Data Parallelism in PyTorch

ML4G - Project

Kolly Florian

Version 0.3 (September 21, 2022 11:08am +02:00) - Initial draft for first review

Contents

1 Disclaimer

This document is heavily based on the available documentation online, that is the officiel PyTorch documentation, the Huggingsface documentation and many others. All drawings are made with draw.io unless a source is specified in the caption.

All faults are my own, and no warranty is given about the content and code in this document. Some examples of code are given and briefly described in the annex section of this document. The Python version used here is 3.6.

Finally, the author would like to thank the ML4G organizers and the EffiSciences team behind for the camp where this document began.

2 Introduction to Data Parallelism

Data parallelism is a technical concept referring to scenarii in which an operation is concurrently performed on elements in a source collection. The goal is to partition the source so that multiple processes can compute on different segments at the same time. Nowadays, we mostly use GPUs and TPUs to parallelize operations. There are two major motivations for which one wants to use data parallelism:

- Firstly, today's models are too voluminous to be stored on a single GPU. Note that this trend is (at the time of writing) not slowing down, we thus expect SOTA models to get even bigger in the near future
- Secondly, we crave speed. While a model might take years to run on a single GPU (or a single machine), we want it to run in a matter of hours

2.1 First, a bit of terminology

All across this documentation and most of the ones you can find online, a few terms will often come back. Let's explain them briefly here (we will come back to them later for more discussion).

- **World**: term used to describe all the processing units we possess, not depending on which machine they run nor which process they are handling. The **world size** refers to the total number of processing unit
- **Node**: number of devices that are connected to the same backend
- **Rank**: unique ID given to a processing unit
- **Local rank**: same as the rank, but on a specific node
- **Root**: specific process considered as the main one. It is usually given rank 0, but this is no obligation

For clarity, consider the following example:

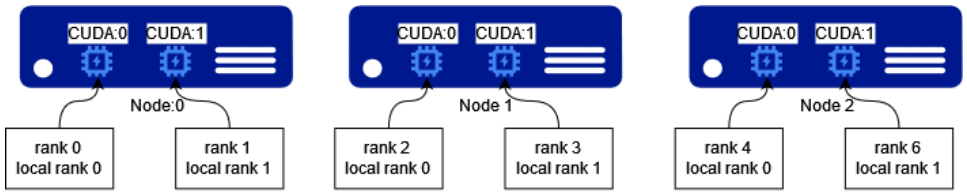


Figure 1: Take some time to compare the rank and the local rank

We have three different machine (thus 3 nodes) that each contains 2 GPUs. All of them combined give us the world, with a world-size of 6. Notice how the rank is a unique number across the entire world.

2.2 Type of parallelization

We can distinguish between different type or methods of parallelization: whether we parallelize the data, the tensor or the model. We will later come back to some methods in more detail.

- Model Parallelism (MP): the model is split up across multiple GPUs. We split along the layers, or group of layers and each GPU will handle a specific block
- Data Parallelism (DP): the same setup is replicated multiple times, and each is being fed a slice of the data. We synchronize the setups at the end of each training step
- Tensor Parallelism (TP): each tensor is split up into multiple chunks, residing separately on a designated GPU. After each shard gets processed separately, the results are synced
- Pipeline Parallelism (PP): the model is split up across multiple GPUs. Each GPU processes in parallel different stages of the pipeline on a small chunk of the batch

- Zero Redundancy Optimizer (ZeRO): somewhat similar to Tensor Parallelism, except that the whole tensor gets reconstructed in time for a forward or backward computation (no modification on the model). We will come back to this technique later in more details

Another way to distinguish between types of parallelism is called the Flynn's taxonomy. It is based on the number of concurrent instruction and data streams:

- SISD (Single Instruction Single Data): single uniprocessor machine that can execute a single instruction and fetch single data stream from memory
- SIMD (Single Instruction Multiple Data): a device that can issue the same single instruction to multiple data simultaneously (e.g. GPU)
- MISD (Multiple Instruction Single Data): architecture used for fault tolerance, where several systems act on the same data and must agree (we won't talk about it)
- MIMD (Multiple Instruction Multiple Data): multiple processors executing different instruction on different data (distributed systems)
- SPMD (Single Program Multiple Data): single program executed by a multi-processor device or cluster of devices
- MPMD (Multiple Program Multiple Data): multiple programs executed by multi-processor device or cluster of devices

2.2.1 Model Parallelism (MP)

Also called vertical data parallelization, the idea of model parallelism is to spread layers or groups of layers across multiple GPUs. In PyTorch, this is done by moving the layers (and data right before it reaches the layer) with the `.to()` method to the desired device.

