PyTorch Distributed: Experiences on Accelerating Data Parallel Training

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

{shenli, yanlizhao, rvarm1, osalpekar}@fb.com,
pcnoordhuis@gmail.com, tengli@fb.com, adam.paszke@gmail.com,
{jeffksmith, bvaughan, pritam.damania, soumith}@fb.com

†Facebook AI ‡University of Warsaw

ABSTRACT

This paper presents the design, implementation, and evaluation of the PyTorch distributed data parallel module. PyTorch is a widely-adopted scientific computing package used in deep learning research and applications. Recent advances in deep learning argue for the value of large datasets and large models, which necessitates the ability to scale out model training to more computational resources. Data parallelism has emerged as a popular solution for distributed training thanks to its straightforward principle and broad applicability. In general, the technique of distributed data parallelism replicates the model on every computational resource to generate gradients independently and then communicates those gradients at each iteration to keep model replicas consistent. Despite the conceptual simplicity of the technique, the subtle dependencies between computation and communication make it non-trivial to optimize the distributed training efficiency. As of v1.5, PyTorch natively provides several techniques to accelerate distributed data parallel, including bucketing gradients, overlapping computation with communication, and skipping gradient synchronization. Evaluations show that, when configured appropriately, the PyTorch distributed data parallel module attains near-linear scalability using 256 GPUs.

PVLDB Reference Format:
Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, Soumith Chintala. PyTorch Distributed: Experiences on Accelerating Data Parallel Training. *PVLDB*, 13(12): 3005-3018, 2020.
DOI: https://doi.org/10.14778/3415478.3415530

*This work was conducted when Pieter Noordhuis was an employee at Facebook.

Proceedings of the VLDB Endowment, Vol. 13, No. 12
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3415478.3415530

1. INTRODUCTION

Deep Neural Networks (DNN) have powered a wide spectrum of applications, ranging from image recognition [20], language translation [15], anomaly detection [16], content recommendation [38], to drug discovery [33], art generation [28], game play [18], and self-driving cars [13]. Many applications pursue higher intelligence by optimizing larger models using larger datasets, craving advances in distributed training systems. Among existing solutions, distributed data parallel is a dominant strategy due to its minimally intrusive nature. This paper presents the design, implementation, and evaluation of the distributed data parallel package in PyTorch v1.5 [30].

Training a DNN model usually repeatedly conducts three steps [26], the forward pass to compute loss, the backward pass to compute gradients, and the optimizer step to update parameters. The concept of data parallelism is universally applicable to such frameworks. Applications can create multiple replicas of a model, with each model replica working on a portion of training data and performing the forward and backward passes independently. After that, model replicas can synchronize either their gradients or updated parameters depending on the algorithm. It's nominally possible to build a working version of data parallel purely on the application side, as it only requires inserting appropriate communications into every iteration. However, squeezing out the last bit of performance takes an enormous amount of effort in design and tuning. Providing native distributed data parallel APIs on the platform side would help application developers focus on optimizing their models, while the platform developing team could continuously and transparently improve the training speed. To provide a general distributed data parallel package, the challenges are three-fold.

- **Mathematical equivalence**: The purpose of data parallel is to speed up training on large datasets. Applications expect to harvest the same result model as if all training had been performed locally without model replication. This requires mathematical equivalence to local training despite its distributed nature.
- **Non-intrusive and interceptive API**: Application developments usually start from local models and then scale out when necessary. To avoid the exorbitant

hurdles during the transition, the API must be non-intrusive in application code. On the other hand, the API needs to allow the internal implementation to timely *intercept* signals to carry out communications and system optimizations.

- **High performance**: Data parallel training is subject to subtle dependencies between computations and communications. The design and implementation have to explore the solution space to efficiently convert more resources into higher training throughput.

PyTorch provides distributed data parallel as an `nn.Module` class, where applications provide their model at construction time as a sub-module. To guarantee mathematical equivalence, all replicas start from the same initial values for model parameters and synchronize gradients to keep parameters consistent across training iterations. To minimize the intrusiveness, the implementation exposes the same `forward` [7] API as the user model, allowing applications to seamlessly replace subsequent occurrences of a user model with the distributed data parallel model object with no additional code changes. Several techniques are integrated into the design to deliver high-performance training, including bucketing gradients, overlapping communication with computation, and skipping synchronization.

Evaluations were conducted on an exclusive 32-GPU cluster and on 256 GPUs from a much larger shared entitlement. We developed benchmarks to evaluate the distributed package across different scales to present an in-depth view of the performance implications of different optimization techniques and configurations. Experiments also cover the comparison between NCCL and Gloo communication libraries. The results show that 1) communication is the dominant training latency contributor, and its impact increases with model sizes; 2) bucket sizes considerably affect communication efficiency, which could lead to more than 2X speedup if configured properly; 3) skipping synchronizations appropriately would significantly reduce amortized communication overhead without noticeably degrading convergence speed.

Techniques described in this paper were first released in PyTorch v1.1. During the past year, we have seen significant adoption both internally and externally. Within Facebook, a workload study from `05/11/20` to `06/05/20` shows that more than 60% of production GPU hours during that period were spent on the PyTorch distributed data parallel package across a wide variety of applications, including speech, vision, mobile vision, translation, etc. There are three main contributions in this paper. First, this paper reveals the design and implementation of a widely adopted industrial state-of-the-art distributed training solution. Second, this paper highlights real-world caveats (*e.g.*, due to pluralized graphs) that were overlooked by prior work. Third, we share performance tuning experiences collected from serving internal teams and open-source community users and summarized several directions for future improvements.

The remainder of the paper is organized as follows. Section 2 briefly introduces PyTorch and data parallelism. Section 3 elaborates the design for the PyTorch distributed data parallel module. Implementations and evaluations are presented in Section 4 and Section 5 respectively. Then, Section 6 discusses lessons learned and opportunities for future improvements, and Section 7 surveys related work. Finally, Section 8 concludes the paper.

2. BACKGROUND

Before diving into distributed training, let us briefly discuss the implementation and execution of local model training using PyTorch. Then, we explain and justify the idea of data parallelism and describe communication primitives.

2.1 PyTorch

PyTorch organizes values into `Tensor`s which are generic n-dimensional arrays with a rich set of data manipulating operations. A `Module` defines a transform from input values to output values, and its behavior during the forward pass is specified by its `forward` member function. A `Module` can contain `Tensor`s as parameters. For example, a `Linear Module` contains a `weight` parameter and a `bias` parameter, whose `forward` function generates the output by multiplying the input with the `weight` and adding the `bias`. An application composes its own `Module` by stitching together native `Modules` (*e.g.*, linear, convolution, etc.) and `Functions` (*e.g.*, relu, pool, etc.) in the custom `forward` function. A typical training iteration contains a forward pass to generate losses using inputs and labels, a backward pass to compute gradients for parameters, and an optimizer step to update parameters using gradients. More specifically, during the forward pass, PyTorch builds an autograd graph to record actions performed. Then, in the backward pass, it uses the autograd graph to conduct backpropagation to generate gradients. Finally, the optimizer applies the gradients to update parameters. The training process repeats these three steps until the model converges.

2.2 Data Parallelism

PyTorch offers several tools to facilitate distributed training, including `DataParallel` for single-process multi-thread data parallel training using multiple GPUs on the same machine, `DistributedDataParallel` for multi-process data parallel training across GPUs and machines, and `RPC` [6] for general distributed model parallel training (*e.g.*, parameter server [27]). The remainder of this paper focuses on `DistributedDataParallel`. Data parallelism enables distributed training by communicating gradients before the optimizer step to make sure that parameters of all model replicas are updated using exactly the same set of gradients, and hence model replicas can stay consistent across iterations.

Parameter averaging is another popular technique to scale out model training. Similarly, it can launch multiple processes across multiple machines, but instead of synchronizing gradients, parameter averaging directly computes the average of all model parameters. This occurs after the local optimizer step, meaning that parameter averaging can be implemented completely as an auxiliary step and does not need to interact with local training steps at all, which is attractive as it can easily and cleanly decouple the code of distributed training and local iterations. There are several caveats with parameter averaging.

- Parameter averaging can produce vastly different results compared to local training, which, sometimes, can be detrimental to model accuracy. The root cause is that parameter averaging is **not** mathematically equivalent to processing all input data locally, especially when the optimizer relies on past local gradients values (*e.g.*, momentum). As different model replicas are likely to see different gradients, the states in optimizers can gradually diverge, causing conflicting gradient descent directions. This can result in inexplicable differences in performance when switching from locally optimized models to large scale deployed models.

