Data Parallel Training

COMP4901Y

Binhang Yuan

Data Parallel Paradigm

Recall the Mini-Batch SGD

RELAXED SYSTEM LAB

- Basic ideas:
  - To reduce the variance of stochastic gradients;
  - Split the training data into smaller batches;
  - Sampling batch (usually without replacement)
- Suppose we have:
  - B is the batch size:
  - K is the number of GPUs.
- ***Distribute the computation by the data dimension.***

$$w_{t+1} = w_t - \alpha_t \cdot \frac{1}{\mathrm{B}} \sum_{i=1}^{B} \nabla f(w_t; \xi_i)$$

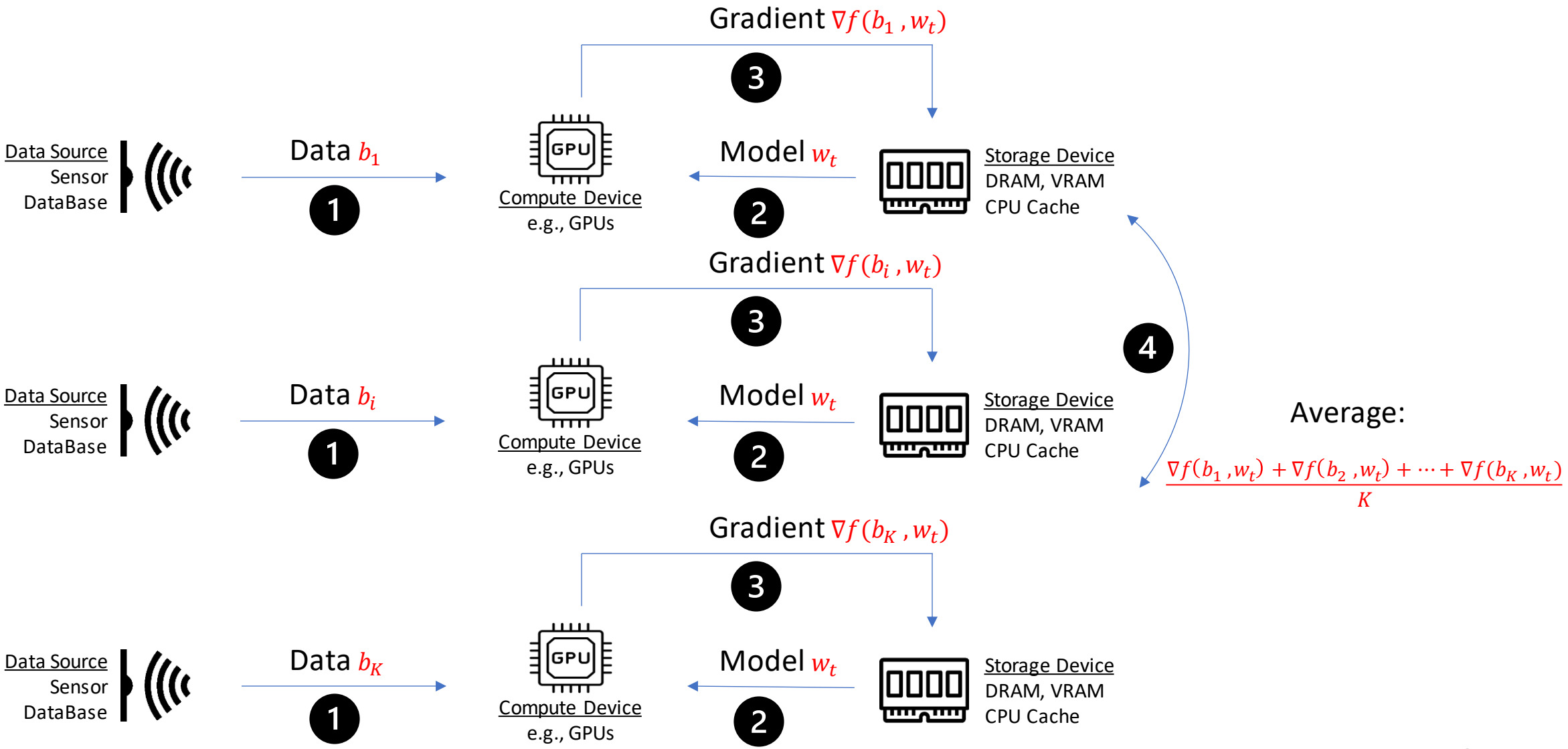
Local Computation

$$w_{t+1} = w_t - \alpha_t \cdot \left(\underbrace{\frac{1}{\mathrm{B}} \sum_{i_1=1}^{\frac{B}{K}} \nabla f(w_t; \xi_{i_1})}_{\textbf{GPU 1}} + \underbrace{\frac{1}{\mathrm{B}} \sum_{i_2=1}^{\frac{B}{K}} \nabla f(w_t; \xi_{i_2})}_{\textbf{GPU 2}} + \cdots + \underbrace{\frac{1}{\mathrm{B}} \sum_{i_K=1}^{\frac{B}{K}} \nabla f(w_t; \xi_{i_K})}_{\textbf{GPU K}} \right)$$

Aggregation

Data Parallel SGD

RELAXED SYSTEM LAB



Basic Implementation

RELAXED SYSTEM LAB

Core idea of data parallelism:

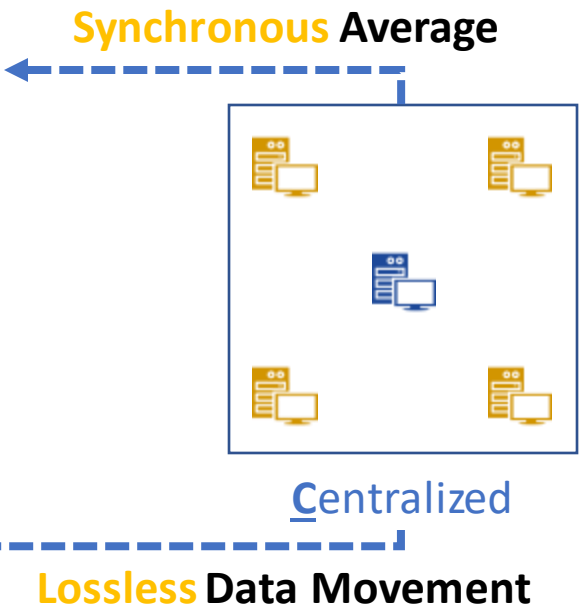
- Distribute batch gradient calculation to multiple workers;
- Synchronize gradients among GPU workers:
  - A central server (or AllReduce).
  - Synchronous average;
  - Lossless data movement.

```
w = sync_model()

b = get_data()

g = get_grad(w,b)

w = update(w,g)
```



Basic Implementation

RELAXED SYSTEM LAB

- Suppose:
  - C is the total computation in the original forward and backward computation (counted by FLOPs);