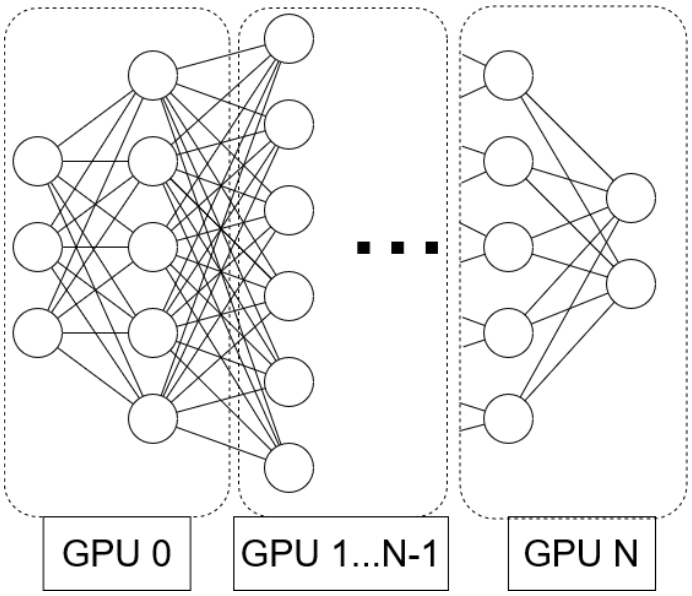


Figure 2: We split our models vertically, separating groups of layers in multiple GPUs

Note that there is an information overhead when the data transit from one GPU to another, especially if the GPUs are not on the same node. Also, to be able to compute the loss, we often need to send the data back to the first layer where the label are once the last layer is done.

The main advantage of model parallelism is that it allows for bigger model to be trained, as the GPUs share the memory load. However, all but one GPU is idle at any given time and it has an overhead of copying the data between the devices (especially if they are not on the same node). Remark that it is better to have a 24GB card instead of 4x6GB cards using this parallelism paradigm.

2.2.2 Pipeline parallelism (PP)

This technique is very similar to the previous one, except that we artificially introduce a pipeline to use all GPUs concurrently. Let's compare a typical forward and backward pass. First, here is the flow for a simple model parallelism:

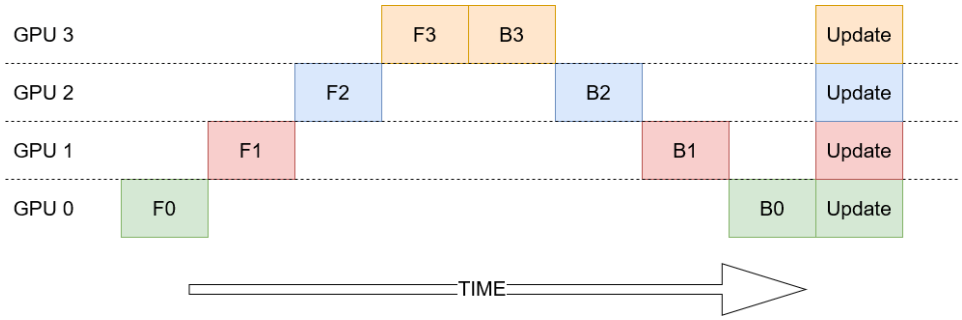


Figure 3: Notice the poor use of each GPU

Pipeline parallelism creates mini-batches of data to keep most of the GPUs running most of the time.

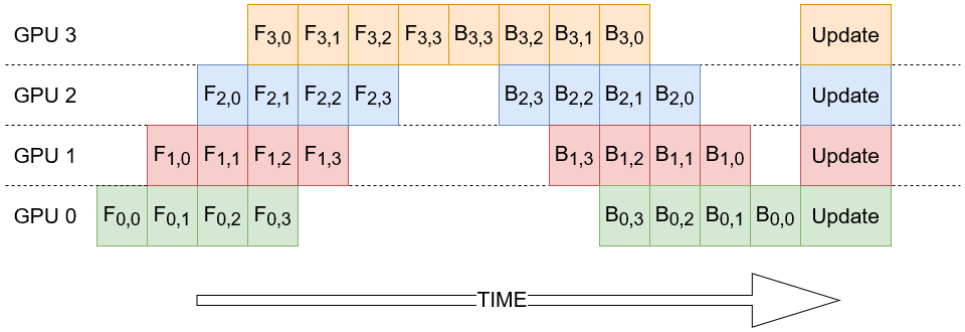


Figure 4: This is much better. Take a moment to look at the indices!

The idle parts are referred to as the "bubble". This method introduces a new parameter, the *chunks*, that define how many chunks of data are sent in a sequence through the same pipe stage (the chunk is 4 in the image above). Adding a layer of data parallelism over the pipeline parallelism, we need to distinguish between mini-batches and micro-batches. The global data is first split into mini-batches as usual (e.g. a batch size of 1024 gets split up into 4 mini-batches of 256 each). The pipeline then splits each mini-batches by the number of chucks to obtain a micro-batch (e.g. 32 chunks yield micro-batches of size 8, continuing on the example above). Each stage works with a single micro-batch at a time. One needs to be careful when choosing those sizes: too small and the GPUs won't be used enough, too big and the micro-batches become too tiny.

2.2.3 Tensor parallelism (TP)

In tensor parallelism, each GPU processes only a slice of a tensor. The full tensor is aggregated only for operations that require the whole thing. Let's take a quick detour through linear algebra to better understand it.

