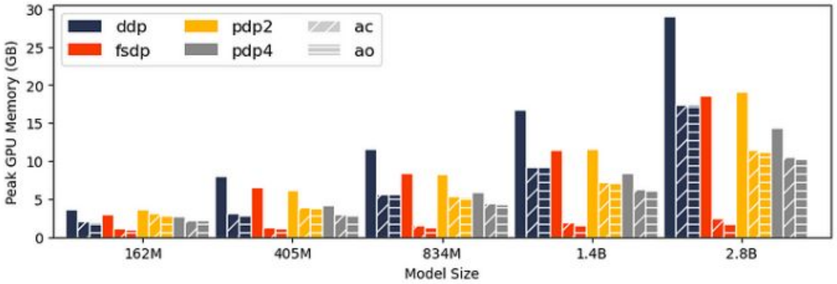


Figure 11. Peak GPU Memory vs Model Size (162M — 2.8B)

But FSDP can train large models, with low memory footprint.

5.5. Performance Analysis

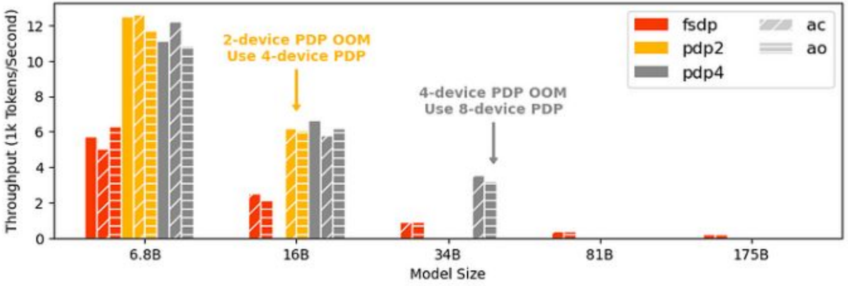


Figure 12. Throughput vs Model Size (6.8B — 175B)

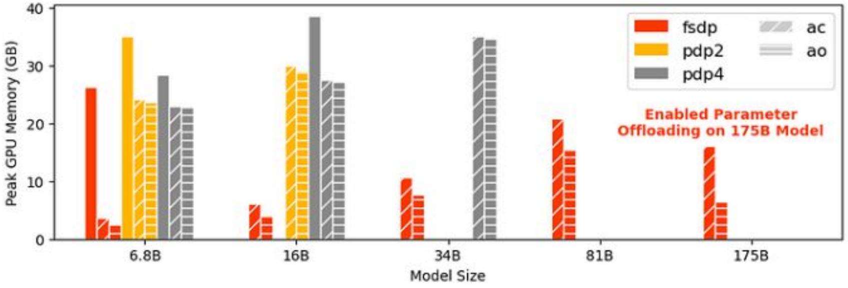


Figure 13. Peak GPU Memory vs Model Size (6.8B — 175B)

Only FSDP can train extremely large models.

5.5. Performance Analysis

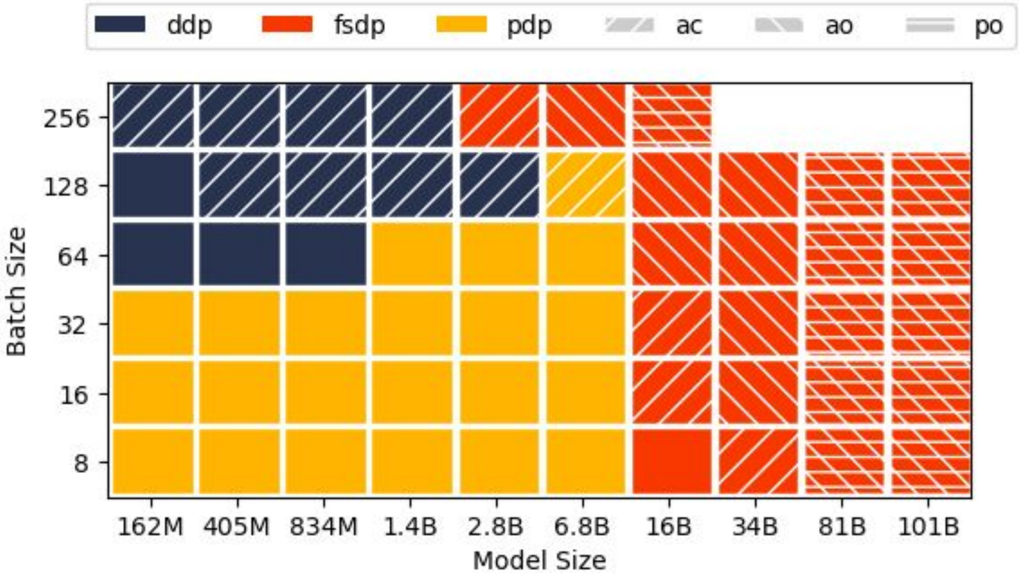


Figure 20. Paradigm Recommendations for 100Gbps Ethernet

Acknowledgement

PYTORCH DISTRIBUTED OVERVIEW [https://pytorch.org/tutorials/beginner/dist_overview.html]

GETTING STARTED WITH DISTRIBUTED DATA PARALLEL [https://pytorch.org/tutorials/intermediate/ddp_tutorial.html]

WRITING DISTRIBUTED APPLICATIONS WITH PYTORCH [https://pytorch.org/tutorials/intermediate/dist_tuto.html]

DISTRIBUTED RPC FRAMEWORK [https://pytorch.org/docs/stable/rpc.html]

Launching and configuring distributed data parallel applications [https://github.com/pytorch/examples/blob/main/distributed/ddp/README.md]

PIPELINE PARALLELISM [https://pytorch.org/docs/stable/pipeline.html]

TRAINING TRANSFORMER MODELS USING DISTRIBUTED DATA PARALLEL AND PIPELINE PARALLELISM [https://pytorch.org/tutorials/advanced/ddp_pipeline.html]

ELASTIC AGENT [https://pytorch.org/docs/stable/elastic/agent.html]

Acknowledgement

All-Reduce Implementation approaches [https://algopoolja.tistory.com/95]

Automatic Cross-Replica Sharding of Weight Update in Data-Parallel Training [https://arxiv.org/pdf/2004.13336.pdf]

Fully Sharded Data Parallel: faster AI training with fewer GPUs [https://engineering.fb.com/2021/07/15/open-source/fsdp/]

Paradigms of Parallelism [https://colossalai.org/docs/concepts/paradigms_of_parallelism/]

GPipe: Easy Scaling with Micro-Batch Pipeline Parallelism [https://arxiv.org/pdf/1811.06965.pdf]

Parallelism in Distributed Deep Learning [https://insujang.github.io/2022-06-11/parallelism-in-distributed-deep-learning/]

Zero Offloading brief explanation
[https://moon-walker.medium.com/large-model-%ED%95%99%EC%8A%B5%EC%9D%98-game-changer-ms%EC%9D%98-deepspeed-zero-1-2-3-%EA%B7%B8%EB%A6%AC%EA%B3%A0-zero-infinity-74c9640190de]

ZeRO-Offload: Democratizing Billion-Scale Model Training [https://arxiv.org/df/2101.06840.pdf]

PyTorch Data Parallel Best Practices on Google Cloud [https://medium.com/pytorch/pytorch-data-parallel-best-practices-on-google-cloud-6c8da2be180d]