CSE599-O Assignment 2: Scaling with Data Parallelism

Version 1.0

CSE 599-O Staff

Fall 2025

Acknowledgment: This is partially adapted from Assignment 2 of Stanford CS336 (Spring 2025).

1 Assignment Overview

In this assignment, you will gain some hands-on experience with scaling training to multiple GPUs.

1.1 What you will implement.

1. NCCL Test
2. Distributed data parallel training with overlapping optimizations
3. Optimizer state sharding

1.2 What the code looks like

All the assignment code as well as this writeup are available on GitHub at:

`https://github.com/uw-syfi/assignment2-systems`

Please git clone the repository. If there are any updates, we will notify you and you can git pull to get the latest.

1. cse599o-basics/: In this assignment, you'll be profiling some of the components that we built in assignment 1. This folder contains the solution code for assignment 1, so you will find a cse599o-basics/pyproject.toml and a cse599o-basics/cse599o_basics/* module in here. If you want to use your own implementation of the model (highly encouraged!), you can modify the pyproject.toml file in the base directory to point to your own package.
2. /: The cse599o-systems base directory. We created an empty module named cse599o_systems. Note that there's no code in here, so you should be able to do whatever you want from scratch.
3. tests/*.py: This contains all the tests that you must pass. These tests invoke the hooks defined in tests/adapters.py. You'll implement the adapters to connect your code to the tests. Writing more tests and/or modifying the test code can be helpful for debugging your code, but your implementation is expected to pass the original provided test suite.

4. README.md: This file contains more details about the expected directory structure, as well as some basic instructions on setting up your environment.

1.3 How to submit

You will submit the following files to Gradescope:

- code.zip: Contains all the code you've written.
- Submit your analytical answers (texts and figures) on Gradescope.

Run the script in test_and_make_submission.sh to create the code.zip file.

1.4 Please use Slurm!

For HW2, multi-GPU training is common. Therefore, all students are required to use `Slurm` to schedule their jobs on the Tomago or Tempura machines to ensure efficient resource coordination among students. Each student is assigned to one node according to the spreadsheet on Ed. For HW2, you may use up to 8 GPUs on that node (not limited to the specific device IDs assigned to you). We have already provided an introduction to Slurm in the HW1 Part 2 write-up. If you would like to review it again, please refer to Section 1.4 of HW1 Part 2.

1.5 Model Sizing

Throughout this assignment, we will set the model size as described below; other parameters will be discussed later.

d_model	d_ff	num_layers	num_heads
1280	5120	36	20

Table 1: Model Size Specification

1.6 Grading

There are 10 problems in total, each accounting for 10% of the total grade for this assignment.

2 Distributed Data Parallel Training

In this part, we'll explore methods for using multiple GPUs to train our language models, focusing on data parallelism. We'll start with a primer on distributed communication in PyTorch. Then, we'll study a naive implementation of distributed data parallel training and then implement and benchmark various improvements for improving communication efficiency.

2.1 Single-Node Distributed Communication in PyTorch

Let's start by looking at a simple distributed application in PyTorch, where the goal is to generate four random integer tensors and compute their sum.

In the distributed case below, we will spawn four worker processes, each of which generates a random integer tensor. To sum these tensors across the worker processes, we will call the `all-reduce` collective communication operation, which replaces the original data tensor on each process with the all-reduced result (i.e., the sum).

Now let's take a look at some code.

```python
import os
import torch
import torch.distributed as dist
import torch.multiprocessing as mp

def setup(rank, world_size):
    os.environ["MASTER_ADDR"] = "localhost"
    os.environ["MASTER_PORT"] = "29500"
    # Initialize NCCL for GPU-based distributed training
    dist.init_process_group(backend="nccl", rank=rank, world_size=world_size)
    torch.cuda.set_device(rank)

def cleanup():
    dist.destroy_process_group()

def distributed_demo(rank, world_size):
    setup(rank, world_size)

    # Create data tensor on this rank' s GPU
    data = torch.randint(0, 10, (3,), device=f"cuda:{rank}")
    print(f"Rank {rank} data (before all-reduce): {data}")

    # Perform all-reduce across GPUs
    dist.all_reduce(data)
    print(f"Rank {rank} data (after all-reduce):  {data}")

    cleanup()

if __name__ == "__main__":
    world_size = torch.cuda.device_count()  # auto-detect GPUs
    if world_size < 2:
        raise RuntimeError("Need at least 2 GPUs to run this example.")
    mp.spawn(distributed_demo, args=(world_size,), nprocs=world_size, join=True)
```