- The structure of parameter averaging orchestrates computation (*i.e.*, backward pass) and communication (*i.e.*, computing average) into non-overlapping phases, using optimizer `step()` functions as a hard separation point. Regardless of how vigorously we optimize the computation or communication, one type of resource will stay idle at any given time instance, giving up a substantial performance optimization opportunity.

Given the above fundamental pitfalls, we decided to implement distributed training using data parallelism to synchronize gradients instead of parameters. Note that, applications can still easily build parameter averaging using PyTorch. In fact, the collective communication feature described in Section 3.3 is an appropriate solution for this use case. Applications just need to explicitly launch `AllReduce` operations to calculate averaged parameters accordingly.

2.3 AllReduce

`AllReduce` is the primitive communication API used by `DistributedDataParallel` to compute gradient summation across all processes. It is supported by multiple communication libraries, including NCCL [2], Gloo [1], and MPI [4]. The `AllReduce` operation expects each participating process to provide an equally-sized tensor, collectively applies a given arithmetic operation (*e.g.*, `sum`, `prod`, `min`, `max`) to input tensors from all processes, and returns the same result tensor to each participant. A naive implementation could simply let every process broadcast its input tensor to all peers and then apply the arithmetic operation independently. However, as `AllReduce` has significant impact on distributed training speed, communication libraries have implemented more sophisticated and more efficient algorithms, such as ring-based `AllReduce` [2] and tree-based `AllReduce` [23]. As one `AllReduce` operation cannot start until all processes join, it is considered to be a synchronized communication, as opposed to the P2P communication used in parameter servers [27].

3. SYSTEM DESIGN

PyTorch [30] provides a `DistributedDataParallel` (`DDP`[1]) module to help easily parallelize training across multiple processes and machines. During distributed training, each process has its own local model replica and local optimizer. In terms of correctness, distributed data parallel training and local training must be mathematically equivalent. `DDP` guarantees the correctness by making sure that all model replicas start from the exact same model state, and see the same parameter gradients after every backward pass. Therefore, even though optimizers from different processes are all independent, they should be able to bring their local model replicas to the same state at the end of every iteration[2]. Fig. 1 illustrates building blocks of `DDP`, which contains a Python API frontend, C++ gradient reduction core algorithm, and employs the `c10d` collective communication library. The following sections are presented in the top-down order of this stack graph.

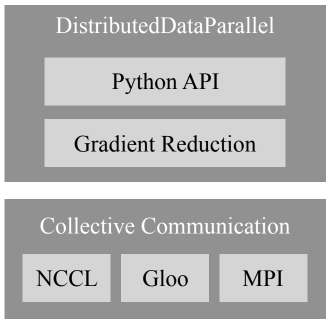


Figure 1: Building Blocks

Section 3.1 presents general principles that drive `DDP` API design. Section 3.2 explains gradient reduction techniques used in the PyTorch distributed data parallel package. Finally, Section 3.3 discusses the collective communication backend options for `DDP`.

[1]For simplicity, the rest of the paper uses the acronym `DDP` to represent `DistributedDataParallel` henceforth.

[2]For optimizers with intrinsic randomness, different processes can initialize their states using the same random seed.

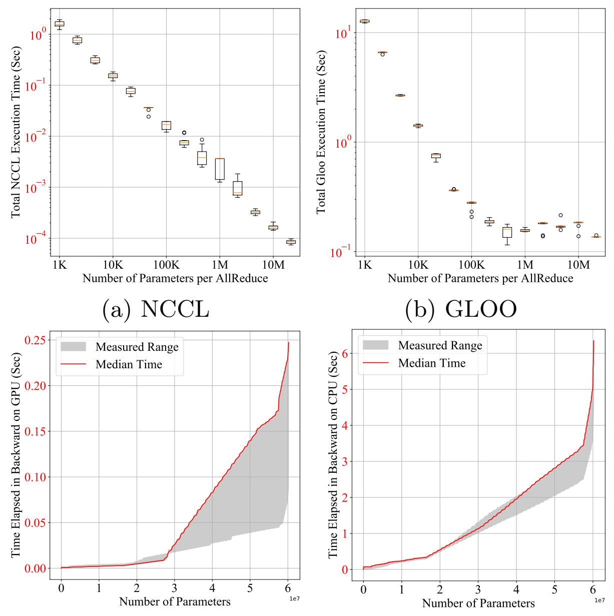
3.1 API

When designing the API, we have defined two design goals to achieve the necessary functionality.

- **Non-intrusive**: The API must be non-intrusive to applications. Application developers usually start from writing local training scripts, and scale out when hitting the resource limit on a single machine. At that point, it is unacceptable to ask developers to rewrite the entire application to enable distributed data parallel training. Instead, the developer 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. Distributed data parallel aims at accelerating training by using more computational resources. This process requires subtle optimizations in both computations and communications to achieve the best performance. Hence, the API must expose as many optimization opportunities as possible to the internal implementation.

Given the above requirements, we implemented distributed data parallel as an `nn.Module`, which takes the local model as a constructor argument and transparently synchronizes gradients in the backward pass. The code snippet below shows an example of using `DDP` module. This example uses an `nn.Linear` layer to create a local model on line 10. Then, it converts the local model into a distributed training model on line 11 and sets up the optimizer on line 12. Line 14 through 23 are typical forward pass, backward pass, and optimizer step implementations. In this toy distributed training example, line 11 is the only difference that converts a local training application into a distributed one, which satisfies the non-intrusive requirement. It also fulfills the interceptive requirement. The constructor allows `DDP` to inspect the model structure and parameters. After construction, the local model is replaced by the distributed one, which can then easily intercept the `forward()` call to perform necessary actions accordingly. For the backward pass, `DDP` relies on backward hooks to trigger gradient reduction, which will be invoked by the autograd engine when executing `backward()` on the loss tensor.



(a) NCCL (b) GLOO

(c) GPU (d) CPU

Figure 2: Communication vs Computation Delay

```python
import torch
import torch.nn as nn
import torch.nn.parallel as par
import torch.optim as optim

# initialize torch.distributed properly
# with init_process_group

# setup model and optimizer
net = nn.Linear(10, 10)
net = par.DistributedDataParallel(net)
opt = optim.SGD(net.parameters(), lr=0.01)

# run forward pass
inp = torch.randn(20, 10)
exp = torch.randn(20, 10)
out = net(inp)

# run backward pass
nn.MSELoss()(out, exp).backward()

# update parameters
opt.step()
```

3.2 Gradient Reduction

The gradient reduction algorithm in DDP has evolved over the past releases. To introduce the structure of the current implementation, let us start from a naive solution, gradually introduce more complexities, and land in the current version as of today in PyTorch v1.5.0. This will also explain how the same simple API described in Section 3.1 allows us to install various performance optimization algorithms.

3.2.1 A Naive Solution

As mentioned in the beginning of Section 3, DDP guarantees correctness by letting all training processes (1) start from the same model state and (2) consume the same gradients in every iteration. The former can be achieved by broadcasting model states from one process to all others at the construction time of DDP. To implement the latter, a naive solution can insert a gradient synchronization phase after the local backward pass and before updating local parameters. However, the API shown in Section 3.1 does not provide an explicit entry point for this phase as there is nothing between `backward()` and `step()`. Fortunately, the PyTorch autograd engine accepts custom backward hooks. DDP can register autograd hooks to trigger computation after every backward pass. When fired, each hook scans through all local model parameters, and retrieves the gradient tensor from each parameter. Then, it uses the `AllReduce` collective communication call to calculate the average gradients on each parameter across all processes, and writes the result back to the gradient tensor.

The naive solution works correctly, but there are two performance concerns.

- Collective communication performs poorly on small tensors, which will be especially prominent on large models with a massive number of small parameters.
- Separating gradient computation and synchronization forfeits the opportunity to overlap computation with communication due to the hard boundary in between.

The following sections elucidates solutions to address the above two concerns.

3.2.2 Gradient Bucketing

The idea of gradient bucketing is motivated by the observation that collective communications are more efficient on large tensors. Fig. 2 (a) and (b) provide a quantitative view, which show the total execution time to `AllReduce` 60M `torch.float32` parameters with different numbers of parameters per `AllReduce`. To maximize the bandwidth utilization, `AllReduce` operations are launched asynchronously and block waiting on all of them together, mimicking DDP's gradient reduction algorithm. The experiments are conducted on an NVLink [3] enabled server with two NVIDIA Quadro GP100 GPUs. NCCL [2] `AllReduce` runs on CUDA input tensors directly, while Gloo [1] `AllReduce` runs on CPU input tensors to eliminate the overhead of copying between CUDA memory to CPU memory when using Gloo backend. The total communication time clearly decreases when using larger input tensors, for both NCCL and Gloo. Gloo reaches pinnacle speed at around 500K parameters per input tensor, while there is no clear saturation signal for NCCL on NVLink with even 20M-parameter GPU tensors.