  - D is the total number of parameters of the model;
  - K is the total number of GPUs in the cluster;
- Under Data parallel training:
  - What is the total computation on each device? $\frac{C}{K}$
  - What is the communication volume? **ALLReduce** a D dimensional buffer.

PyTorch-DDP Practice

Overview

RELAXED
SYSTEM LAB

- PyTorch **DistributedDataParallel** (DDP) enables data parallel training in PyTorch.
- PyTorch **DistributedSampler** ensures each device gets a non-overlapping input batch.
- The model is replicated on all the devices.
- Each replica calculates gradients and synchronizes with the others using the **AllReduce** operation.

DDP API

DISTRIBUTEDDATAPARALLEL

CLASS torch.nn.parallel.DistributedDataParallel(*module, device_ids=None, output_device=None, dim=0, broadcast_buffers=True, process_group=None, bucket_cap_mb=25, find_unused_parameters=False, check_reduction=False, gradient_as_bucket_view=False, static_graph=False, delay_all_reduce_named_params=None, param_to_hook_all_reduce=None, mixed_precision=None, device_mesh=None*) [SOURCE]

Implement distributed data parallelism based on `torch.distributed` at module level.

This container provides data parallelism by synchronizing gradients across each model replica. The devices to synchronize across are specified by the input `process_group`, which is the entire world by default. Note that `DistributedDataParallel` does not chunk or otherwise shard the input across participating GPUs; the user is responsible for defining how to do so, for example through the use of a `DistributedSampler`.