After running the script above, we get the output below. As expected, each worker process initially holds different data tensors. After the all-reduce operation, which sums the tensors across all of the worker processes, data is modified in-place on each of the worker processes to hold the all-reduced result. [1]

```
$ srun --gpus-per-node=8 uv run python distributed_hello_world.py
Rank 2 data (before all-reduce): tensor([3, 2, 4], device='cuda:2')
Rank 3 data (before all-reduce): tensor([5, 2, 9], device='cuda:3')
Rank 4 data (before all-reduce): tensor([0, 4, 7], device='cuda:4')
Rank 7 data (before all-reduce): tensor([2, 2, 7], device='cuda:7')
Rank 1 data (before all-reduce): tensor([6, 6, 6], device='cuda:1')
Rank 5 data (before all-reduce): tensor([5, 4, 0], device='cuda:5')
Rank 0 data (before all-reduce): tensor([6, 9, 5], device='cuda:0')
Rank 6 data (before all-reduce): tensor([4, 6, 3], device='cuda:6')
Rank 3 data (after all-reduce):  tensor([31, 35, 41], device='cuda:3')
Rank 5 data (after all-reduce):  tensor([31, 35, 41], device='cuda:5')
Rank 4 data (after all-reduce):  tensor([31, 35, 41], device='cuda:4')
Rank 7 data (after all-reduce):  tensor([31, 35, 41], device='cuda:7')
Rank 2 data (after all-reduce):  tensor([31, 35, 41], device='cuda:2')
Rank 0 data (after all-reduce):  tensor([31, 35, 41], device='cuda:0')
Rank 1 data (after all-reduce):  tensor([31, 35, 41], device='cuda:1')
Rank 6 data (after all-reduce):  tensor([31, 35, 41], device='cuda:6')
```

Let's now look back more closely at our script above. The command mp.spawn spawns nprocs processes that run fn with the provided args. In addition, the function fn is called as fn(rank, $*args$), where rank is the index of the worker process (a value between 0 and nprocs-1). Thus, our distributed_demo function must accept this integer rank as its first positional argument. In addition, we pass in the world_size, which refers to the total number of worker processes.

Each of these worker processes belong to a process group, which is initialized via dist.init_process_group. The process group represents multiple worker processes that will coordinate and communicate via a shared master. The master is defined by its IP address and port, and the master runs the process with rank 0. Collective communication operations like all-reduce operate on each process in the process group.

In this case, we initialized our process group with the "nccl" backend, which uses NVIDIA' s NCCL collective communication library. This backend is optimized for GPU-based distributed training and provides the best performance when working with CUDA tensors. If you want to run on CPU-only machines, please use the "gloo" backend instead. As a general rule of thumb, use NCCL for GPU training and Gloo for CPU training or local development. We are using NCCL in this example because our setup runs on GPUs.

[1] If you run this script multiple times, you'll notice that the order of the printed output is not deterministic. Since this application is running in a distributed setting, we cannot control the exact order in which commands are being run-our only guarantee is that after the all-reduce operation is complete, the separate processes will hold bitwise identical result tensors.

When running multi-GPU jobs, make sure that different ranks use different GPUs. One method for doing this is to call torch.cuda.set_device(rank)) in the setup function, so that tensor.to("cuda") will automatically move it to the specified device. Alternatively, you can explicitly create a per-rank device string (e.g., device = f"cuda:{rank}"), and then use this device string as the target device for any data movement (e.g., tensor.to(f"cuda:{rank}")).