These experiments suggest that, instead of launching a dedicated `AllReduce` immediately when each gradient tensor becomes available, DDP can achieve higher throughput and lower latency if it waits for a short period of time and buckets multiple gradients into one `AllReduce` operation. This would be especially helpful for models with many small parameters. However, DDP should not communicate all gradients in one single `AllReduce`, otherwise, no communication can start before the computation is over. Fig. 2 (c) and (d) show the GPU and CPU backward computations time of a ResNet152 [20] that contains roughly 60M parameters. The X axis is the number of ready gradients and the Y axis the time elapsed since the beginning of the backward pass. The backward on GPU takes about 250ms to complete, which is in the same order of magnitude as NCCL on NVLink. This conclusion also applies to Gloo and CPU backward. These measurements herald that, with relatively small bucket sizes, DDP can launch `AllReduce` operations concurrently with the

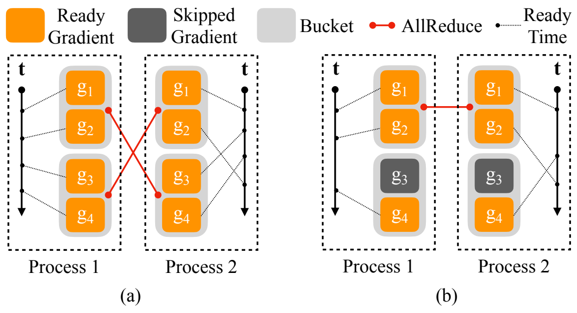


Figure 3: Gradient Synchronization Failures

backward pass to overlap communication with computation, which would make a difference in per iteration latency.

3.2.3 Overlap Computation with Communication

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. Under such settings, triggering `AllReduce` at the end of the backward pass is no longer sufficient. It needs to react to more frequent signals and launches `AllReduce` more promptly. Therefore, `DDP` registers one autograd hook for each gradient accumulator. The hook fires after its corresponding accumulator updating the gradients, and will inspect the bucket it pertains. If hooks of all gradients in the same buckets have fired, the last hook will trigger an asynchronous `AllReduce` on that bucket.

Two caveats require caution. First, the reducing order must be the same across all processes, otherwise, `AllReduce` contents might mismatch, resulting in incorrect reduction result or program crash. However, PyTorch dynamically builds the autograd graph in every forward pass, and different processes might not agree on the gradient ready order. Fig. 3 (a) shows one example, where the two vertical axes represent time and dotted lines indicate when a gradient is ready. In process 1, the four gradients are computed in order, but the gradient g_2 are computed after g_3 and g_4 on process 2. In this case, if all processes `AllReduce` buckets as soon as they become ready, the `AllReduce` content would mismatch. Therefore, all processes must use the same bucketing order, and no process can launch `AllReduce` on bucket `i+1` before embarking bucket `i`. If bucket `0` is the last one that becomes ready, there is no way that communication can overlap with computation. PyTorch v1.5.0 addresses this problem by using the reverse order of `model.parameters()` as the bucketing order, assuming that, layers are likely registered according to the same order as they are invoked in the forward pass. Hence, the reverse order should approximately represent the gradient computation order in the backward pass. Admittedly, this is not a perfect solution, but is an approximation that we can rely on with minimum engineering overhead.

Second, it is possible that one training iteration only involves a sub-graph in the model and the sub-graph can be different from iteration to iteration, meaning that some gradients might be skipped in some iterations. However, as gradient-to-bucket mapping is determined at the construction time, those absent gradients would leave some buckets never seeing the final autograd hook and failing to mark the bucket as ready. As a result, the backward pass could hang. Fig. 3 (b) shows an example, where the parameter corresponding to gradient g_3 is skipped in one iteration, leading to the absent of the ready signal for g_3. To address this problem, `DDP` traverses the autograd graph from the output tensors of the forward pass to find all participating parameters. The readiness of those participating tensors is a sufficient signal to conclude the completion of the backward pass. Therefore, `DDP` can avoid waiting for the rest of the parameter gradients by proactively marking them ready at the end of the forward pass. Note that, this change does not prevent us from developing non-intrusive APIs, because application directly invokes the `forward` function on `DDP` and hence `DDP` can easily insert this step in its member function.

Algorithm 1: DistributedDataParallel

```
Input: Process rank $r$, bucket size cap $c$, local model $net$
1  Function constructor($net$):
2      if $r=0$ then
3          broadcast $net$ states to other processes
4      init buckets, allocate parameters to buckets in the
         reverse order of net.parameters()
5      for $p$ in $net.parameters()$ do
6          acc ← $p$.grad_accumulator
7          acc → add_post_hook(autograd_hook)
8  Function forward($inp$):
9      out = $net$(inp)
10     traverse autograd graph from out and mark
         unused parameters as ready
11     return out
12 Function autograd_hook($param\_index$):
13     get bucket $b_i$ and bucket $offset$ using $param\_index$
14     get parameter $var$ using $param\_index$
15     view ← $b_i.narrow(offset, var.size())$
16     view.copy_(var.grad)
17     if all grads in $b_i$ are ready then
18         mark $b_i$ as ready
19     launch AllReduce on ready buckets in order
20     if all buckets are ready then
21         block waiting for all AllReduce ops
```

Algorithm 1 presents the pseudo-code of `DDP`. The constructor contains two major steps, broadcasting model states and installing autograd hooks. `DDP`'s `forward` function is a simple wrapper of the local model's `forward`. It traverses the autograd graph to mark unused parameters accordingly. The `autograd_hook` takes the internal parameter index as input, which helps to find the parameter tensor and its belonging bucket. It writes the local gradient to the correct offset in the bucket and then launches the asynchronous `AllReduce` operation. There is an additional finalizing step omitted in the pseudo-code that waits for `AllReduce` operations and writes the value back to gradients at the end of the backward pass. Fig. 4 elucidates how `DDP` interacts with the local model during the forward and backward passes.

The above solution works for most use cases. However, as `DDP` always computes the average of all gradients and writes them back to parameter `.grad` field, an optimizer cannot distinguish whether a gradient has participated in the last backward pass or not. Due to the decoupled design of `DDP` and the optimizer, there is no side channel for `DDP` to allude

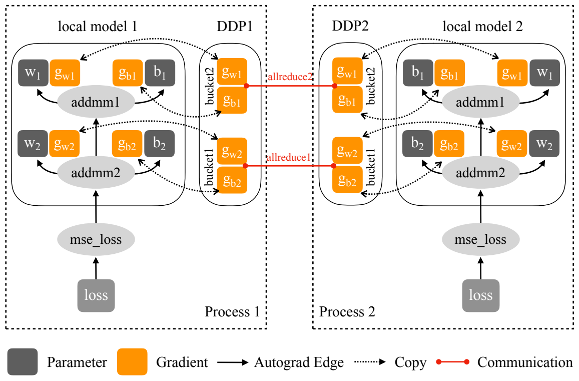


Figure 4: Distributed Gradient Reduction

that information to the optimizer. Without this information, the training process could suffer from regressions on model accuracy, *e.g.*, when the optimizer uses gradient absence information to skip updating momentum values. To tackle this problem, DDP should only touch gradients that are indeed involved in the backward pass. Nevertheless, this information cannot be extracted from the local autograd graph alone, because locally absent gradients might still be involved in the forward/backward pass in a peer DDP process. Therefore, DDP uses a bitmap to keep track of local parameter participants and launches one additional AllReduce to collect globally unused parameters. Unfortunately, DDP cannot coalesce this bitmap into other gradient AllReduce operations due to the potential mismatch in element types. Such additional overhead only materializes when the application explicitly tells DDP to look for unused parameters, and hence the price is only paid when necessary.

3.2.4 *Gradient Accumulation*

One common technique to speed up distributed data parallel training is to reduce gradient synchronization frequencies. Instead of launching AllReduce in every iteration, the application can conduct n local training iterations before synchronizing gradients globally. This is also helpful if the input batch is too large to fit into a device, where the application could split one input batch into multiple micro-batches, run local forward and backward passes on every micro-batch, and only launch gradient synchronization at the boundaries of large batches. Theoretically, this should produce the same results as if all data in the large batch is processed in one shot, as gradients will simply be accumulated to the same tensor. However, this conflicts with the gradient reduction algorithm discussed in Section 3.2.3 to some degree. That algorithm would mark unused parameters as ready at the end of every forward pass, while those unused parameters in one iteration still could participate in subsequent iterations. Moreover, DDP cannot distinguish whether the application plans to immediately invoke `optimizer.step()` after backward or accumulate gradients through multiple iterations. Therefore, we need to introduce one additional interface (*i.e.*, `no_sync`) for this use case.