If we have two matrices for which we want to find the dot product, it is possible to split either column-wise or row-wise. For example, the following product

$$\underbrace{\begin{pmatrix} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \end{pmatrix}}_{A} \cdot \underbrace{\begin{pmatrix} 8 & 9 \\ 10 & 11 \\ 12 & 13 \\ 14 & 15 \end{pmatrix}}_{B} = \underbrace{\begin{pmatrix} 76 & 82 \\ 252 & 274 \end{pmatrix}}_{Y}$$

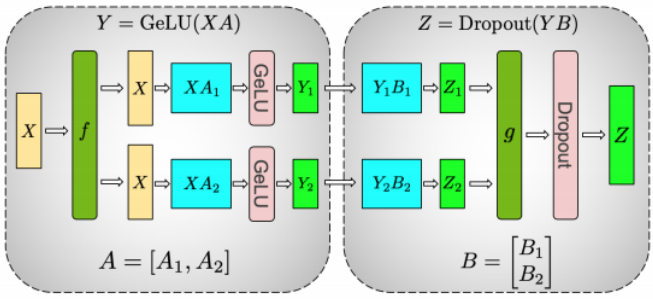
can also be done splitting matrix B column-wise:

$$Y_1 = \begin{pmatrix} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \end{pmatrix} \cdot \begin{pmatrix} 8 \\ 10 \\ 12 \\ 14 \end{pmatrix} = \begin{pmatrix} 76 \\ 252 \end{pmatrix}$$
$$Y_2 = \begin{pmatrix} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \end{pmatrix} \cdot \begin{pmatrix} 9 \\ 11 \\ 13 \\ 15 \end{pmatrix} = \begin{pmatrix} 82 \\ 274 \end{pmatrix}$$
$$Y = Y_1 \parallel Y_2$$

and by splitting matrix A by rows and matrix B by columns before summing the results:

$$Y_1 = \begin{pmatrix} 0 & 1 \\ 4 & 5 \end{pmatrix} \cdot \begin{pmatrix} 8 & 9 \\ 10 & 11 \end{pmatrix} = \begin{pmatrix} 10 & 11 \\ 82 & 91 \end{pmatrix}$$
$$Y_2 = \begin{pmatrix} 2 & 3 \\ 6 & 7 \end{pmatrix} \cdot \begin{pmatrix} 12 & 13 \\ 14 & 15 \end{pmatrix} = \begin{pmatrix} 66 & 71 \\ 170 & 183 \end{pmatrix}$$
$$Y = Y_1 + Y_2$$

Using these techniques, we can then split a weight matrix A column-wise across N GPUs and perform the matrix multiplications in parallel. It is also possible to feed them into an activation function independently.



(a) MLP

Figure 5: Example of tensor parallelism, through the activation function (credit: https://huggingface.co/docs/transformers/v4.16.2/en/parallelism)

We need to synchronize only at the end, where the output vector has to be reconstructed.

One major issue unfortunately is that tensor parallelism requires a fast transfer of data. It is thus not advisable to do tensor parallelism over more than one node.

2.2.4 Combinations

There are many ways to combine the parallelism tools seen above. Some will be detailed in greater details later. Usually, these methods are implemented by known frameworks such as DeepSpeed, Megatron-LM, SageMaker or OSLO.

2.2.5 Which methods should I use?

It is almost impossible to give a straight answer, as the best techniques to use will always depend on many factors such as the size of the model, the compute time available, the compute ressources at disposition and of course the price one is willing to pay to train his model. However, here are a few guidelines:

- Single GPU
    1. Model fits onto a single GPU: no parallelization necessary

2. Model doesn't fit onto a single GPU: ZeRO + Offload CPU
3. Largest layer doesn't fit into a single GPU: Memory Centric Tiling*

- Single Node / Multi-GPU
  1. Model fits onto a single GPU: Distributed DP / ZeRO
  2. Model doesn't fit onto a single GPU: PP / ZeRO / TP
  3. Largest layer doesn't fit into a single GPU: ZeRO / TP
- Multi-node / Multi-GPU
  1. ZeRO (simplest) / PP+TP+DP (massive changes to the model)

* Memory Centric Tiling allows to run arbitrarily large layers by automatically splitting them and executing them sequentially.

2.3 Parallelization in PyTorch

In PyTorch, the `torch.distributed` package contains much of the utilities necessary to begin working with parallelized systems. The official documentation categorizes the package into three main components:

- Distributed Data-Parallel Training (DDP): the model is replicated on every process and every model replica will be fed with a different set of input data. The gradient is synchronized on every replicas, along with its computation to speed up the process.
- RPC-Based Distributed Training (RPC): RPC supports training structures such as distributed pipeline parallelism, parameter sever paradigm for examples. Its main avantage is the extension of the autograd engine beyond machine-boundaries.
- Collective Communication (c10d): this library is the API on which DDP and RPC are built (resp. collective communication and P2P). It allows for finer-grain control over what is communicated between the replicas but in exchange gives up the performance optimizations.

2.3.1 Data Parallel Training Paradigm

When working on projects that are expected to gradually grow in complexity, a common development strategy is as follows:

1. Single-machine single-GPU training: default behaviour
2. Single-machine multi-GPU training: DataParallel module
3. Single-machine multi-GPU distributed training: DistributedDataParallel module
4. Multi-machine multi-GPU distributed training: DistributedDataParallel module, additional scripting required
5. Dynamic multi-machine multi-GPU distributed training: elastic package

3 Data-Parallel Training

`nn.parallel.DataParallel` is the simplest package to allow parallelism during the training process, as it requires only minor additional code (one line is often enough). However, it does not offer the best performance as it replicates the model in every forward pass. The major issue is that it is a single-process multi-thread parallelism that bounds the speed due the infamous GIL single-lock.