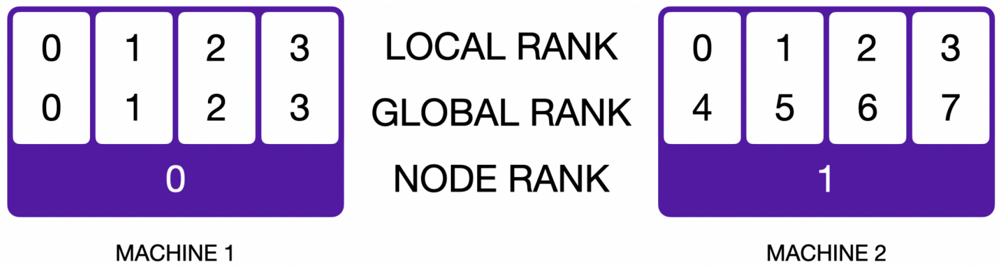

Figure 1: A schematic representation of a distributed application running on 2 nodes with a world size of 8. Each worker process is identified by a global rank (from 0 to 7) and a local rank (from 0 to 3). Figure taken from `https://lightning.ai/docs/fabric/stable/advanced/distributed_communication.html`

Terminology. In the rest of the assignment (and various other resources you might see online), you may encounter the following terms in the context of PyTorch distributed communication. Though we will focus on single-node, multi-process distributed training in this assignment, the terminology is useful for understanding distributed training in general. See Figure 1 for a visual representation.

- **Node:** A machine on the network.
- **Worker:** An instance of a program that participates in distributed training. In this assignment, each worker runs a single process, so we use worker, process, and worker process interchangeably. However, in general, a worker may spawn multiple processes (e.g., for data loading), so these terms are not always equivalent in practice.
- **World size:** The total number of workers in a process group.
- **Global rank:** An integer ID (between 0 and `world_size-1`) that uniquely identifies a worker within the process group. For example, with a world size of two, one process will have global rank 0 (the master process), and the other will have rank 1.
- **Local world size:** When running applications across multiple nodes, the local world size is the number of workers running locally on a given node. For example, if an application spawns four workers on each of two nodes, the global world size is 8, and the local world size on each node is 4. Note that when running on a single node, the local world size equals the global world size.
- **Local rank:** An integer ID (between 0 and `local_world_size-1`) that uniquely identifies the index of a local worker on a machine. For example, if an application spawns four processes on each of two nodes, each node will have workers with local ranks 0, 1, 2, and 3. Note that in a single-node multi-process setup, the local rank of a process is equivalent to its global rank.

2.2 Benchmarking Distributed Applications

Throughout this portion of the assignment you will be benchmarking distributed applications to better understand the overhead from communication. Here are a few best practices:

- Whenever possible, run benchmarks on the same machine to facilitate controlled comparisons.
- Perform several warm-up steps before timing the operation of interest. This is especially important for NCCL communication calls. 5 iterations of warmup is generally sufficient.
- Call torch.cuda.synchronize() to wait for CUDA operations to complete when benchmarking on GPUs. Note that this is necessary even when calling communication operations with async_op=False, which returns when the operation is queued on the GPU (as opposed to when the communication actually finishes). [2]
- Timings may vary slightly across different ranks, so it's common to aggregate measurements across ranks to improve estimates. You may find the all-gather collective (specifically the dist.all_gather_object function) to be useful for collecting results from all ranks.
- In general, you can debug locally with Gloo on CPU, and then as required in a given problem, benchmark with NCCL on GPU. Switching between the backends just involves changing the init_process_group call and tensor device casts.

Problem (distributed_communication_single_node)

Write a script to benchmark the runtime of the all-reduce operation in the single-node multi-process setup. The example code above may provide a reasonable starting point. Experiment with varying the following settings:

- **Backend + device type:** NCCL + GPU.
- **All-reduce data size:** float32 data tensors ranging over 1MB, 10MB, 100MB, 1GB.
- **Number of processes:** 2, 4, or 8 processes.
- **Resource requirements:** Up to 8 GPUs. Each benchmarking run should take less than 5 minutes.