Under the hood, the implementation for `no_sync` is very simple. The context manager just toggles a flag on entering and exiting the context, and the flag is consumed in the `forward` function of DDP. In `no_sync` mode, all DDP hooks are disabled, and the first backward pass out of the context will synchronize the accumulated gradients altogether. The information of globally unused parameters also accumulates in the bitmap, and serves when the next communication takes place. Below is an example code snippet.

```python
ddp = DistributedDataParallel(net)
with ddp.no_sync():
  for inp, exp in zip(inputs, expected_outputs):
    # no synchronization, accumulate grads
    loss_fn(ddp(inp), exp).backward()
# synchronize grads
loss_fn(ddp(another_inp), another_exp).backward()
opt.step()
```

3.3 Collective Communication

Distributed data parallel training uses a special communication pattern, where every participant provides an equally-sized tensor and collects the global sum across all participants. This can certainly be implemented as a gather operator followed by local reductions on every participant using point-to-point communication, but that would forfeit opportunities for performance optimizations [23]. DDP is built on top of collective communication libraries, including three options, NCCL [2], Gloo [1], and MPI [4]. [3] DDP takes the APIs from the three libraries and wraps them into the same ProcessGroup API. The name heralds that ProcessGroup expects multiple processes to work collectively as a group. All ProcessGroup instances construct at the same time by using a rendezvous service, where the first arrival will block waiting until the last instance joins. For NCCL backend, the ProcessGroup maintains a dedicated set of CUDA streams for communication, so that communications will not block the computation in the default stream. As all communications are collective operations, subsequent operations on all ProcessGroup instances must match in size and type and follow the same order. Using the same ProcessGroup API for all libraries allows us to experiment with different communication algorithms with the same DDP implementation. For example, PyTorch v1.5 provides a composite round-robin ProcessGroup implementation, which takes a list of ProcessGroup instances and dispatches collective communications to those ProcessGroup instances in a round-robin manner. By using round-robin ProcessGroups, DDP can attain higher bandwidth utilization if a single NCCL, Gloo, or MPI ProcessGroup is unable to saturate the link capacity.

4. IMPLEMENTATION

The implementation of DDP has evolved several times in the past few releases. This section focus on the current status as of PyTorch v1.5.0. DDP implementation lives in both Python and C++ files, with Python exposing the API and composing non-performance-critical components, and C++ serving the core gradient reduction algorithm. The Python API calls into C++ core through Pybind11 [5].

4.1 Python Front-end

The DDP `nn.module` is implemented in `distributed.py`, which contains user-facing components, including the constructor, the `forward` function, and the `no_sync` context manager. Besides the general ideas highlighted in Section 3, there are several implementation details in the Python front-end that shapes the behavior of DDP.

[3]Please refer to documents of the three libraries for their design and implementation.

Configurable Knobs are exposed in the DDP constructor API, including 1) process_group to specify a process group instance for DDP to run AllReduce, which helps to avoid messing up with the default process group, 2) bucket_cap_mb to control the AllReduce bucket size, where applications should tune this knob to optimize training speed, and 3) find_unused_parameters to toggle whether DDP should detect unused parameters by traversing the autograd graph.

Model Device Affinity in the local model also governs DDP's behavior, especially if the model spans multiple devices, which is common when the model is too large to fit into a single device. For large models, applications can place different layers of the model onto difference devices, and use Tensor.to(device) API to move intermediate output from one device to another. DDP also works with multi-device models. As long as the device_ids argument is None or an empty list, DDP will inspect the model, perform sanity checks and apply configurations accordingly. Then, it treats the multi-device model as one entirety.

Model Buffers are necessary when layers need to keep track of states like the running variance and the running mean (*e.g.*, BatchNorm). DDP supports model buffers by letting the process with the rank 0 to take the authority. If the model contains buffers, DDP will broadcast the buffer values from rank 0 process to all other processes before starting the forward pass on the local model. This behavior is also compatible with the no_sync mode. When no_sync mode is enabled, it sets a flag in the forward pass properly to indicate whether it expects gradient reductions in the immediate backward pass. If the communication takes place, DDP will then broadcast buffers prior to the subsequent forward pass.

4.2 Core Gradient Reduction

Major development efforts are spent in gradient reduction as it is the most performance-critical step in DDP. The implementation lives in reducer.cpp which consists of four main components, namely, building parameter-to-bucket map, installing autograd hooks, launching bucket AllReduce, and detecting globally unused parameters. This section expatiates on these four components.

Parameter-to-Bucket Mapping has a considerable impact on DDP speed. In every backward pass, tensors are copied from all parameter gradients to buckets, and averaged gradients are copied back after AllReduce. To accelerate copy operations, buckets are always created on the same device as the parameters. If the model spans multiple devices, DDP takes device affinity into consideration to make sure that all parameters in the same bucket are on the same device. The order of AllReduce also makes a difference, as it dictates how much communication can overlap with computation. DDP launches AllReduce in the reverse order of model.parameters().

Autograd Hook is the entry point for DDP in the backward pass. During construction, DDP loops over all parameters in the model, finds the gradient accumulator on every parameter, and installs the same post-hook function to every gradient accumulator. The gradient accumulator will fire post hooks when the corresponding gradient is ready, and DDP will figure out when an entire bucket is ready to launch an AllReduce operation. However, as there is no guarantee on the order of gradient readiness, DDP cannot selectively pick parameters to install hooks. In the current implementation, each bucket keeps a count of pending gradients. Each post-hook function decrements the count, and DDP marks a bucket as ready when that count reaches zero. In the next forward pass, DDP replenishes the pending gradient count for every bucket.

Bucket AllReduce is the main source of communication overhead in DDP. On one hand, packing more gradients into the same bucket would reduce the amortized system overhead of communication. One the other hand, using a large bucket size would result in longer lead time for reduction, as each bucket needs to wait for more gradients. Hence, bucket size is the key trade-off. By default, each bucket is 25MB in size. Applications should measure their impact empirically and set it to the optimal value for their use cases.

Globally Unused Parameters' gradients should stay intact during the forward and the backward passes. Detecting unused parameters requires global information, as one parameter could be absent in one DDP process during one iteration, but participates training in the same iteration in another process. DDP maintains local unused parameter information in a bitmap, and launches an additional AllReduce to gather a global bitmap. As the bitmap is much smaller than tensor sizes, instead of creating per-bucket bitmaps, all parameters in the model share the same bitmap. The bitmap lives on CPU to avoid launching dedicated CUDA kernels for each update. However, some ProcessGroup backends might not be able to run AllReduce on CPU tensors. For example, ProcessGroupNCCL only supports CUDA tensors. Moreover, as DDP should work with any custom ProcessGroup backend, it cannot make assumptions that all backends support CPU tensors. To address this problem, DDP maintains another bitmap on the same device as the first model parameter, and invokes a non-blocking copy to move the CPU bitmap to the device bitmap for collective communications.

5. EVALUATION

This section presents the evaluation results of PyTorch DDP using an exclusive 32 GPU cluster and a shared entitlement. Fig. 5 shows the interconnection of the 8 GPUs within the same server. In the exclusive cluster, the GPUs are located on 4 servers, connected using Mellanox MT27700 ConnectX-4 100GB/s NIC. All 4 servers reside in the same rack, and each server is equipped with 8 NVIDIA Tesla V100 GPUs. We only use the shared entitlement when a set of experiments require more than 32 GPUs. In the shared entitlement, we submit jobs to run on different numbers of GPUs where different jobs can run on different machines, and hence the hardware and network connectivity can vary from job to job. Although the disparity in the test environment can lead to different latency measures even for the same code, we pack the same set of

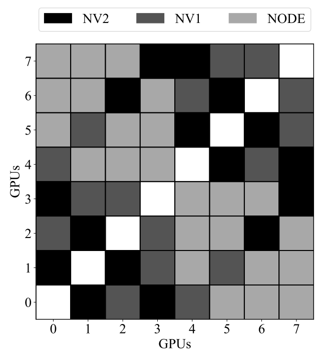


Figure 5: GPU Topology

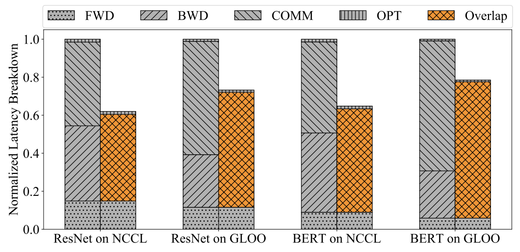


Figure 6: Per Iteration Latency Breakdown

experiments into the same job, so that the trend shown in the same curve is still meaningful.