3.1 Using DP on a model

The package `DataParallel` handles everything on its own. To allow data parallelization on a model in PyTorch, one must simply tell PyTorch to use the module on a given model. We can also indicate which devices to use using the `device_ids` parameter:

```
model = nn.DataParallel(model, device_ids=[0,1])
```

4 Distributed Data-Parallel Training

While being harder to setup and use, the package `nn.parallel.DistributedDataParallel` is a far more common choice than simply using `nn.parallel.DataParallel`. This is due to two main advantages of `DistributedDataParallel`:

- Each process maintains its own optimizer and performs a complete optimization step with each iteration. Thus, no parameter broadcast step is needed, reducing the tensors transfer time
- Each process contains an independent Python interpreter, thus avoiding the GIL single-lock and making heavy use of the Python runtime

The second advantage is that `dataParallel` is multithreading while `distributedDataParallel` is multiprocessing. This can make a huge difference in Python.

4.1 Understanding the backends

The `torch.distributed` package supports three built-in backends that allow the transport of data across multiple processes and/or machines. This part is important only for people who want or need to dig deeper into the parallelism world, and can be skipped otherwise.

4.1.1 MPI

MPI (Message Passing Interface) is a library standard for programming distributed memory. MPI is portable and fast, as it has been implemented and optimized on almost all architectures and hardwares. It has language bindings for Fortran, C and C++ and several MPI modules exist in Python.

MPI model consists of communications among processes through messages called *inter-process communication*. All the nodes execute the same program but each have a unique ID (rank), allowing the user to select which part should be executed by which process. This is very important to understand: the program is the same across the nodes! To ask a specific device to go through a part of the code, one can use its rank in a condition before.

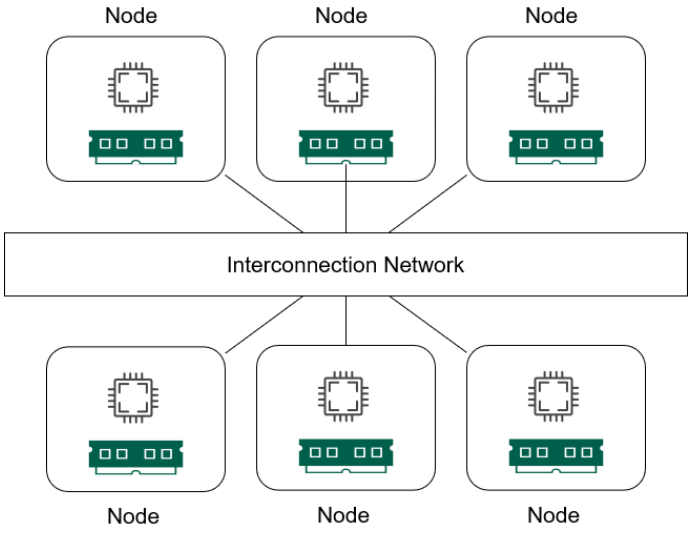


Figure 6: Multiple nodes, each with a compute capabilities and memory, are linked by an interconnection network

MPI Communicator An MPI communicator is a "communication universe" for a group of processes. The default MPI communicator is called `MPI_COMM_WORLD` and represents the collection of all processes. Almost every MPI command needs to provide a communicator as argument.

Handling the process rank As seen at the beginning, a rank is used to distinguish a process from the others: it is a unique integer value assigned to a process within a communicator. Here is a classical way to setup and use ranks with MPI (in C):

```
int size, rank;
MPI_Comm_size(MPI_COMM_WORLD, &size);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
// The root node has usually rank 0
if(rank == 0) {...}
```

Initiating and closing a computation Among the most important MPI commands are those used for initiating and closing a computation:

```
int main(int argc, char** argv) {
  MPI_Init(&argc, &argv); // initiates an MPI computation
  ...
  MPI_Finalize(); // terminates the MPI computation and cleans up
}
```

Synchronization Often, parallel algorithms require that no process continues on before all the processes have reached a certain state at certains points in the program. For that, it is possible to ask for explicit synchronization:

```
int MPI_Barrier(MPI_Comm comm);
```

Blocking communication In MPI, communication consists of sending a copy of the data to another process. On the sender side, the communication requires to know the rank of the receiver, the data type, the size and the location where the message needs to be sent. The receiver does *not* need to know who the sender is, but only the data type, size and what storage location to use to put the resulting message.

More strictly speaking, MPI defines a function that performs blocking send:

```
int MPI_Send(const void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)
```

where `buf`, `count` and `datatype` define the message buffer (the *data*), `dest` and `comm` identify the destination process. The `tag` parameter is optional. The last three described are collectively called the *envelope* The message gets sent and the buffer empties out before the function exits. On the other end, a function performs a blocking receive:

```
int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm,
  MPI_Status *status)
```

This function waits for a message from `source` and `comm` and once received stores it into the buffer defined by `buf`, `count` and `datatype`. One can used `MPI_ANY_SOURCE` to receive from any source and/or `MPI_ANY_TAG` to receive from any tag. Note that receiving fewer `datatype` elements than `count` is fine, but receiving more results in an error. The `status` is a useful structure that contains information about (for example) the source (`status.MPI_SOURCE`) or the tag (`status.MPI_TAG`).