Deliverable: Plot(s) and/or table(s) comparing the various settings, with 2–3 sentences of commentary about your results and thoughts about how the various factors interact.

2.3 A Naïve Implementation of Distributed Data Parallel Training

Now that we've seen the basics of writing distributed applications in PyTorch, let's build a minimal implementation of distributed data parallel (DDP) training.

[2] See `github.com/pytorch/pytorch/issues/68112#issuecomment-965932386` for more details.

Data parallelism splits batches across multiple devices (e.g., GPUs), enabling training on large batch sizes that do not fit on a single device. For example, given four devices that can each handle a maximum batch size of 32 , data parallel training would enable an effective batch size of 128 .

Here are the steps for naïvely doing distributed data parallel training. Initially, each device constructs a (randomly-initialized) model. We use the broadcast collective communication operation to send the model parameters from rank 0 to all other ranks. At the start of training, each device holds an identical copy of the model parameters and optimizer states (e.g. the accumulated gradient statistics in Adam).

1. Given a batch with n examples, the batch is sharded and each device receives n/d disjoint examples (where d is the number of devices used for data parallel training). n should divide d, since the training time is bottlenecked by the slowest process.

2. Each device uses its local copy of the model parameters to run a forward pass on its n/d examples and a backward pass to calculate the gradients. Note that at this point, each device holds the gradients computed from the n/d examples it received.

3. We then use the all-reduce collective communication operation to average the gradients across the different devices, so each device holds the gradients averaged across all n examples.

4. Next, each device runs an optimizer step to update its copy of the parameters-from the optimizer's perspective, it is simply optimizing a local model. The parameters and optimizer states will stay in sync on all of the different devices since they all start from the same initial model and optimizer state and use the same averaged gradients for each iteration. At this point, we've completed a single training iteration and can repeat the process.

Problem (toymodel_naive_ddp)

Deliverable: Write a script (`naive_ddp.py`) to naively perform distributed data parallel training by all-reducing individual parameter gradients after the backward pass. To verify the correctness of your DDP implementation, use it to train a small toy model on randomly-generated data and verify that its weights match the results from single-process training. You can directly use the `ToyModel` from `tests/common.py`. Use a small number of iterations (e.g., <10) to minimize runtime. The course staff will execute your script to check correctness. [a]

[a] If you're having trouble writing this test, it might be useful to look at `tests/test_ddp_individual_parameters.py`. Notably, you do not need to support testing `naive_ddp` with `tests/test_ddp_individual_parameters.py`, as that file is reserved for later, more complex implementations.

Problem (transformer_naive_ddp_benchmarking)

In this naïve DDP implementation, parameters are individually all-reduced across ranks after each backward pass. To better understand the overhead of data-parallel training, extend `naive_ddp.py` beyond the `ToyModel` to train your previously implemented language model using this naïve DDP mechanism. Measure the total time per training step and the proportion of time spent on communicating gradients. It is reasonable to use a small number of iterations instead of long-term training for this test. Collect measurements in the single-node setting (1 node $\times$ 2 GPUs) for the model size described in Table 1.

Deliverable: A benchmark script (naive_ddp.py) to collect timing results. On Gradescope, provide your analytical answers supported by the measured per-iteration training time and gradient communication time.

2.4 Improving Upon the Minimal DDP Implementation

The minimal DDP implementation that we saw in last section has a couple of key limitations:

1. It conducts a separate all-reduce operation for every parameter tensor. Each communication call incurs overhead, so it may be advantageous to batch communication calls to minimize this overhead.
2. It waits for the backward pass to finish before communicating gradients. However, the backward pass is incrementally computed. Thus, when a parameter gradient is ready, it can immediately be communicated without waiting for the gradients of the other parameters. This allows us to overlap communication of gradients with computation of the backward pass, reducing the overhead of distributed data parallel training.