We measure DDP per iteration latency and scalability using two popular models, ResNet50 [20] and BERT [15], to represent typical vision and NLP applications. Most experiments use randomly generated synthetic inputs and labels, which are sufficient as the purpose is to compare per iteration latency instead of model accuracy. Experiments compute losses using the CrossEntropyLoss function and update parameters using the SGD optimizer. Configurations for accuracy-related experiments will be explained in detail close to their presentations.

5.1 Latency Breakdown

A typical training iteration contains three steps: forward pass to compute loss, backward pass to compute gradients, and optimizer step to update parameters. With DDP, the backward pass involves local computation and AllReduce communication. To demonstrate the effectiveness of overlapping computation with communication, Fig. 6 plots the latency breakdown when using NCCL and Gloo backends for ResNet50 and BERT models respectively. All experiments are conducted using 32 GPUs across 4 machines. To visually compare the speedup on different model and backend combinations, we normalize the total latency to 1 for all non-overlapping cases. The results demonstrate that the backward pass is the most time-consuming step with PyTorch DDP training, as AllReduce communications (*i.e.*, gradient synchronization) are completed in this step. This observation justifies that the DDP backward pass deserves the most efforts for improvements. Within the backward pass, the communication step takes more than half of the total delay and this is exacerbated with the increase of the model size. Between these two backends, NCCL is considerably faster than GLOO. The speedup is most effective when the computation and communication take roughly the same amount of time as they can overlap more. The overlapping approach helps ResNet and BERT on NCCL attain 38.0% and 35.2% speedup. With GLOO backend, the gain shrinks to 26.8% and 21.5% respectively, as GLOO communication becomes the dominating delay in the backward pass.

5.2 Bucket Size

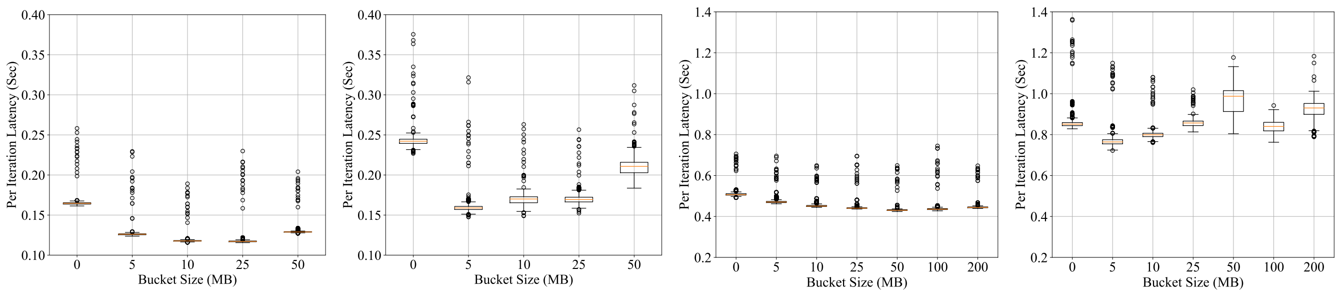
To avoid launching an excessive number of AllReduce operations, DDP organizes small gradients into larger buckets and synchronizes each bucket using an AllReduce operation. With this design, bucket size is an important configuration knob. DDP exposes this knob to applications through bucket_cap_mb argument. No single bucket size can best serve all applications. This value should be measured and determined empirically. The default value of bucket_cap_mb is 25MB, which is our best effort estimation based experiences. The following experiments also confirm this is a reasonable choice for ResNet50 and BERT. This section compares per iteration latency across different bucket sizes using 16 GPUs on two machines. Zero bucket size means each gradient will be communicated on its own as soon as it is ready. This serves as a baseline on one extreme of the bucket size spectrum. The other extreme is communication all gradients in one short, which is skipped as results in Fig. 7 and Fig. 8 clearly show the best option for both ResNet50 and BERT is somewhere in the middle.

Fig. 7 (a) uses box-whisker to illustrate how bucket size affects per iteration latency on ResNet50 with NCCL backend. The x-axis is the bucket size in MBs, and Y-axis per iteration latency in seconds. The outliers are the tiny delay spikes at 100 iteration boundaries caused by DDP instance re-construction and input data regeneration. Other than that, delays of most iterations concentrate in a very narrow time range, which also agrees with the results shown in Fig. 6 (a). The results show that the highest speed is achieved between 10MB and 25MB bucket sizes. Fig. 7 (b) presents the same measurements for Gloo backend. The results are different from NCCL backend in two ways, 1) per iteration latency falls into a large range, 2) the 5MB bucket size attains higher speed compared to 10MB and 25MB. The first difference matches with Fig. 6 (b). To understand the second difference, let us revisit Fig. 2 (b) on Gloo AllReduce latency across different tensor sizes. It's clear that the total AllReduce time fluctuates around the same level when the bucket size is larger than 512KB. Therefore, larger bucket sizes beyond 512KB with Gloo backend would only mean longer waiting time for gradients, which leads to longer per iteration latency. Fig. 7 (c) and (d) show the measurements for BERT model. As BERT model contains 15X more parameters compared to ResNet50, intuitively, it should benefit from larger buckets as larger communication overheads would dwarf the waiting time for the first bucket. The results verified the intuition with NCCL backend, where 50MB bucket size leads to the best performance. However, with Gloo backend, 5MB bucket size still wins with the lowest per iteration latency.

Fig. 8 presents the results of the same set of experiments but on 32 GPUs. In this case, the outliers span a larger range, which is not surprising as synchronizations usually take longer with more participants and the impact of strangler is more prominent. Fig. 8 (a) and (b) both suggest that 0MB bucket size leads to obviously longer per iteration latency on 32 GPUs compared to 16 GPUs, as per-gradient reductions on a larger cluster are expected to be slower. However, when bucket size is set to above 5MB, scaling from 16 GPUs to 32 GPUs does not lead to a noticeable speed regression. This is probably because although individual AllReduce operations is expected to be slower, asynchronous execution and parallelism could help to hide the overall delay.

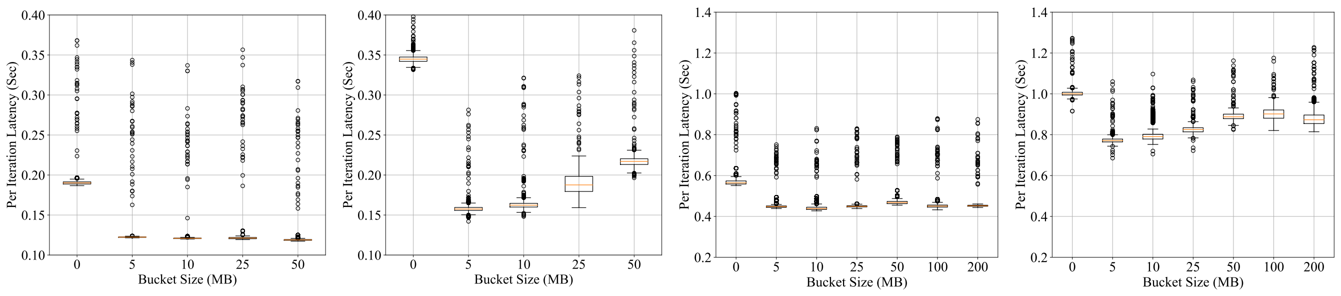
5.3 Scalability

To understand the scalability of DDP, we measure per iteration training latency of ResNet50 and BERT using NCCL and Gloo backend on up to 256 GPUs in the shared entitlement. Results are presented in Fig. 9. The X-axis is the number of GPUs, and Y-axis the latency. Figure 9 (a) shows that the per iteration latency steadily increases as it scales



(a) ResNet50 on NCCL (b) ResNet50 on Gloo (c) BERT on NCCL (d) BERT on Gloo

Figure 7: Per Iteration Latency vs Bucket Size on 16 GPUs



(a) ResNet50 on NCCL (b) ResNet50 on Gloo (c) BERT on NCCL (d) BERT on Gloo

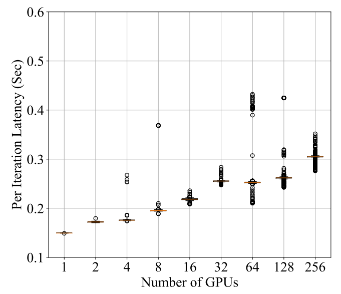
Figure 8: Per Iteration Latency vs Bucket Size on 32 GPUs

out. Using 256 GPUs leads to 100% slow down in each iteration compared to local training, meaning that the real scaling factor is 256 × 50% = 128. With the BERT model, the per-iteration latency significantly increases due to the larger model size. Another observation is that the 16-GPU case suffers a longer per-iteration delay compared to the 32-GPU case in Figure 9 (c). We suspect this is because either the 16-GPU experiments were on a slow or congested link or there are other workflows in the shared entitlement competing for resources with our job. Fig. 9 (b) and (d) show the results for Gloo backend and the per-iteration slowdown is about 3X for ResNet and 6X for BERT when using 256 GPUs. The deteriorated training speed with larger model sizes indicates that the network is the bottleneck resource when using Gloo backend in this experiment.

