LLM Sys

11868 LLM Systems

Distributed Data Parallel Training

Lei Li

Carnegie Mellon University
Language Technologies Institute

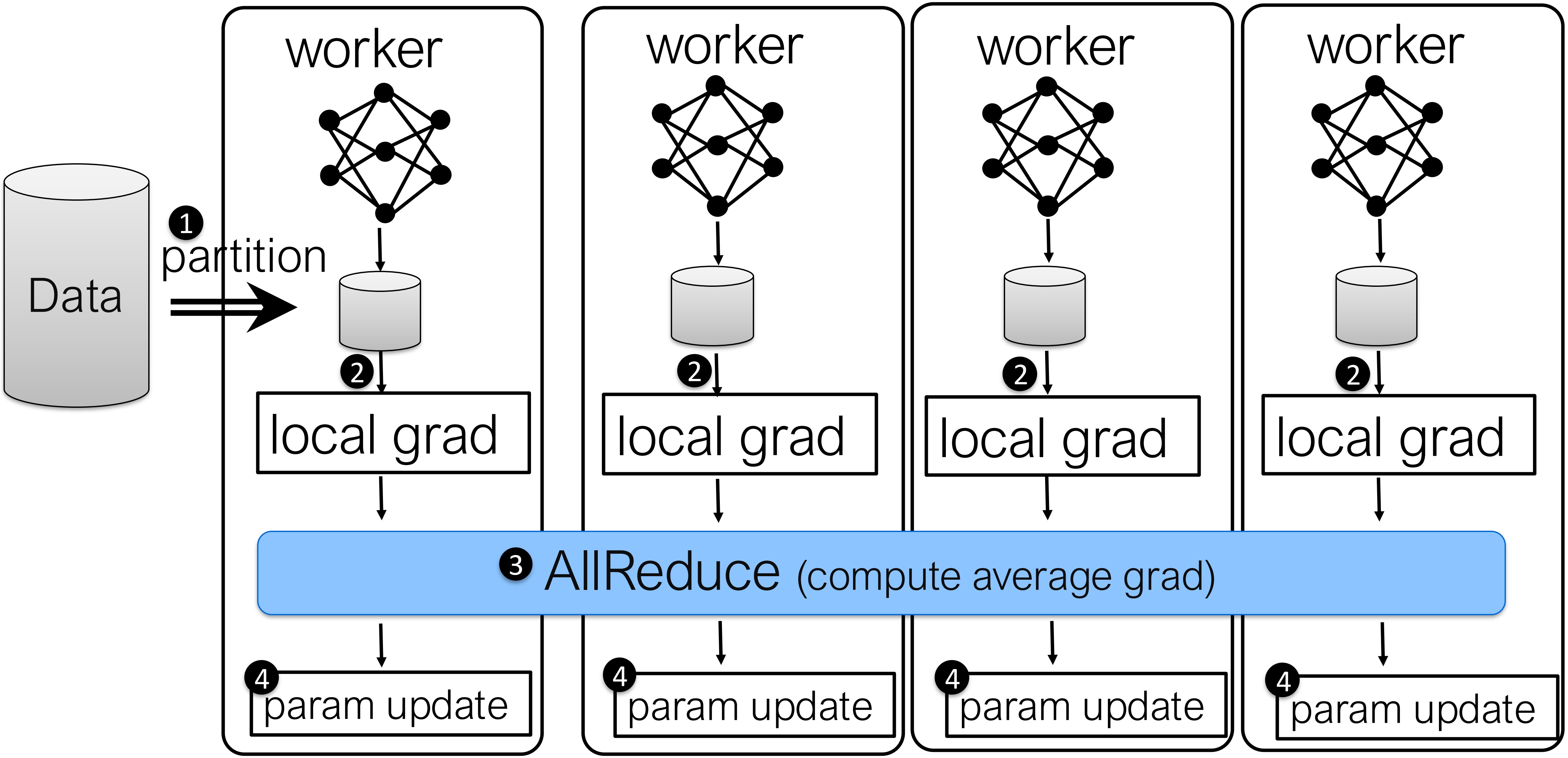
Recap

- Overall idea: partition the data, distribute the forward/backward
- Parameter Server
    - server to update and distribute parameters, worker to get local grad
- NCCL Multi-GPU communication
    - using ring and batching to reduce the latency for Broadcast
- Data Parallel via All Reduce
    - Efficient Ring AllReduce (ScatterReduce+AllGather)