In this part of the assignment, we'll address each of these limitations in turn and measure the impact on training speed.

2.4.1 Reducing the Number of Communication Calls

Rather than issuing a communication call for each parameter tensor, let see if we can improve performance by batching the all-reduce. Concretely, we'll take the gradients that we want to all-reduce, concatenate them into a single tensor, and then all-reduce the combined gradients across all ranks. It might be helpful to use torch._utils._flatten_dense_tensors and torch._utils._unflatten_dense_tensors.

Problem (minimal_ddp_flat_benchmarking)

Modify your minimal DDP implementation to communicate a tensor with flattened gradients from all parameters. Compare its performance with the minimal DDP implementation that issues an all-reduce for each parameter tensor under the previously used conditions (1 node × 2 GPUs, model size as described in Table 1).

Deliverable: Extend your `naive_ddp.py` to support flat_ddp. On Gradescope, provide your analytical answers with the measured time per training iteration and time spent communicating gradients under distributed data parallel training with a single batched all-reduce call.

2.4.2 Overlapping Computation with Communication of Individual Parameter Gradients

While batching the communication calls might help lower the overhead associated with issuing a large number of small all-reduce operations, all of the communication time still directly contributes to the overhead. To resolve this, we can take advantage of the observation that the backward pass incrementally computes gradients for each layer (starting from the loss and moving toward the input)-thus, we can all-reduce parameter gradients as soon as they're ready, reducing the overhead of data parallel training by overlapping computation of the backward pass with communication of gradients.

We'll start by implementing and benchmarking a distributed data parallel wrapper that asynchronously all-reduces individual parameter tensors as they become ready during the backward pass. The following pointers may be useful:

Backward hooks: To automatically call a function on a parameter after its gradient has been accumulated in the backward pass, you can use the register_post_accumulate_grad_hook function. [3]

Asynchronous communication: All PyTorch collective communication operations support synchronous (async_op=False) and asynchronous execution (async_op=True). Synchronous calls will block until the collective operation is queued on the GPU. This does not mean that the CUDA operation is completed since CUDA operations are asynchronous. That being said, later function calls using the output will behave as expected. [4]. In contrast, asynchronous calls will return a distributed request handle - as a result, when the function returns, the collective communication operation is not guaranteed to have been queued on GPU, let alone completed. To wait for the operation to be queued on GPU (and therefore for the output to be usable in later operations), you can call handle.wait() on the returned communication handle.

For example, the following two examples all-reduce each tensor in a list of tensors with either a synchronous or an asynchronous call:

```python
tensors = [torch.rand(5) for _ in range(10)]

# Synchronous: block until operation is queued on GPU.
for tensor in tensors:
    dist.all_reduce(tensor, async_op=False)

# Asynchronous: return immediately after each call and
# wait on results at the end.
handles = []
for tensor in tensors:
    handle = dist.all_reduce(tensor, async_op=True)
    handles.append(handle)

# Possibly execute other commands that don't rely on the all_reduce results.
# ...

# Ensure that all-reduce calls were queued so that
# operations depending on the results can proceed.
for handle in handles:
    handle.wait()
handles.clear()
```

[3]See pytorch.org/docs/stable/generated/torch.Tensor.register_post_accumulate_grad_hook.html for more information and usage examples.

[4]In advanced cases, if you are using multiple CUDA streams, you may need explicit synchronization across streams to ensure that the output is ready for later operations. See `http://pytorch.org/docs/stable/notes/cuda.html#cuda-streams`

Problem (ddp_overlap_individual_parameters)

Implement a Python class to handle distributed data parallel training. The class should wrap an arbitrary PyTorch `nn.Module` and take care of broadcasting the weights before training (so all ranks have the same initial parameters) and issuing communication calls for gradient averaging. We recommend the following public interface:

- `def __init__(self, module: torch.nn.Module):` Given an instantiated PyTorch `nn.Module` to be parallelized, construct a DDP container that will handle gradient synchronization across ranks.
- `def forward(self, *inputs, **kwargs):` Calls the wrapped module's `forward()` method with the provided positional and keyword arguments.
- `def finish_gradient_synchronization(self):` When called, wait for asynchronous communication calls to be queued on the GPU.