See also: Basics and Use nn.parallel.DistributedDataParallel instead of multiprocessing or nn.DataParallel. The same constraints on input as in `torch.nn.DataParallel` apply.

Creation of this class requires that `torch.distributed` to be already initialized, by calling `torch.distributed.init_process_group()`.

`DistributedDataParallel` is proven to be significantly faster than `torch.nn.DataParallel` for single-node multi-GPU data parallel training.

To use `DistributedDataParallel` on a host with N GPUs, you should spawn up `N` processes, ensuring that each process exclusively works on a single GPU from 0 to N-1. This can be done by either setting `CUDA_VISIBLE_DEVICES` for every process or by calling:

```
>>> torch.cuda.set_device(i)
```

where i is from 0 to N-1. In each process, you should refer the following to construct this module:

```
>>> torch.distributed.init_process_group(
>>>     backend='nccl', world_size=N, init_method='...'
>>> )
>>> model = DistributedDataParallel(model, device_ids=[i], output_device=i)
```

RELAXED SYSTEM LAB

Initialize the Process Group

RELAXED SYSTEM LAB

Initialize the Process Group

```
import os
import sys
import tempfile
import torch
import torch.distributed as dist
import torch.nn as nn
import torch.optim as optim
import torch.multiprocessing as mp

from torch.nn.parallel import DistributedDataParallel as DDP


def setup(rank, world_size):
    os.environ['MASTER_ADDR'] = 'localhost'
    os.environ['MASTER_PORT'] = '12355'

    # initialize the process group
    dist.init_process_group(”nccl", rank=rank, world_size=world_size)

def cleanup():
    dist.destroy_process_group()
```

Use DDP API

RELAXED SYSTEM LAB

Define the Toy Model

```
class ToyModel(nn.Module):
    def __init__(self):
        super(ToyModel, self).__init__()
        self.net1 = nn.Linear(10, 10)
        self.relu = nn.ReLU()
        self.net2 = nn.Linear(10, 5)

    def forward(self, x):
        return self.net2(self.relu(self.net1(x)))
```

Use DDP API

RELAXED SYSTEM LAB

Use DDP API

```
def demo_basic(rank, world_size):
    print(f"Running basic DDP example on rank {rank}.")
    setup(rank, world_size)

    # create model and move it to GPU with id rank
    model = ToyModel().to(rank)
    ddp_model = DDP(model, device_ids=[rank])

    loss_fn = nn.MSELoss()
    optimizer = optim.SGD(ddp_model.parameters(), lr=0.001)

    optimizer.zero_grad()
    outputs = ddp_model(torch.randn(20, 10))
    labels = torch.randn(20, 5).to(rank)
    loss_fn(outputs, labels).backward()
    optimizer.step()

    cleanup()


def run_demo(demo_fn, world_size):
    mp.spawn(demo_fn,
             args=(world_size,),
             nprocs=world_size,
             join=True)
```