NCCL Primitives

- Broadcast
- Reduce
- ReduceScatter
- AllGather
- AllReduce

Data Parallel



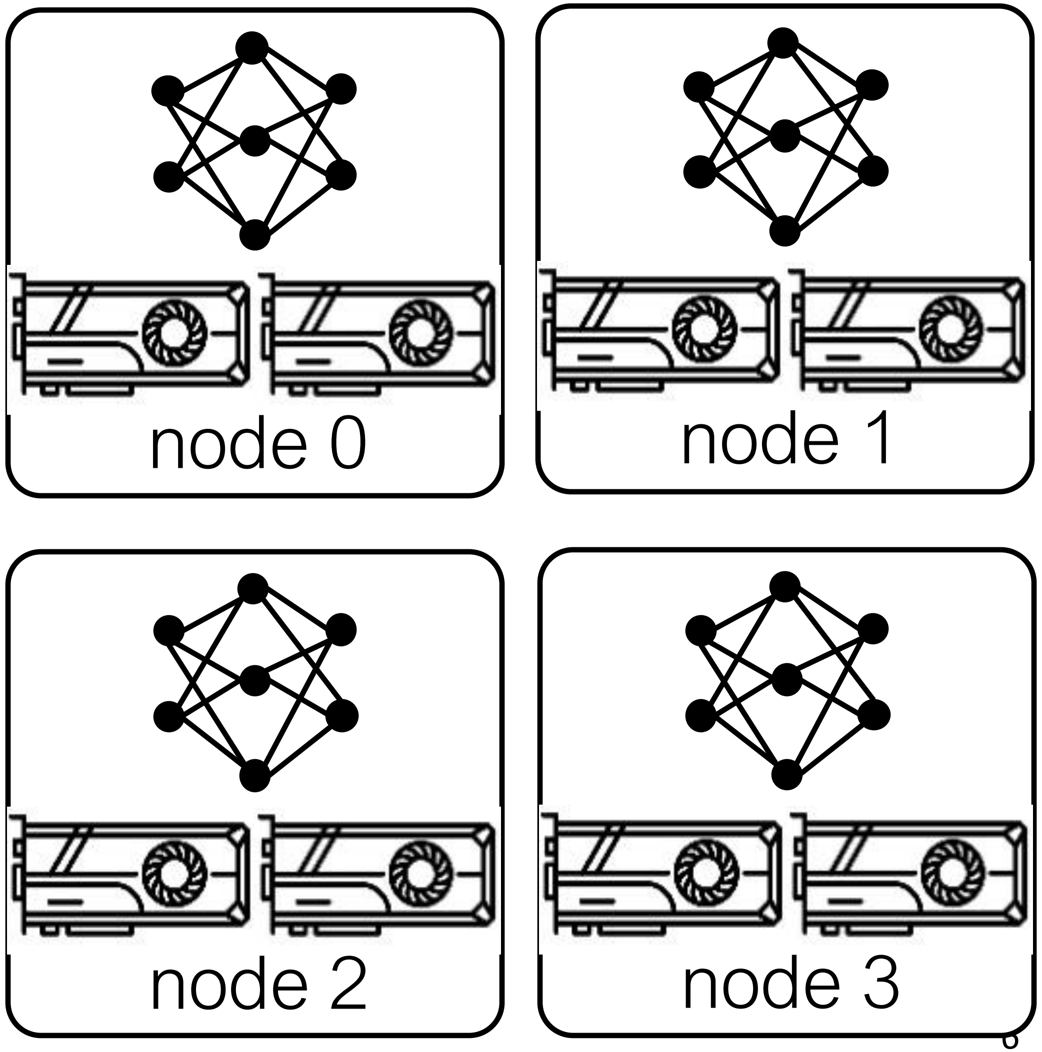
Outline

- Distributed Data Parallel Training
- Design and implementation of Distributed Data Parallel
- Code walkthrough:
    - Using DDP in PyTorch

Distributed Data Parallel

- Same as Data Parallel

- multiple nodes, each with multiple GPUs
  - Create replicas of a model on multiple nodes
  - Each model performs the forward pass and the backward pass independently
  - Gather average gradients across nodes
  - Optimizers run locally (identical)



PyTorch Distributed: Experiences on Accelerating Data Parallel Training. VLDB 2020.

Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, Soumith Chintala

Design Goal of DDP

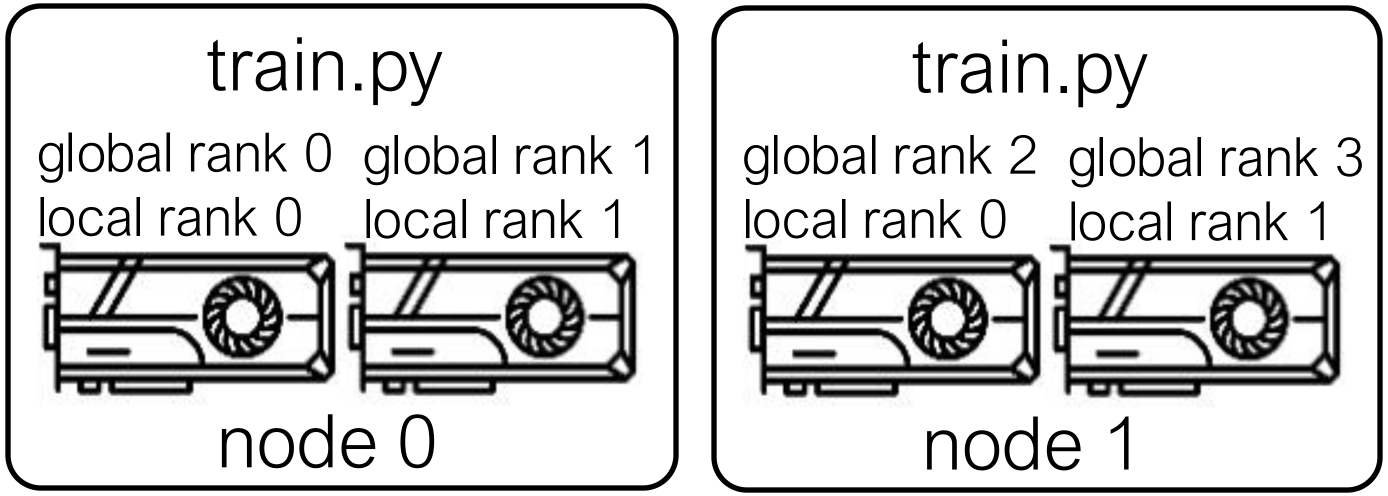
- Non-intrusive: Developers should be able to reuse the local training script with minimal modifications.

- Interceptive: The API needs to allow the implementation to intercept various signals and trigger appropriate algorithms promptly. The API must expose as many optimization opportunities as possible to the internal implementation.

Li et al. PyTorch Distributed: Experiences on Accelerating Data Parallel Training, VLDB 2020.

Setting up the Distributed Process

- World size
  - total number of processes W
- Global rank
  - global process id
- Local rank
  - local process id



Launch Distributed Processes

The launch.py (torch/distributed/launch.py) will pass world size, global rank, master address, master port via env vars, and local rank as a commandline parameter to every instance

Env Vars: "MASTER_ADDR", "MASTER_PORT", "RANK", "WORLD_SIZE"

```
if __name__ == "__main__":
  parser = argparse.ArgumentParser()
  parser.add_argument("--local_rank", type=int, default=0)
  parser.add_argument("--local_world_size", type=int, default=1) args =
parser.parse_args()
  local_proc(args.local_world_size, args.local_rank)
```

Launching Local Process

```
def local_proc(local_world_size, local_rank):
  dist.init_process_group(backend="nccl")

  local_train(local_world_size, local_rank)

  dist.destroy_process_group()
```

start process group

tear down process group

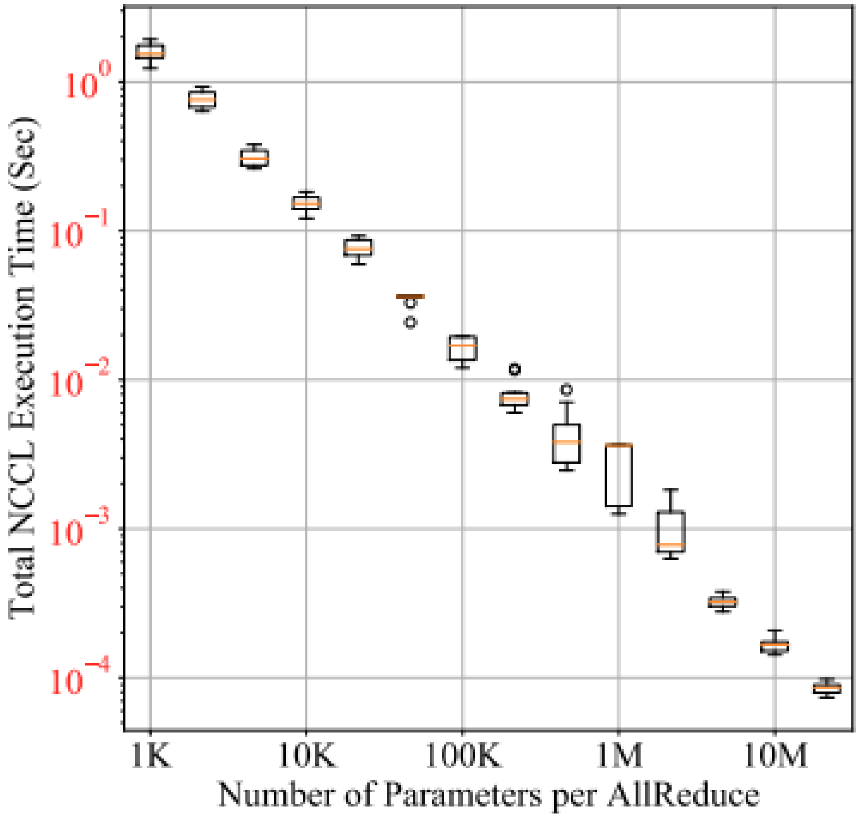
```
def demo_basic(local_world_size, local_rank):
    n = torch.cuda.device_count() // local_world_size
    device_ids = list(range(local_rank * n, (local_rank + 1) * n))
    model = MyModel().cuda(device_ids[0])
    ddp_model = DDP(model, device_ids)
    loss_fn = nn.MSELoss()
    optimizer = optim.SGD(ddp_model.parameters(), lr=0.001)
    optimizer.zero_grad()
    outputs = ddp_model(torch.randn(20, 10))
    labels = torch.randn(20, 5).to(device_ids[0])
    loss_fn(outputs, labels).backward()
    optimizer.step()
```

How to Implement Distributed Data Parallel

- Naïve solution: synchronize gradients after the *entire* backward pass finishes
  - What can be improved?