To use this class to perform distributed training, we'll pass it a module to wrap, and then add a call to `finish_gradient_synchronization()` before we run `optimizer.step()` to ensure that the optimizer step—an operation that depends on the gradients—may be queued:

```
model = ToyModel().to(device)
ddp_model = DDP(model)
for _ in range(train_steps):
    x, y = get_batch()
    logits = ddp_model(x)
    loss = loss_fn(logits, y)
    loss.backward()
    ddp_model.finish_gradient_synchronization()
    optimizer.step()
```

Deliverable: Implement a container class to handle distributed data parallel training. This class should overlap gradient communication and the computation of the backward pass. To test your DDP class, first implement the adapters `adapters.get_ddp_individual_parameters` and `adapters.ddp_individual_parameters_on_after_backward` (the latter is optional, depending on your implementation).
Then, to execute the tests, run:

```
uv run pytest tests/test_ddp_individual_parameters.py
```

We recommend running the tests multiple times (e.g., 5) to ensure that it passes reliably.

Problem (ddp_overlap_individual_parameters_benchmarking)

(a) Create a script (individual_ddp.py) to use the container class you implemented to wrap the transformer model. Benchmark the performance of your DDP implementation when overlapping backward pass computation with communication of individual parameter gradients. Compare its performance with our previously studied settings (the minimal DDP implementation that either issues an all-reduce for each parameter tensor, or a single all-reduce on the concatenation of all parameter tensors) under the same setup: 1 node, 2 GPUs, and the model size in Table 1.

Deliverable: The benchmark script (individual_ddp.py). The measured time per training iteration when overlapping the backward pass with communication of individual parameter gradients.

(b) Instrument your benchmarking code with the Nsight profiler, comparing between the initial DDP implementation and this DDP implementation that overlaps backward computation and communication. Visually compare the two traces, and provide a profiler screenshot demonstrating that one implementation overlaps compute with communication while the other does not.

Deliverable: Two screenshots (one from the initial DDP implementation and another from this overlapping DDP implementation) that visually show whether communication is overlapped with the backward pass.

2.4.3 Overlapping Computation with Communication of Bucketed Parameter Gradients

In the previous section, we overlapped backprop computation with the communication of individual parameter gradients. However, we previously observed that batching communication calls can improve performance, especially when we have many parameter tensors (as is typical in deep Transformer models). Our previous attempt at batching sent all of the gradients at once, which requires waiting for the backward pass to finish. In this section, we'll try to get the best of both worlds by organizing our parameters into buckets (reducing the number of total communication calls) and all-reducing buckets when each of their constituent tensors are ready (enabling us to overlap communication with computation).

Problem (ddp_overlap_bucketed)

Implement a Python class to handle distributed data parallel training, using **gradient bucketing** to improve communication efficiency. The class should wrap an arbitrary input PyTorch `nn.Module` and take care of broadcasting the weights before training (so all ranks have the same initial parameters) and issuing bucketed communication calls for gradient averaging. We recommend the following interface:

- `def __init__(self, module: torch.nn.Module, bucket_size_mb: float):` Given an instantiated PyTorch `nn.Module` to be parallelized, construct a DDP container that will handle gradient synchronization across ranks. Gradient synchronization should be bucketed, with each bucket holding at most `bucket_size_mb` of parameters.

- `def forward(self, *inputs, **kwargs):` Calls the wrapped module's `forward()` method with the provided positional and keyword arguments.

- `def finish_gradient_synchronization(self):` When called, wait for asynchronous communication calls to be queued on the GPU.