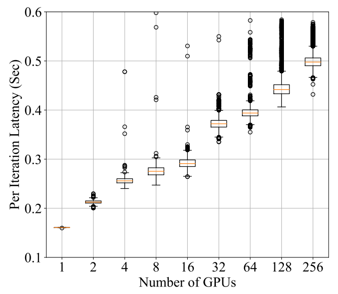
In general, scaling out to more GPUs slows down individual iterations. One option to mitigate the overhead is skipping gradient synchronizations, *i.e.*, perform gradient reduction every n iterations. This approach helps to considerably reduce the amortized latency. Fig. 10 depicts the average per iteration latency for conducting gradient reduction every 1, 2, 4, and 8 iterations. To visually compare the effectiveness of this method, we consolidated different skipping configurations for the same model and backend combination into the same figure. ResNet50 on NCCL and Gloo sees 38% and 57% speed up with 256 GPUs when conducting gradient sync every 8 iterations. There is a sudden jump in delay with NCCL backend when scaling from 128 to 256 and this occurs to all experiments shown in this figure. We believe this is caused by slow or congested links among some of those 256 nodes which are not included in the 128-GPU experiments. Besides the per iteration latency, it's also crucial to measure the convergence speed to verify if the acceleration might be erased by convergence slowdown. The experiments use MNIST [25] dataset to train the ResNet. The learning rate is set to 0.02 and the batch size is 8. Results are plotted in Fig. 11 (a), which only contains the measurements for NCCL backend as the communication layer does not change the convergence speed. X-axis is the number of iterations and Y-axis the loss. Please note that the goal of this experiment is not developing the best model for MNIST, instead, it only aims to show the impact of skipping synchronization on the model convergence. The raw loss data oscillate severely, which are presented by the tiny dots. Directly connecting them into a line would result in the last curve covering all previous drawn ones, making them less visible. Therefore, we apply an order 3 low pass filter by using `filtfilt` from SciPy [8] and plot the smoothed loss curve. The figure confirms that using `no_sync` in this case only leads to negligible exacerbation to the convergence speed. However, we must emphasize that the impact of `no_sync` could depend on the configuration. Fig. 11 (b) shows similar measurements by replacing batch size to 256 and learning rate to 0.06. As highlighted by the red box in the right bottom corner, `no_sync` hurts the final training loss. It is because large batch size and `no_sync` cause more gradients to be accumulated between consecutive communications and optimizer steps, which implicitly requires using a smaller learning rate. In summary, when skipping synchronizations properly, `DDP` attains near linear scalability with negligible accuracy penalty.

5.4 Round-Robin Process Group

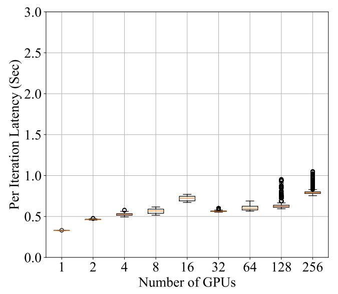
Another technique to speed up training is to use multiple process groups to work around subtle intrinsic concurrency limitations in process group backend implementations. The



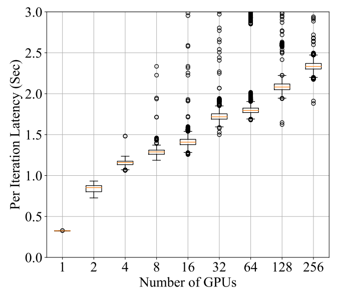
(a) ResNet50 on NCCL



(b) ResNet50 on Gloo

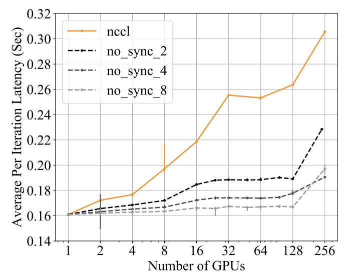


(c) BERT on NCCL

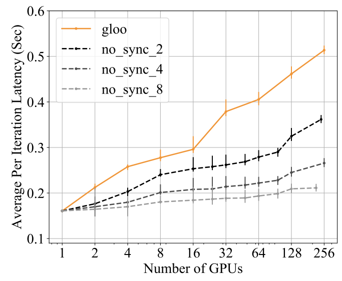


(d) BERT on Gloo

Figure 9: Scalability

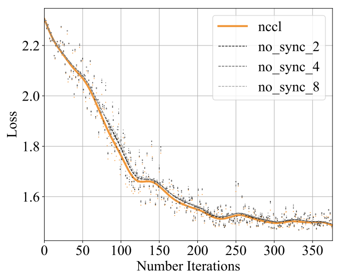


(a) ResNet50 on NCCL

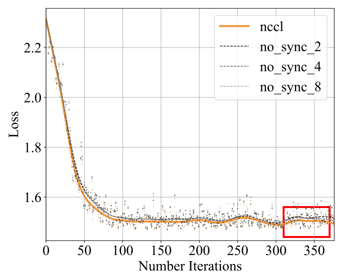


(b) ResNet50 on Gloo

Figure 10: Skip Gradient Synchronization

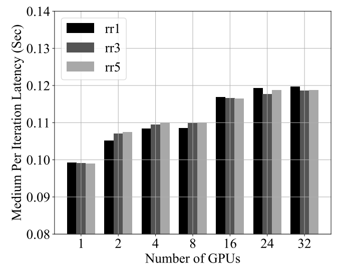


(a) Batch Size = 8

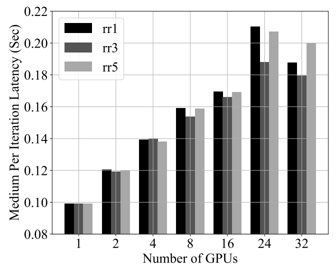


(b) Batch Size = 256

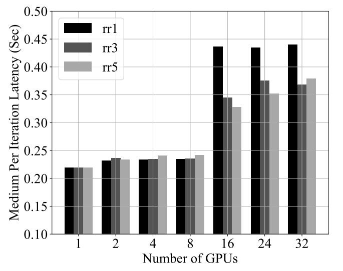
Figure 11: Accuracy with Skipping Synchronization



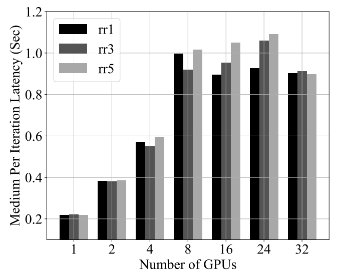
(a) ResNet50 on NCCL



(b) ResNet50 on Gloo



(c) BERT on NCCL



(d) BERT on Gloo

Figure 12: Round-Robin Process Group

concurrency limitations could come from NCCL streams or Gloo threads, depending on the type of the backend, which might prevent one process group instance to fully utilize all link bandwidth. The PyTorch distributed package supports composing a Round-Robin process group with multiple NCCL or Gloo process groups, which dispatches collective communications to different process group instances in Robin-Robin order. Fig. 12 plots the per iteration latency of Round-Robin process group using 1, 3, and 5 NCCL or Gloo process groups, where `rrx` stands for **R**ound-**R**obin with **x** process group instances. ResNet50 on NCCL backend sees negligible differences with different amounts of process groups, meaning that for relatively small models like ResNet50, bandwidth is not the bottleneck resource. Noticeable difference can be observed in ResNet50 on Gloo, where `rr3` consistently outperforms `rr1`. The most prominent acceleration occurs in BERT model with NCCL backend, where `rr3` achieves 33% speedup compared to `rr1` on 16 GPUs, revealing that one NCCL group is incompetent to saturate the link capacity.

6. DISCUSSION

This section discusses lessons learned from our experiments and past experiences. We then present several ideas for future improvements.

6.1 Lessons Learned

Distributed data parallel training is a conceptually simple but practically subtle framework. There are various techniques to improve the training speed, creating a complex configuration space. Based on our observations, there is no single configuration that would work for all use cases, as it would highly depend on the model size, model structure, network link bandwidth, etc. However, on individual configuration dimensions, we summarized intuitions to help application developers to quickly navigate to a small range which likely contains the optimal solution for a given use case. The specific value of the configuration needs to be determined using empirical measurements for every different deployment.