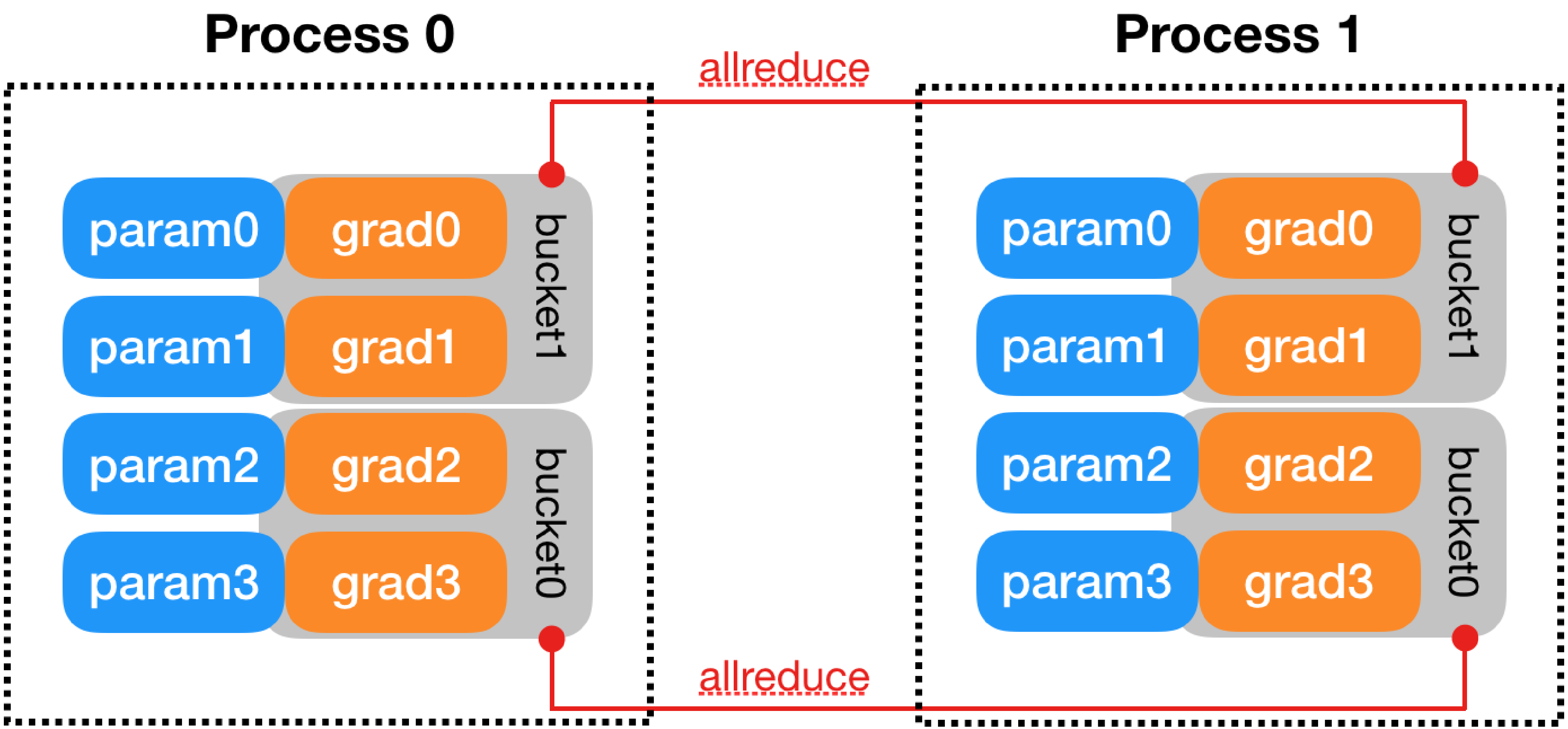
Implementing Distributed Data Parallel

- Naïve solution: synchronize gradients after the *entire* backward pass finishes
  - We can overlap gradient computation and synchronization!
- But how often should we synchronize? Per parameter?
  - Too much synchronization slows down execution