Non-blocking communication Of course, non-blocking communications are also possible: the following methods return immediately, and the computation and communication can be overlapped:

```
int MPI_Isend(const void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm,
   MPI_Request *request)
int MPI_Irecv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm,
  MPI_Request *request)
```

Collective communication Collective communication refers to the process of synchronizing and moving data or creating pool of collective computation. It involves all the processes within the scope of a communicator. By default, all the processes are in the `MPI_COMM_WORLD` communicator, but it is possible to define groups of communicators.

- Broadcasting: share data from the process with rank root to all other processes of the communicator including the root process

```
int MPI_Bcast(void *buf, int count, MPI_Datatype datatype, int root, MPI_Comm comm)
```

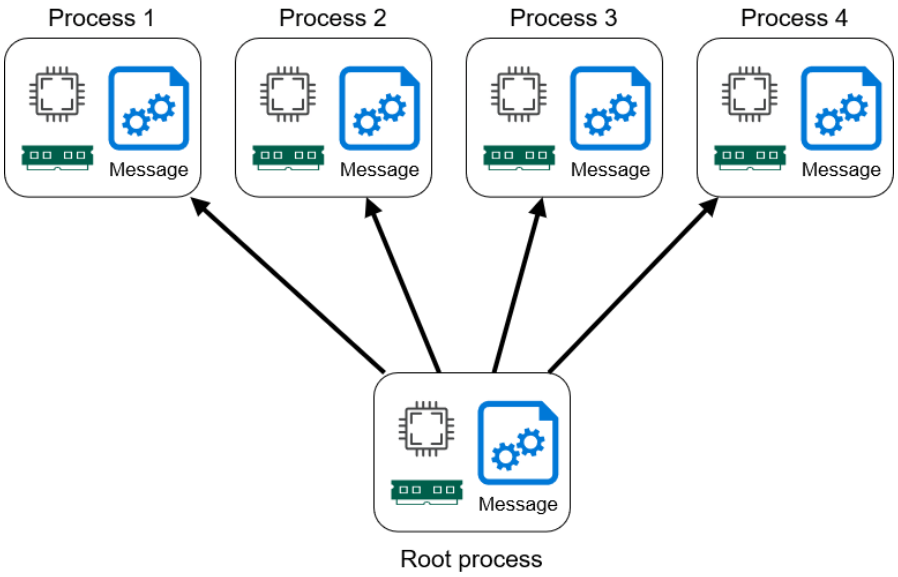


Figure 7: Broadcasting data

- Scattering: send a fraction of data to each node

```
int MPI_Scatter(const void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf,
int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)
```

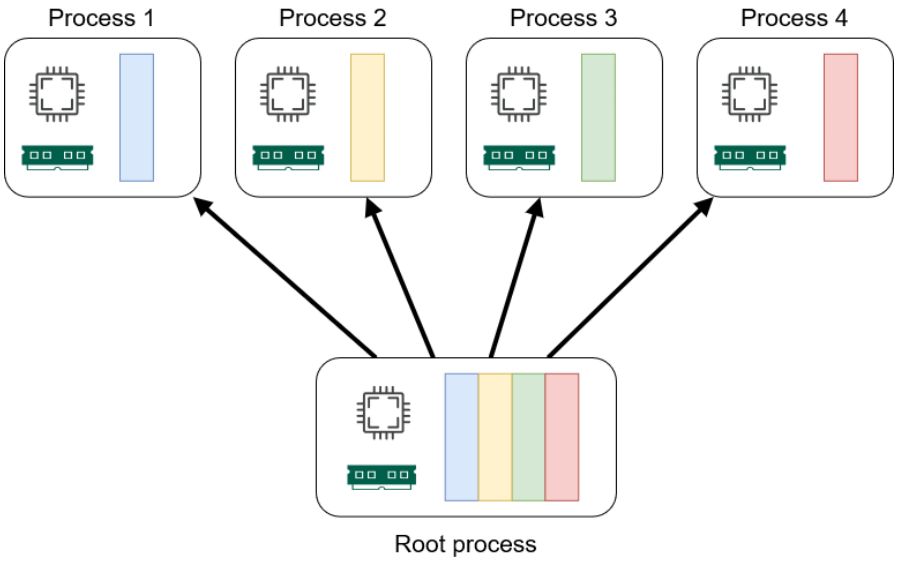


Figure 8: Scattering data

- Gathering: retrieve a portion of data from different processes

```
int MPI_Gather(const void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf,
int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)
```