Beyond the addition of a `bucket_size_mb` initialization parameter, this public interface matches that of our previous DDP implementation that individually communicated each parameter. We suggest allocating parameters to buckets using the reverse order of `model.parameters()`, since the gradients will become ready in approximately that order during the backward pass.

Deliverable: Implement a container class to handle distributed data parallel training. This class should overlap gradient communication and the computation of the backward pass. Gradient communication should be bucketed to reduce the total number of communication calls.

To test your implementation, complete the following adapters: `adapters.get_ddp_bucketed`, `adapters.ddp_bucketed_on_after_backward`, and `adapters.ddp_bucketed_on_train_batch_start` (the latter two are optional depending on your implementation).

Then, to execute the tests, run:

```
pytest tests/test_ddp.py
```

We recommend running the tests multiple times (e.g., 5) to ensure that it passes reliably.

Problem (ddp_bucketed_benchmarking)

Create a script (individual_bucketed_ddp.py) to use the container class you implemented to wrap the transformer model. Benchmark your bucketed DDP implementation using the same configuration as the previous experiments (1 node, 2 GPUs, model size in Table 1), varying the maximum bucket size ($1, 10, 100, 1000$ MB). Compare your results to the previous experiments without bucketing—do the results align with your expectations? If they do not align, why not? You may need to use the PyTorch profiler to better understand how communication calls are ordered and/or executed. What changes in the experimental setup would you expect to yield results that better match your expectations?

Deliverable: A script (individual_bucketed_ddp.py). Measured time per training iteration for various bucket sizes, along with a 3–4 sentence commentary discussing the results, expectations, and possible reasons for any mismatches.

3 4D Parallelism

While much more complex in implementation, there are more axes along which we can parallelize our training process. Most commonly we discuss 5 methods of parallelism:

- Data parallelism (DP) - Batches of data are split across multiple devices, and each device computes gradients for their own batch. These gradients must somehow be averaged across devices.
- Fully-Sharded Data Parallelism (FSDP) - Optimizer states, gradients, and weights are split across devices. If we're using only DP and FSDP, every device needs to gather the weight shards from all other devices before we can perform our forward or backward pass.

- Tensor Parallelism (TP) - Activations are sharded across a new dimension, and each device computes the output results for their own shard. With Tensor Parallel we can either shard along the inputs or the outputs the operation we are sharding. Tensor Parallelism can be used effectively together with FSDP if we shard the weights and the activations along corresponding dimensions.
- Pipeline Parallelism (PP) - The model is split layerwise into multiple stages, where each stage is run on a different device.
- Expert Parallelism (EP) - We separate experts (in Mixture-of-Experts models) onto different devices, and each device computes the output results for their own expert.

Typically, we always combine FSDP and TP, so we can think of them as a single axis of parallelism. This leaves us with 4 axes of parallelism: DP, FSDP/TP, PP, and EP. We will also focus on dense models (not MoEs) and so will not discuss EP further.

When reasoning about distributed training, we often describe our cluster as a mesh of devices, where the axes of the mesh are the axes along which we define our parallelism. For instance, if we have 16 GPUs and a model that is much larger than we can fit on a single device, we might be inclined to organize our mesh into a 4×4 grid of GPUs, where the first dimension represents DP, and the second dimension represents combined FSDP and TP.

See the overview in Part 5 of the TPU Scaling Book (Austin et al. (2025)) for more details on how these methods work and how to derive their communication and memory costs (this will be especially helpful to tackle the problem below). For a more detailed pipeline parallel discussion, see The Ultra-Scale Playbook Appendix Tazi et al. (2025). The rest of this book also has a lot of other information you might find useful.

3.1 Optimizer State Sharding

Distributed data parallel training is conceptually simple and often very effective, but requires each rank to hold a distinct copy of the model parameters and optimizer state. This redundancy can come with significant memory costs. For example, the AdamW optimizer maintains two floats per parameter, meaning that it consumes twice as much memory as the model weights. Rajbhandari et al. (2020) describe several methods for reducing this redundancy in data-parallel training by partitioning the (1) optimizer state, (2) gradients, and (3) parameters across ranks, communicating them between workers as necessary.