- **Communication Backend**: NCCL is considerably faster than Gloo in most use cases. When available, applications should seek to use NCCL as the primary collective communication backend.
- **Bucket Size**: Both excessively small or large bucket sizes are detrimental to communication performance. The optimal value lives in between and depends on the type of communication backend employed. The optimal bucket sizes are likely to increase with the size of the model in a sub-linear manner.
- **Resource Allocation**: There is a significant slowdown with NCCL backend when scaling models across machine boundaries, if the bandwidth across machines is considerably lower than that between same-machine GPUs. In such cases, it is recommended to keep the `DDP` group within the same machine. If the training requires larger scale, developers can explore enabling `no_sync` mode if it attains acceptable convergence speed.

6.2 Future Improvements

While we implement and maintain the `DDP` package, several ideas for improvements popped up. This section discusses the basic ideas behind those improvements.

6.2.1 Gradient Order Prediction

Although `DDP` cannot deterministically detect the backward computation order on all parameters at construction time, the order usually does not change that often in practice. One viable solution is to trace the backward order using autograd hooks and update parameter to bucket mapping accordingly. As bucket re-allocation will introduce noticeable overhead, it should be conducted infrequently. Given the existing complexities in `DDP`, tracing overhead should be negligible. Nevertheless, if there are disparities among tracing results from different iterations, additional complexities will be necessary to reach a consensus.

6.2.2 Layer Dropping

One technique to accelerate training and avoid overfitting is to randomly drop layers during the forward pass [17]. This works well with local training. As every forward pass would build a new autograd graph, those skipped layers will not participate in the backward pass either. This idea also works with `DDP`, because parameters in skipped layers can be marked as ready in the forward pass and `DDP` will not wait for their autograd hooks during the backward pass. Although `DDP` would produce the correct result, this technique alone is inadequate to accelerate distributed data parallel training the same way as local training due to the fixed parameter-to-bucket mapping. As `AllReduce` uses a bucket as the minimum granularity, it cannot judiciously react to vacancies in buckets (*i.e.*, skipped layers or parameters). Consequently, regardless of how the forward pass skips layers, there is always the same amount of data to be communicated across the wire during the backward pass. Besides, `DDP` cannot afford the luxury to adjust all buckets to cooperate with randomly skipped layers, as that would result in unacceptable memory allocation overhead. To tackle this problem, one solution is to keep bucket buffers intact but modify parameter-to-bucket mappings accordingly. Another option is to perform layer skips at the bucket level, *i.e.*, `DDP` can map layers instead of parameters to buckets and all processes skip the same bucket in the same iteration. Both options require extra coordination across all `DDP` processes, which can be implemented by using the same random seed or having an authority process to broadcast the plan.

6.2.3 Gradient Compression

Another potential improvement for `DDP` is to reduce the volume of data for communication by compressing gradients. The absolute value of gradients are usually small, which might not require `float32` or `float64` types. Current `DDP` implementation always uses the parameter type as the gradient type that can become an overkill especially when the model is approaching convergence. In this case, `DDP` would benefit from adaptive compression levels by only communicating gradients with the necessary precision. Some recent research work [34] even proposes more aggressive compression schemes, where by trading a tiny amount of model accuracy, applications can significantly accelerate distributed training by communicating just 1 bit for each gradient.

7. RELATED WORK

Distributed training algorithms can be categorized into different types from different perspectives. Below are three popular categorizations.

- Synchronous update vs Asynchronous update: With the former, all model replicas can use `AllReduce` to collectively communicate gradients or parameters, while the asynchronous scheme employs P2P communication to update gradients or parameters independently.
- Cross-iteration vs Intra-iteration: Cross-iteration parallelism (e.g., pipeline parallelism) allows the lifetime of multiple iterations to overlap with each other, while intra-iteration scheme focuses on parallelizing training within one iteration.
- Data parallel vs Model parallel: Data parallel training distributes input data to multiple model replicas, while model parallelism divides the model into smaller pieces, which is especially helpful when the model is too large to fit in one device or machine.

Table 1 summarizes some recent distributed training solutions by marking which scheme they can support. Besides advances in training schemes, prior work has also explored different communication algorithms, including tree-based `AllReduce` [23], heterogeneity-aware interconnection structure [39], and `AllReduce` decomposition [14]. As this paper focuses on `DDP`, the remainder of this section only elaborates and compares closely related techniques, *i.e.*, Synchronous, Intra-iteration, and Data parallel training schemes.

The techniques presented in this paper were first implemented and released in PyTorch v1.1. Similar computation-communication overlap techniques are also introduced in TensorFlow v2.2 as the Multi Worker Mirrored Strategy [10]. This technique is researched in academia as well. GradientFlow [37] combines bucketing `AllReduce` with skipping parameter synchronizations. Compared to PyTorch `DDP`, instead of skipping the entire synchronization step in one iteration, GradientFlow selectively communicates a subset of gradients. Although this strategy helps to reduce communication overhead for gradients, it requires an additional communication phase to attain consensus on which gradients to synchronize. As a result, the overhead incurred to acquire consensus might overshadow the speedup achieved in gradient synchronizations, especially for small models or large network round-trip delays.

Another approach to speed up distributed training is preempting and prioritizing communications based on the order of downstream computations. Jayarajan *et al.* [22] proposed to prioritize gradient synchronizations and parameter updates based on the forward order instead of the backward order, meaning that gradient buckets containing the initial layers should receive higher priorities than those in the final layers. Communications should still start from final layer gradients, as they will become ready earlier, but higher priority gradients (*i.e.*, in initial layers) can preempt lower priority ones. This design allows the forward pass in the next iteration to start sooner, even before finishing gradients communications in the previous iteration, creating more opportunities to overlap computations and communications. ByteScheduler [31] explored scheduling communications for distributed data parallel training as well. However, instead of binding with a single framework, ByteScheduler works for multiple frameworks by inserting a common core scheduler between framework APIs and framework engines and uses per-engine plugins to intercept communication invocations. To integrate with PyTorch, ByteScheduler builds on top of Horovod [35] which launches communication in the optimizer. One downside of this approach is that, there is a hard barrier between the backward pass and the optimizer step. As a result, communication can only overlap with the next forward pass instead of the current backward pass. With dynamic graphs, the next iteration might touch a different set of parameters, which would invalidate the schedule derived from the previous iteration. PACE [12] computes the optimal communication schedule and implements preemption by segmenting primitive `AllReduce` operations into smaller pieces. Although segmenting can indeed mimic preemption, it will on the other hand hurt the total communication time as we have seen in Fig. 2. A more efficient approach would be to natively support prioritization in the communication libraries (e.g., NCCL and Gloo).

The mixture of different parallelism scheme fosters even more powerful training paradigms. Mesh-TensorFlow [36] combines data parallelism with model parallelism. It vertically divides some layers by dimensions and replicating other layers where the given dimension is absent. ZeRO [32] also combines data parallelism with model parallelism, but with minimum model replication to support fast training on super large models. The authors observed that main memory consumption contributors are input data, model parameters, gradients, optimizer states, and activations. Splitting input data is trivial. However, model parameters and activations are compulsory ingredients for backward passes. ZeRO addressed this problem by partitioning parameters, gradients, and optimizer states on each `DDP` instance. Parameters are broadcast from the owner `DDP` instance to all others when necessary. Activations are recomputed during the backward pass. Compared to PyTorch `DDP`, ZeRO can scale to much larger models as each process only needs to maintain a small partition of the model. The high scalability is achieved by sacrificing the training speed, as the additional re-computation, broadcast, and gather would introduce considerable overhead. Hence, applications can choose which techniques to use based on the size of the given model and available resources. PipeDream [29] employs a different approach where the model stack is decomposed into multiple stages, where data parallelism is applied within one stage and pipeline with model parallelisms govern the workload across stages. One subtle detail is that to attain high training speed, PipeDream slightly sacrifices accuracy by using the latest gradients from multiple concurrent passes. Although the gradient might not be derived from the current parameter states, the authors show that this mismatch is tolerable in practice. Parallax [24] explored a hybrid structure that combines parameter-server [27] and collective communications. Models are partitioned based on sparsity, where dense parameters are communicated using `AllReduce` and sparse tensors are placed to parameter servers. This design avoids densifying sparse tensors and communicating empty values, which is especially helpful for NLP models.