(a) NCCL

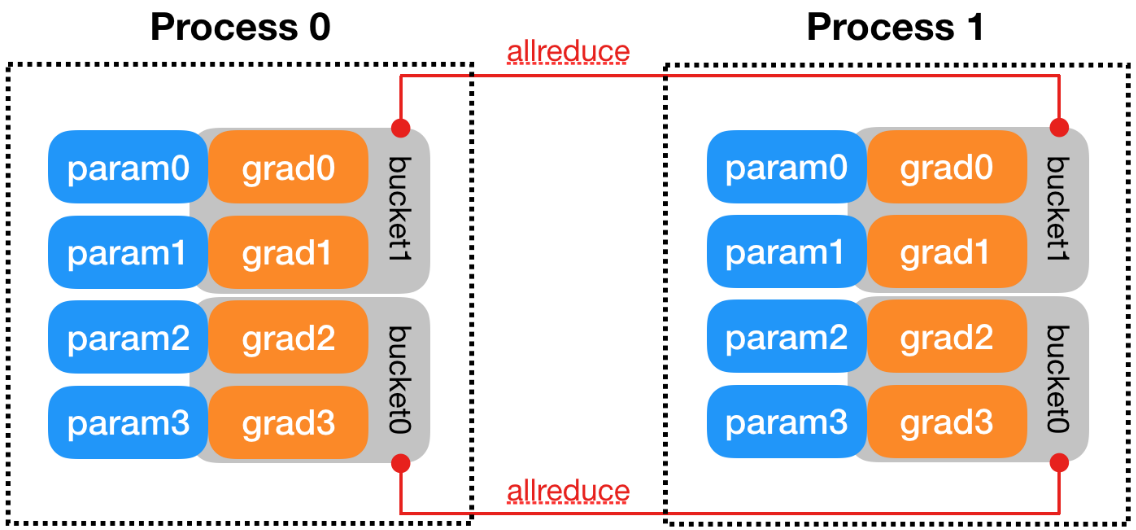
Gradient Bucketing



Gradient Bucketing

- Bucket size can be configured by setting the **bucket_cap_mb** argument in DDP constructor.

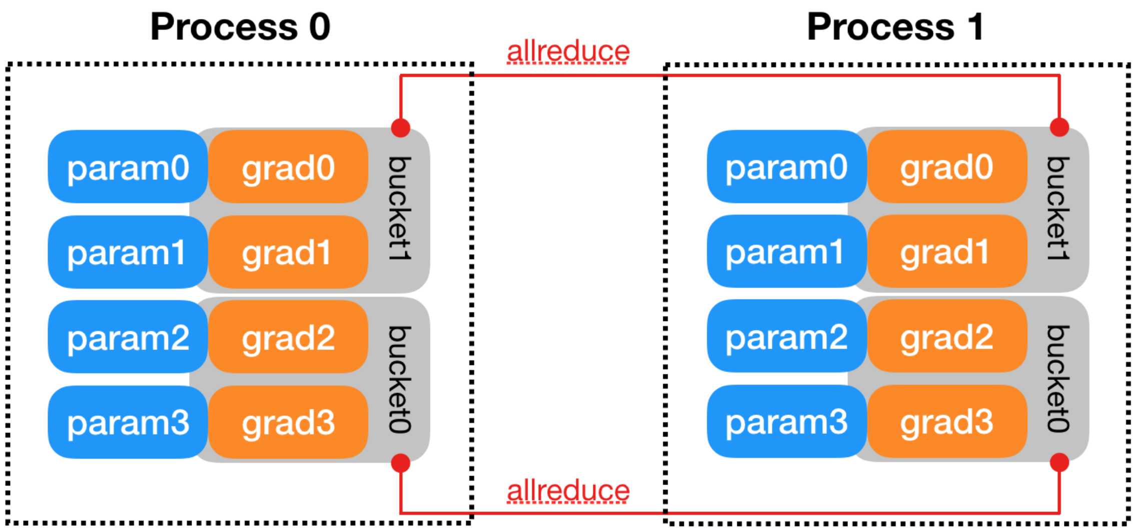
- The mapping from parameter gradients to buckets is determined at the construction time, based on the bucket size limit and parameter sizes.



Gradient Bucketing

- Model parameters are allocated into buckets in (roughly) the reverse order of Model.parameters() from the given model.