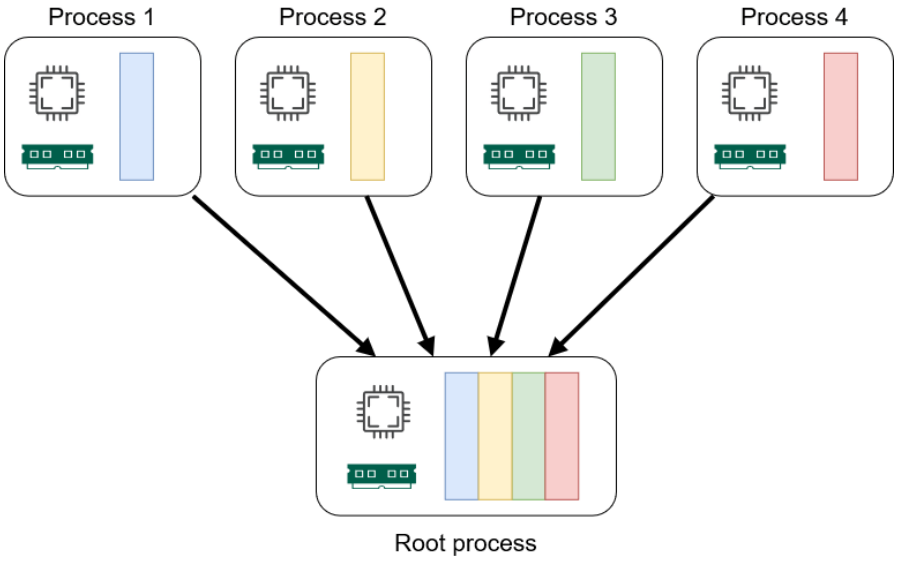


Figure 9: Gathering data

- Reducing: retrieve a portion of data from different processes and manipulate it directly

```
1    int MPI_Reduce(const void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op
      op, int root, MPI_Comm comm)
2
```

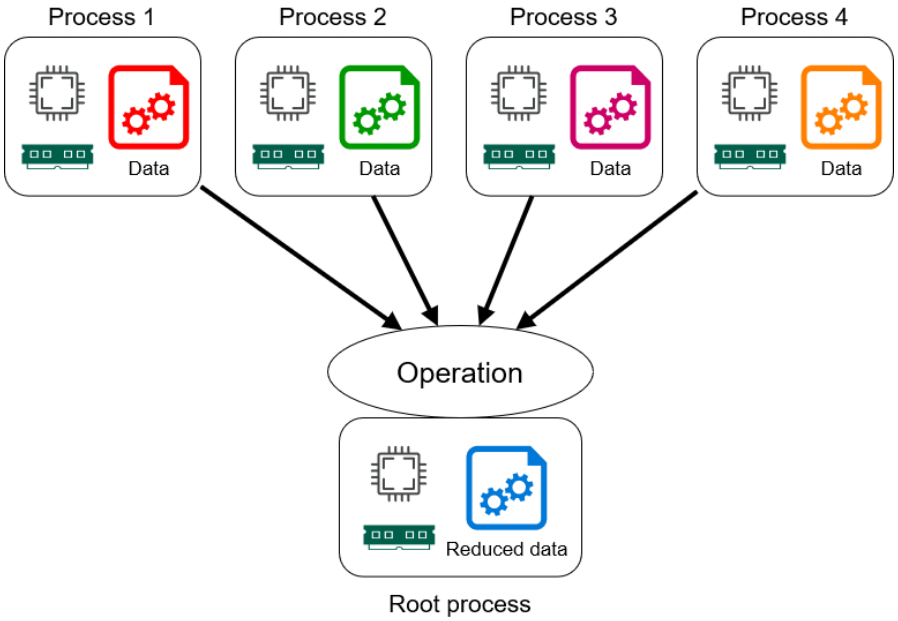


Figure 10: Reducing data

4.1.2 NCCL

NCCL (for NVIDIA Collective Communications Library, pronounced "Nickel") is a library providing inter-GPU communication functions (called primitives). Like MPI, it implements both point-to-point send/receive and collective communication primitives. However, contrary to MPI, it is not a parallel programming framework, but aims at accelerating inter-GPU communication. NCCL implements each collective communication primitives in a single kernel handling both communications and computation operations, allowing for fast synchronization and minimizing the resources needed to reach peak bandwidth. This tool works on any multi-GPU parallelization model (e.g. single-threaded control of all GPUs, multi-threaded (one thread per GPU), multi-process with MPI, ...).

NCCL possesses a C API, accessible from multiple programming languages and follows the MPI collectives API. The most important modification is the presence of a `stream` argument providing direct integration with the CUDA programming model.

The collective communication primitives implemented by NCCL are the following:

- AllReduce
- Broadcast (see MPI broadcast)
- Reduce (see MPI reduce)
- AllGather
- ReduceScatter

Let's draw the schema for the primitives that haven't been covered yet:

- All reducing: same as reduce, but the result is stored in all processes

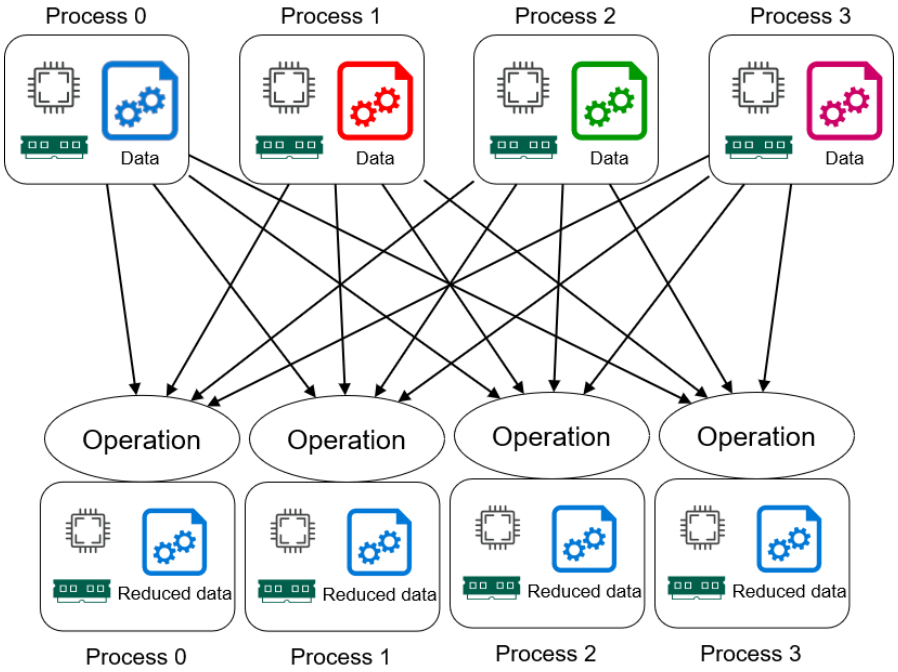


Figure 11: All reducing data

- All gathering: retrieve a portion of data from all processes to all processes

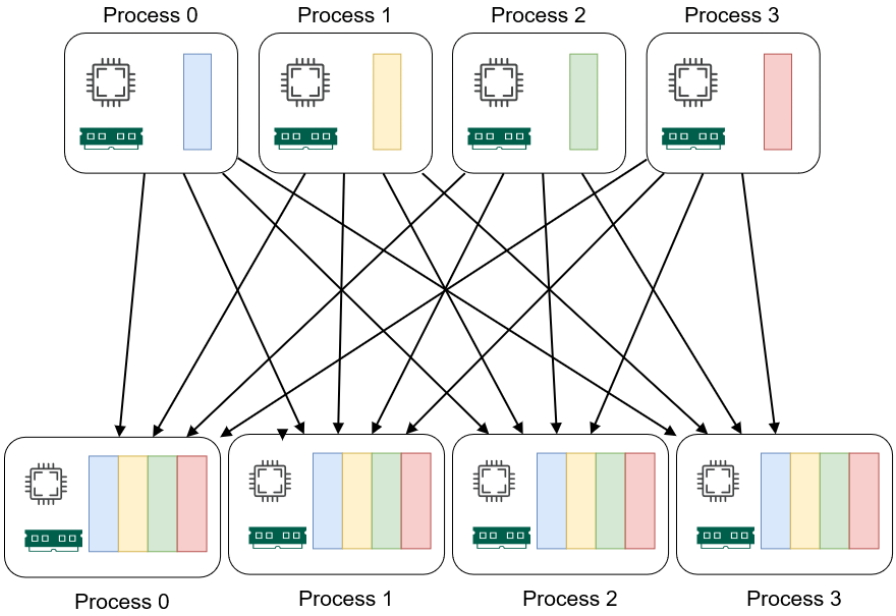


Figure 12: All gathering data

- Reduce scattering: same as reduce, except the result is scattered in equal blocks among ranks

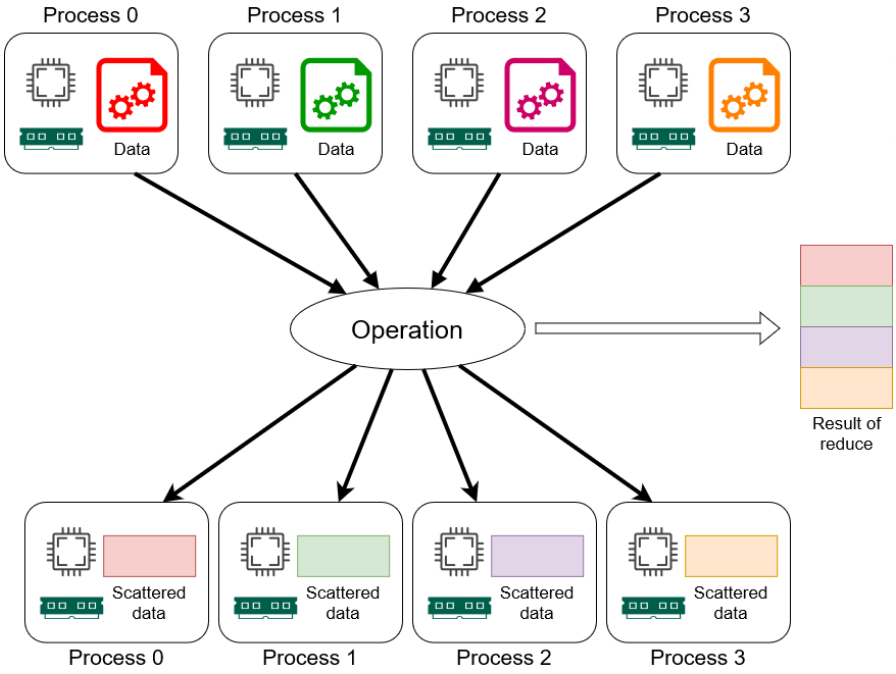


Figure 13: Reduce scattering data

A bit more theory into how NCCL works NCCL is a topology-aware ring-based library, implemented as monolithic CUDA C++ kernels with three primitive operations (Copy, Reduce and ReduceAndCopy). Ring-based means that a GPU is connected to its following one, forming a ring-like structure:

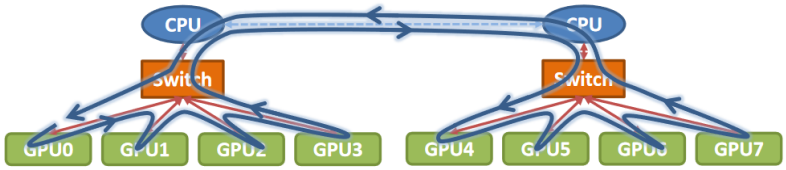


Figure 14: Example of a ring-based collective (credit: https://images.nvidia.com/events/sc15/pdfs/NCCL-Woolley.pdf)

4.1.3 Gloo

Gloo is a collective communications library hosted by the Meta Incubator (yes, the same Meta as Facebook). Its library offers many distributed algorithms: when applicable, these algorithms have an implementation that works with system memory buffers, and one that works with Nvidia GPU memory buffers (using CUDA and NCCL). In the latter case, it is not necessary to copy memory between host and device.