System Optimization

System Optimization

RELAXED SYSTEM LAB

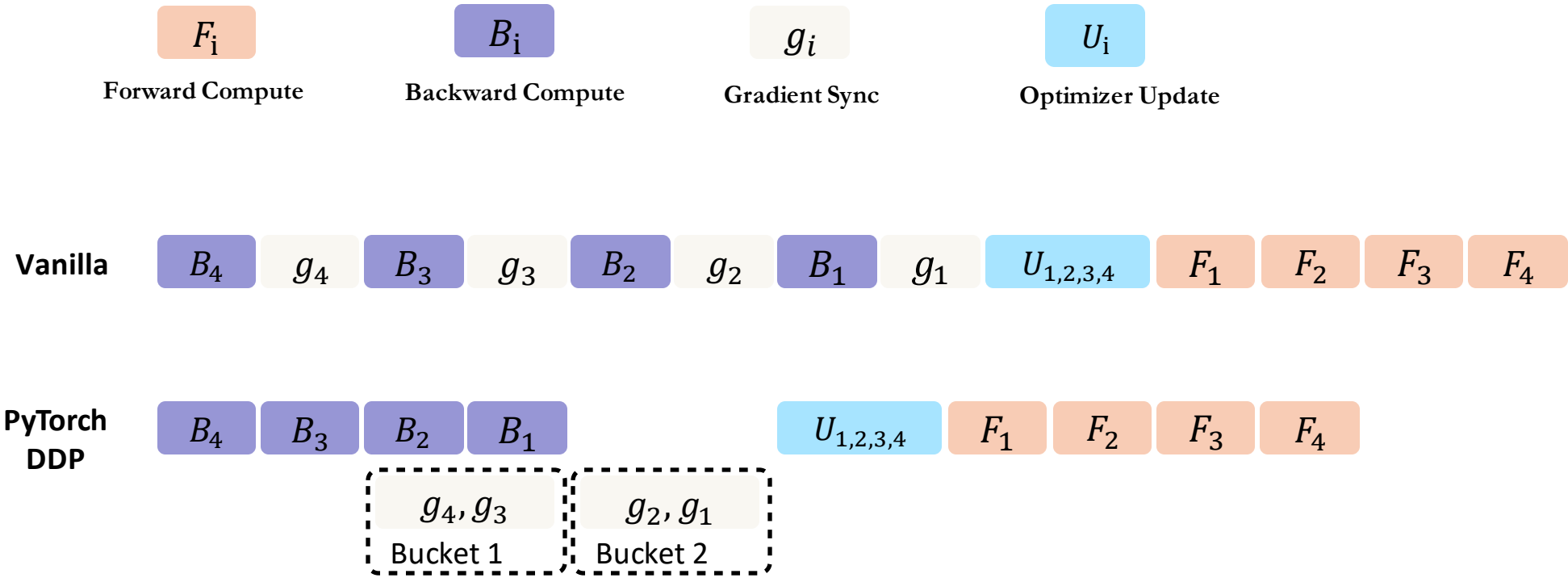
- Gradient Bucketing:
  - Collective communications are more efficient with large tensors.
  - Group gradient tensor to buckets on continuous memory (known as flatten.)
- Overlapping communication and computation during back-propagation:
  - The AllReduce operation on gradients can start before the local backward pass finishes.
  - With bucketing, DDP only needs to wait for all contents in the same bucket before launching communications.

System Optimization

RELAXED SYSTEM LAB

- **PyTorch System:**

Optimize the standard data-parallel computation:



Algorithm Optimization

Overview

RELAXED
SYSTEM LAB

- Three main categories of algorithm design:
  - Loss communication compression;
  - Asynchronous training:
  - Decentralized communication.

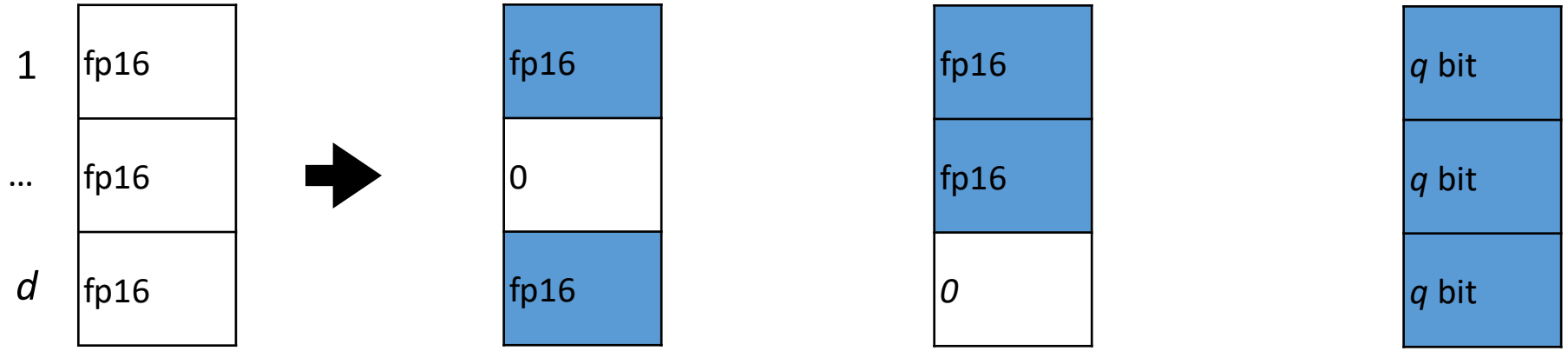
Loss Communication Compression

RELAXED SYSTEM LAB

- Instead of exchanging the gradient as full precision floating point numbers;
- The system first compresses it into a lower precision representation before communication.
- Compression methods:
  - Top-K sparsification;
  - Random sparsification;
  - Quantization.