In this part of the assignment, we'll reduce per-rank memory consumption by implementing a simplified version of optimizer state sharding. Rather than keeping the optimizer states for all parameters, each rank's optimizer instance will only handle a subset of the parameters (approximately 1 / world_size). When each rank's optimizer takes an optimizer step, it'll only update the subset of model parameters in its shard. Then, each rank will broadcast its updated parameters to the other ranks to ensure that the model parameters remain synchronized after each optimizer step.

Problem (optimizer_state_sharding)

Implement a Python class to handle **optimizer state sharding**. The class should wrap an arbitrary input PyTorch `optim.Optimizer` and take care of synchronizing updated parameters after each optimizer step. We recommend the following public interface:

- `def __init__(self, params, optimizer_cls: Type[Optimizer], **kwargs: Any):` Initializes the sharded state optimizer. `params` is a collection of parameters to be optimized (or parameter groups, in case the user wants to use different hyperparameters, such as learning rates, for different parts of the model); these parameters will be sharded across all ranks. The `optimizer_cls` parameter specifies the type of optimizer to be wrapped (e.g., `optim.AdamW`). Finally, any remaining keyword arguments are forwarded to the constructor of `optimizer_cls`. Make sure to call the `torch.optim.Optimizer` super-class constructor in this method.

- `def step(self, closure, **kwargs):` Calls the wrapped optimizer's `step()` method with the provided closure and keyword arguments. After updating the parameters, synchronize with the other ranks.

- `def add_param_group(self, param_group: dict[str, Any]):` Adds a parameter group to the sharded optimizer. This is called during construction of the sharded optimizer by the super-class constructor and may also be called during training (e.g., for gradually unfreezing layers in a model). As a result, this method should handle assigning the model's parameters among the ranks.

Deliverable: Implement a container class to handle optimizer state sharding. To test your sharded optimizer, first implement the adapter `adapters.get_sharded_optimizer`. Then, to execute the tests, run:

```
uv run pytest tests/test_sharded_optimizer.py
```

We recommend running the tests multiple times (e.g., 5) to ensure that it passes reliably.

Now that we've implemented optimizer state sharding, let's analyze its effect on the peak memory usage during training and its runtime overhead.

Problem (optimizer_state_sharding_accounting)

(a) Create a script (optimizer_state_sharding.py) to profile the peak memory usage when training language models with and without optimizer state sharding. Using the standard configuration (1 node, 2 GPUs, model size in Table 1), report the peak memory usage after model initialization, directly before the optimizer step, and directly after the optimizer step. Do the results align with your expectations? Break down the memory usage in each setting (e.g., how much memory is used for parameters, optimizer states, activations, etc.).

Deliverable: A script (optimizer_state_sharding.py). On Gradescope, a 2–3 sentence response with peak memory usage results and a breakdown of how memory is divided among model and optimizer components.

(b) How does our implementation of optimizer state sharding affect training speed? Measure the time taken per iteration with and without optimizer state sharding for the standard configuration (1 node, 2 GPUs, model size in Table 1).

Deliverable: On Gradescope, a 2–3 sentence response reporting your timing measurements.

References

J. Austin, S. Douglas, R. Frostig, A. Levskaya, C. Chen, S. Vikram, F. Lebron, P. Choy, V. Ramasesh, A. Webson, and R. Pope. How to scale your model. `https://jax-ml.github.io/scaling-book/`, 2025.

S. Rajbhandari, J. Rasley, O. Ruwase, and Y. He. Zero: Memory optimizations toward training trillion parameter models. In SC20: International Conference for High Performance Computing, Networking, Storage and Analysis, pages 1–16. IEEE, 2020.

N. Tazi, F. Mom, H. Zhao, P. Nguyen, M. Mekkouri, L. Werra, and T. Wolf. The ultra-scale playbook: Training llms on gpu clusters, 2025.