Table 1: Distributed Training Solutions: Six schemes are **S**ynchronous-Update vs **A**synchronous-Update, **C**ross-Iteration vs **I**ntra-Iteration, **D**ata-Parallel vs **M**odel-Parallel

Scheme	S	A	C	I	D	M
PT DDP [9]	√			√	√	
PT RPC [6]		√	√	√	√	√
TF Mirrored Worker [10]	√			√	√	
TF ParameterServer [11]		√		√	√	√
Mesh TensorFlow [36]	√			√	√	√
GPipe [21]	√		√			√
Horovod [35]	√			√	√	
GradientFlow [37]	√			√	√	
SlowMo [40]		√		√	√	
PipeDream [29]	√		√	√	√	√
ZeRO [32]	√			√	√	√
Parallax [24]	√	√		√	√	√
ByteScheduler [31]	√		√	√	√	
TicTac [19]		√		√	√	√
PACE [12]	√			√	√	

8. CONCLUSION

This paper explained the design and implementation of the distributed data parallel module in PyTorch v1.5, and conducted performance evaluations on NCCL and Gloo backend using ResNet50 and BERT models. `DDP` accelerates training by aggregating gradients into buckets for communication, overlapping communication with computation, and skipping synchronizations. We also highlighted real-world caveats in gradient synchronization which are important for broad adoption. Results showed that `DDP` with NCCL backend can achieve near-linear scalability on 256 GPUs when configured properly. The measurements also revealed that the backward pass in `DDP` is the most expensive step in training and requires efforts from both framework developers to enable optimization algorithms and application developers to empirically configure the knobs. Based on our observations, we shared lessons learned from serving a variety of application, discussed potential future improvements for distributed data parallel training, and enthusiastically encourage open source community to experiment with more novel ideas.

9. REFERENCES

[1] Gloo: a collective communications library. `https://github.com/facebookincubator/gloo`, 2019.

[2] NVIDIA Collective Communications Library (NCCL). `https://developer.nvidia.com/nccl`, 2019.

[3] NVLINK AND NVSWITCH: The Building Blocks of Advanced Multi-GPU Communication. `https://www.nvidia.com/en-us/data-center/nvlink/`, 2019.

[4] Open MPI: A High Performance Message Passing Library. `https://www.open-mpi.org/`, 2019.

[5] Pybind11: Seamless operability between C++11 and Python. `https://pybind11.readthedocs.io/`, 2019.

[6] PyTorch Distributed RPC Framework. `https://pytorch.org/docs/master/rpc.html`, 2019.

[7] PyTorch Module `forward` Function. `https://pytorch.org/docs/stable/nn.html#torch.nn.Module.forward`, 2019.

[8] SciPy: open-source software for mathematics, science, and engineering. `https://docs.scipy.org/`, 2019.

[9] PyTorch DistributedDataParallel. `https://pytorch.org/docs/stable/nn.html#torch.nn.parallel.DistributedDataParallel`, 2020.

[10] TensorFlow Distributed Training MultiWorkerMirroredStrategy. `https://www.tensorflow.org/guide/distributed_training#multiworkermirroredstrategy`, 2020.

[11] TensorFlow Distributed Training ParameterServerStrategy. `https://www.tensorflow.org/guide/distributed_training#parameterserverstrategy`, 2020.

[12] Y. Bao, Y. Peng, Y. Chen, and C. Wu. Preemptive all-reduce scheduling for expediting distributed dnn training. In *IEEE INFOCOM*, 2020.

[13] M. Bojarski, D. Del Testa, D. Dworakowski, B. Firner, B. Flepp, P. Goyal, L. D. Jackel, M. Monfort, U. Muller, J. Zhang, et al. End to end learning for self-driving cars. *arXiv preprint arXiv:1604.07316*, 2016.

[14] M. Cho, U. Finkler, M. Serrano, D. Kung, and H. Hunter. Blueconnect: Decomposing all-reduce for deep learning on heterogeneous network hierarchy. *IBM Journal of Research and Development*, 63(6):1–1, 2019.

[15] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[16] M. Du, F. Li, G. Zheng, and V. Srikumar. Deeplog: Anomaly detection and diagnosis from system logs through deep learning. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1285–1298, 2017.

[17] A. Fan, E. Grave, and A. Joulin. Reducing transformer depth on demand with structured dropout. *arXiv preprint arXiv:1909.11556*, 2019.

[18] X. Guo, S. Singh, H. Lee, R. L. Lewis, and X. Wang. Deep learning for real-time atari game play using offline monte-carlo tree search planning. In *Advances in neural information processing systems*, pages 3338–3346, 2014.

[19] S. H. Hashemi, S. A. Jyothi, and R. H. Campbell. Tictac: Accelerating distributed deep learning with communication scheduling. *arXiv preprint arXiv:1803.03288*, 2018.

[20] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[21] Y. Huang, Y. Cheng, A. Bapna, O. Firat, D. Chen, M. Chen, H. Lee, J. Ngiam, Q. V. Le, Y. Wu, et al. Gpipe: Efficient training of giant neural networks using pipeline parallelism. In *Advances in Neural Information Processing Systems*, pages 103–112, 2019.

[22] A. Jayarajan, J. Wei, G. Gibson, A. Fedorova, and G. Pekhimenko. Priority-based parameter propagation for distributed dnn training. In *Proceedings of Machine Learning and Systems 2019*, pages 132–145, 2019.

[23] S. Jeaugey. Massively Scale Your Deep Learning Training with NCCL 2.4. `https://devblogs.nvidia.com/massively-scale-deep-learning-training-nccl-2-4/`, February 2019.

[24] S. Kim, G.-I. Yu, H. Park, S. Cho, E. Jeong, H. Ha, S. Lee, J. S. Jeong, and B.-G. Chun. Parallax: Sparsity-aware data parallel training of deep neural networks. In *Proceedings of the Fourteenth EuroSys Conference 2019*, pages 1–15, 2019.

[25] Y. LeCun, C. Cortes, and C. Burges. The MNIST Database. `http://yann.lecun.com/exdb/mnist/`, 1999.

[26] Y. LeCun, D. Touresky, G. Hinton, and T. Sejnowski. A theoretical framework for back-propagation. In *Proceedings of the 1988 connectionist models summer school*, volume 1, pages 21–28. CMU, Pittsburgh, Pa: Morgan Kaufmann, 1988.

[27] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B.-Y. Su. Scaling distributed machine learning with the parameter server. In *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*, pages 583–598, 2014.

[28] H. Mao, M. Cheung, and J. She. Deepart: Learning joint representations of visual arts. In *Proceedings of the 25th ACM international conference on Multimedia*, pages 1183–1191, 2017.

[29] D. Narayanan, A. Harlap, A. Phanishayee, V. Seshadri, N. R. Devanur, G. R. Ganger, P. B. Gibbons, and M. Zaharia. Pipedream: generalized pipeline parallelism for dnn training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 1–15, 2019.

[30] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[31] Y. Peng, Y. Zhu, Y. Chen, Y. Bao, B. Yi, C. Lan, C. Wu, and C. Guo. A generic communication

scheduler for distributed dnn training acceleration. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 16–29, 2019.

[32] S. Rajbhandari, J. Rasley, O. Ruwase, and Y. He. Zero: Memory optimization towards training a trillion parameter models. *arXiv preprint arXiv:1910.02054*, 2019.

[33] B. Ramsundar, P. Eastman, P. Walters, and V. Pande. *Deep learning for the life sciences: applying deep learning to genomics, microscopy, drug discovery, and more.* " O'Reilly Media, Inc.", 2019.

[34] F. Seide, H. Fu, J. Droppo, G. Li, and D. Yu. 1-bit stochastic gradient descent and its application to data-parallel distributed training of speech dnns. In *Fifteenth Annual Conference of the International Speech Communication Association*, 2014.

[35] A. Sergeev and M. D. Balso. Horovod: fast and easy distributed deep learning in TensorFlow. *arXiv preprint arXiv:1802.05799*, 2018.

[36] N. Shazeer, Y. Cheng, N. Parmar, D. Tran, A. Vaswani, P. Koanantakool, P. Hawkins, H. Lee, M. Hong, C. Young, et al. Mesh-tensorflow: Deep learning for supercomputers. In *Advances in Neural Information Processing Systems*, pages 10414–10423, 2018.

[37] P. Sun, Y. Wen, R. Han, W. Feng, and S. Yan. Gradientflow: Optimizing network performance for large-scale distributed dnn training. *IEEE Transactions on Big Data*, 2019.

[38] A. Van den Oord, S. Dieleman, and B. Schrauwen. Deep content-based music recommendation. In *Advances in neural information processing systems*, pages 2643–2651, 2013.

[39] G. Wang, S. Venkataraman, A. Phanishayee, J. Thelin, N. Devanur, and I. Stoica. Blink: Fast and generic collectives for distributed ml. *arXiv preprint arXiv:1910.04940*, 2019.

[40] J. Wang, V. Tantia, N. Ballas, and M. Rabbat. Slowmo: Improving communication-efficient distributed sgd with slow momentum. *arXiv preprint arXiv:1910.00643*, 2019.