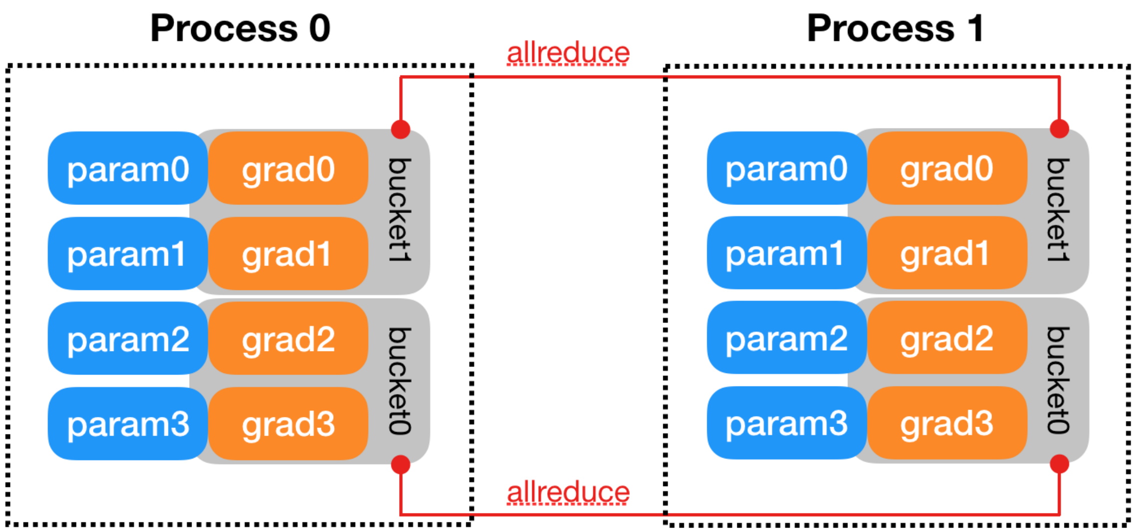
- DDP expects gradients to become ready during the backward pass in approximately that order.



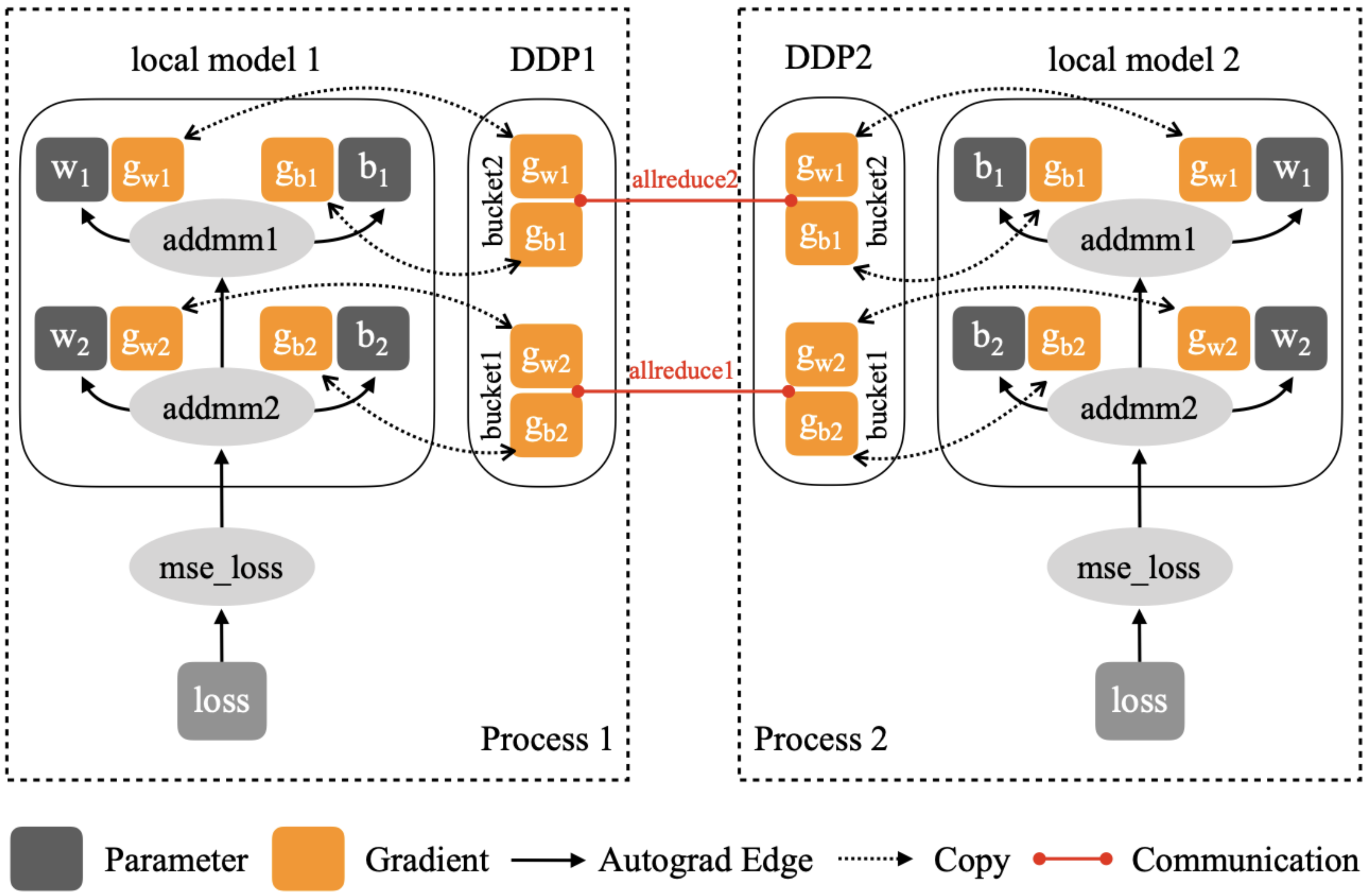
Gradient Bucketing

- When gradients in one bucket are all ready, the Reducer kicks off an asynchronous allReduce on that bucket to calculate average of gradients across all processes.