Compression Methods

RELAXED SYSTEM LAB



Expensive to compute and to encode Indices

Might not keep top values as in Top-K

Only provide up to 32x compression; hard to go aggressive

Asynchronous Training

RELAXED
SYSTEM LAB

- Original paradigm: all workers conduct computation simultaneously and are all blocked until the communication among all machines is finished.
- If some machines are slower than other machines (i.e., there are stragglers), all machines need to wait for the slowest machine to finish the computation.
- Asynchronous training removes the synchronization barriers among all the machines with the consequence that each machine now has access to a staled model.

Decentralized Communication

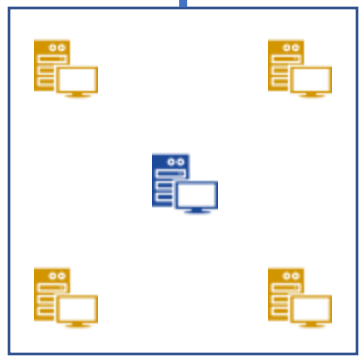
RELAXED SYSTEM LAB

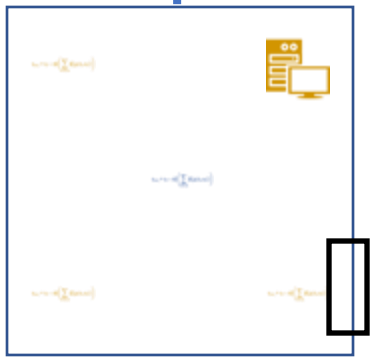
- Lossy communication compression is designed to alleviate system bottlenecks caused by network bandwidth.
- Another type of network bottleneck is caused by latency.
- In that case, when there are K workers, the ring -based **AllReduce** implementation has an $\mathcal{O}(K)$ dependency on the network latency.
- Decentralized training:
  - Suppose a logical ring among K workers;
  - At every single iteration, a worker sends its model replica to the neighbour on its immediate left and neighbour on its immediate right;
  - Latency overhead becomes $\mathcal{O}(1)$.
  - On the downside, however, the information on a single worker only reaches its two adjacent neighbors in one round of communication.

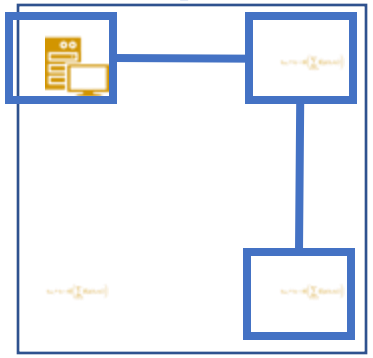
Algorithm Illustration.

RELAXED SYSTEM LAB

```
w = sync_model()
b = get_data()
g = get_grad(w, b)
w = update(w, g)
```







Synchronous Average

Asynchronous Average

Decentralized Average

Centralized, Quantized

Asynchronous

Decentralized

Lossy Data Movement

Lossless Data Movement

Lossless Data Movement

Mathematical Formulation

$$w_{t+1} = w_t - \gamma C\left(\sum_{i=1..n} C(\nabla f_i(x_t, b_i))\right)$$

$$w_{t+1} = w_t - \gamma \nabla f(x_{t-\tau_t}; b_i)$$

staleness caused by async

$$w_{t+1,i} = \frac{w_{t,i-1} + w_{t,i} + w_{t,i+1}}{3} - \gamma \nabla f(w_{t,i}; b_i)$$

References

- https://dl.acm.org/doi/pdf/10.14778/3415478.3415530
- https://pytorch.org/tutorials/beginner/ddp_series_theory.html
- https://ethz.ch/content/dam/ethz/special-interest/infk/inst-cp/ds3lab-dam/documents/ZipML.pdf