- Overlapping computation (backward) with communication (AllReduce)



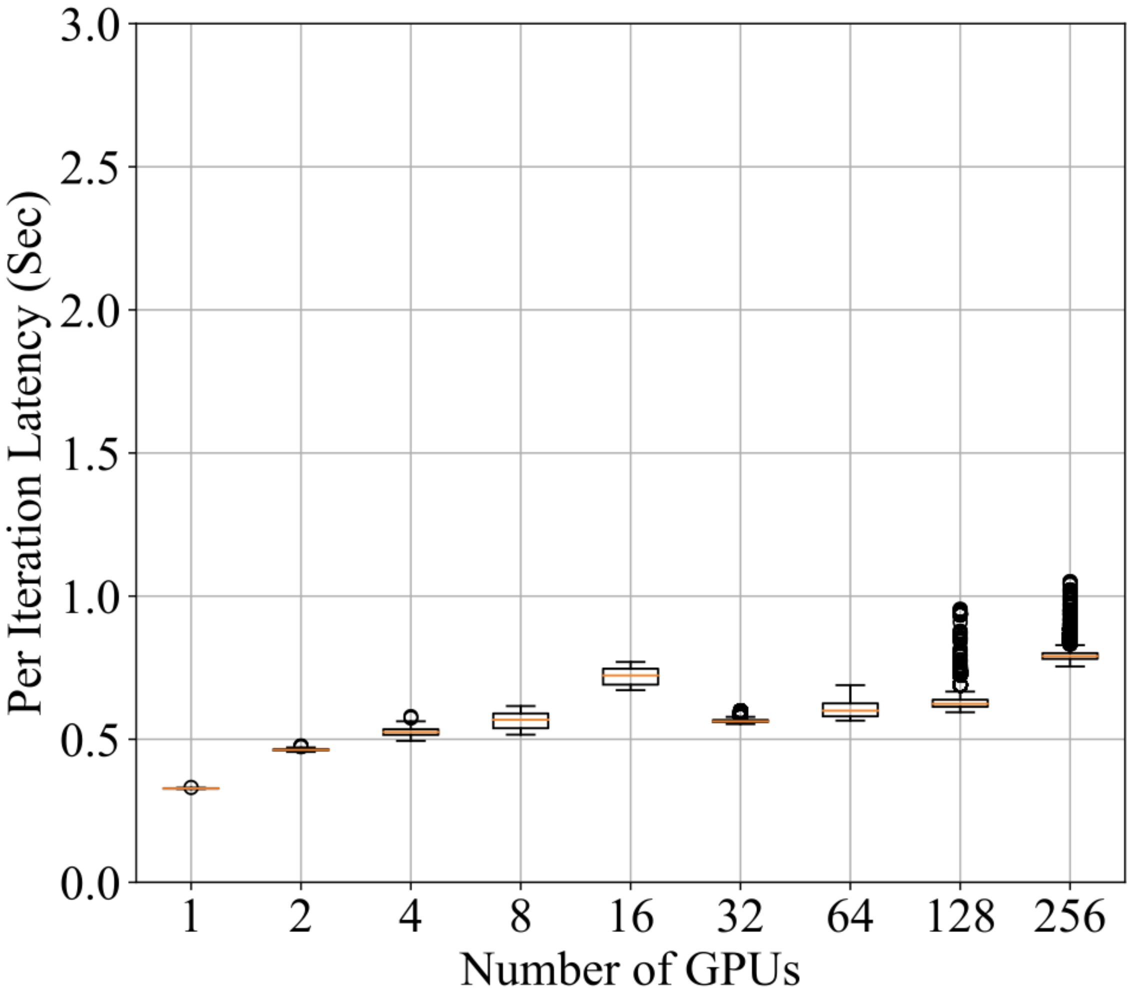
Gradient Reduction



DDP Implementation

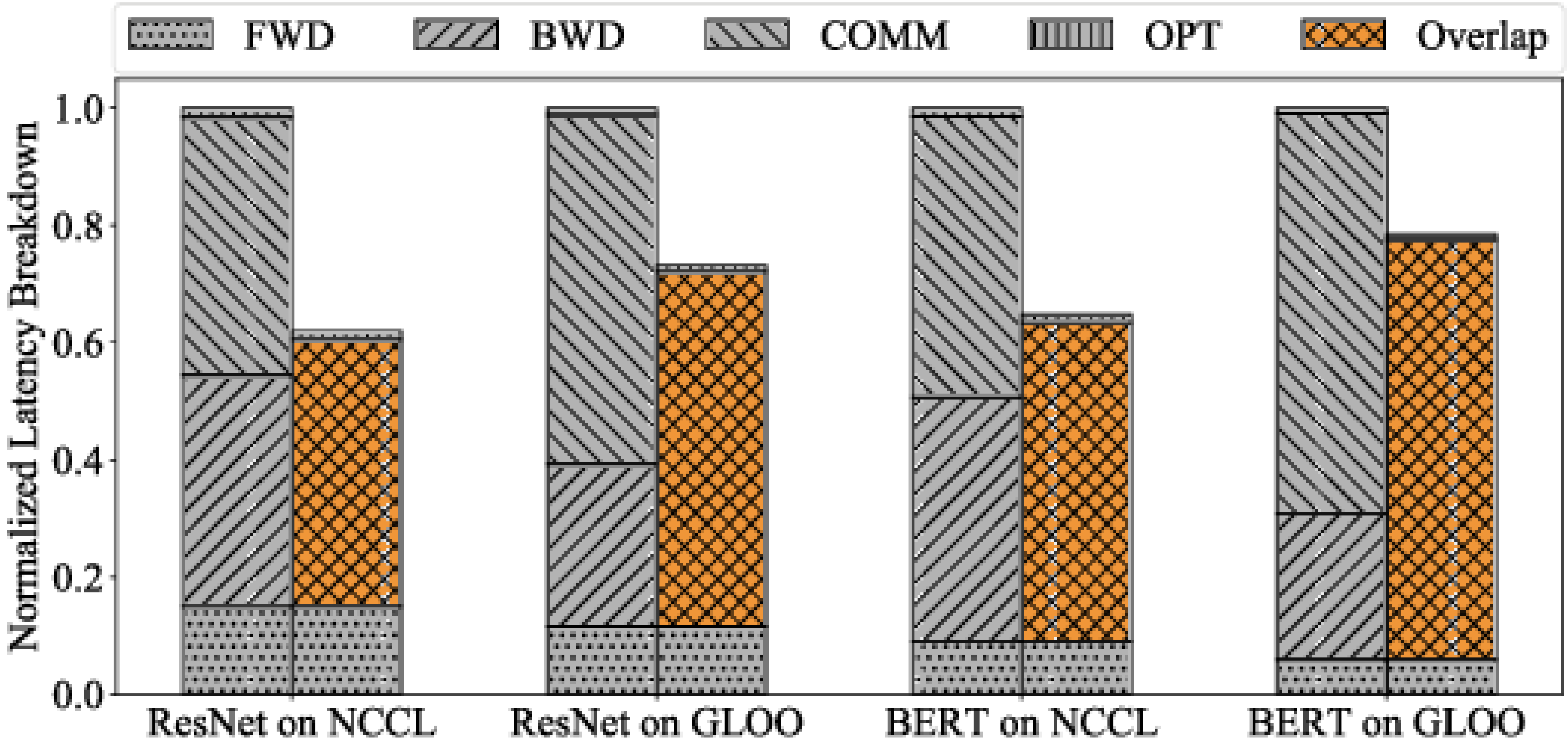
```
// The function `autograd_hook` is called after the gradient for a
// model parameter has been accumulated into its gradient tensor.
// This function is only to be called from the autograd thread.
void Reducer::autograd_hook(size_t index) {
        mark_variable_ready(index);
}

void Reducer::mark_variable_ready(size_t variable_index) {
        const auto& bucket_index = variable_locators_[variable_index];
        auto& bucket = buckets_[bucket_index.bucket_index];

        if (--bucket.pending == 0) {
                mark_bucket_ready(bucket_index.bucket_index);
        }
}

void Reducer::mark_bucket_ready(size_t bucket_index) {
        for (; next_bucket_ < buckets_.size() && buckets_[next_bucket_].pending == 0; next_bucket_++) {
                num_buckets_ready_++;
                auto& bucket = buckets_[next_bucket_];
                all_reduce_bucket(bucket);
        }
}

void Reducer::all_reduce_bucket(Bucket& bucket) {
        auto variables_for_bucket = get_variables_for_bucket(next_bucket_, bucket);
        const auto& tensor = bucket.gradients;
        GradBucket grad_bucket(next_bucket_, buckets_.size(), tensor, bucket.offsets,
                bucket.lengths, bucket.sizes_vec, variables_for_bucket);
        bucket.future_work = run_comm_hook(grad_bucket);
}
```

DDP Scalability



(c) BERT on NCCL

DDP Reduces Latency by Overlapping Communication and Computation



bwd & comm

Figure 6: Per Iteration Latency Breakdown

Code walkthrough

Quiz 7

- on Canvas.

Summary

- Data Parallel via All Reduce
- Distributed Data Parallel Training
  - gradient bucketing
  - overlay backward and AllReduce communication

Reading for next lecture

- Huang et al. GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism. 2018

- Shoeybi et al. Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism. 2019

- Narayanan et al. Efficient Large-Scale Language Model Training on GPU Clusters Using Megatron-LM, SC 2021