The first thing the Gloo framework does is called the *rendezvous*: the nodes find each other and setup connections between them (full mesh, i.e. bidirectional communication, or some subset). A `gloo::Context` class retains information about all the participating processes (the ranks) and the persistent communication channels. Gloo does not maintain global state or thread-local state, meaning one can setup as many contexts as needed!

The algorithms available in Gloo are the following:

- AllReduce (see NCCL above)
- Reduce-Scatter (see NCCL above)
- Broadcast (see MPI above)
- pairwise_exchange (see MPI above)

Note that it is possible to use an existing MPI communicator to create the *rendezvous* context (key/value). Gloo won't use it after.

4.1.4 Let's recap

The last sections described the three backends usable with PyTorch (under certain conditions), that is MPI, NCCL and Gloo. Their implementation of the primitives can differ a lot, thus affecting the speed of computation. There is also a difference in what primitives are implemented. It is therefore impossible to tell the reader which one to use, it all falls down on the use case.

NVIDIA offers a great summary picture for their backend NCCL. You can also find some of the primitives of the other backends:

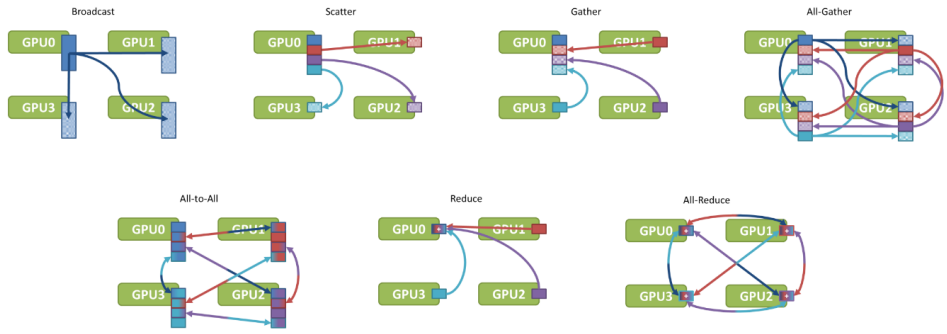


Figure 15: NVIDIA NCCL primitives (credit: https://images.nvidia.com/events/sc15/pdfs/NCCL-Woolley.pdf)

All of these said, let's go back to PyTorch and spend some time writing distributed applications!

4.2 Distributed applications in PyTorch

If the reader wants to focus on writing distributed Machine Learning models, this section can be skipped. In this section, we will hide the choice of backend behind PyTorch to study some basic communication application. As seen previously, each backend provides different primitives and thus we have to be careful when using the PyTorch distributed abstraction. The chart below shows which backend supports which method: feel free to refer to it later!

Backend	`gloo`		`mpi`		`nccl`	
Device	CPU	GPU	CPU	GPU	CPU	GPU
send	✓	χ	✓	✓	χ	✓
recv	✓	χ	✓	✓	χ	✓
broadcast	✓	✓	✓	✓	χ	✓
all_reduce	✓	✓	✓	✓	χ	✓
reduce	✓	χ	✓	✓	χ	✓
all_gather	✓	χ	✓	✓	χ	✓
gather	✓	χ	✓	✓	χ	✓
scatter	✓	χ	✓	✓	χ	χ
reduce_scatter	χ	χ	χ	χ	χ	✓
all_to_all	χ	χ	✓	✓	χ	✓
barrier	✓	χ	✓	✓	χ	✓

4.3 The `distributedDataParallel` module

The most important function in this package is the constructor itself. This container parallelizes the application of the given module by splitting the input across the specified devices by chunking in the batch dimension. The module is replicated on each machine, and every replica handles a portion of the input. During the backwards pass, gradients from each node are averaged. This means that the batch size should be larger than the number of GPUs used locally.

```
from torch.nn.parallel import DistributedDataParallel as DDP
ddp_model = DDP(model, device_ids=[rank])
```

4.3.1 Torch `multiprocessing` package

The `torch.multiprocessing` package wraps the native `multiprocessing` module of Python. I won't enter into details here, instead I want to introduce a very important function, the `spawn` method:

```
torch.multiprocessing.spawn(func, args=(), nprocs, join, daemon, start_method='spawn')
```

As its name entails, this method *spawns* (i.e. creates or generates) `nprocs` processes that each run the given function `func` with the arguments `args`. The function is called as `func(i, *args)`, where `i` is the index of the process (sounds like the rank, doesn't it?) and where the `args` tuple is deconstructed to give each element as a parameter directly. The `join` argument allows to perform a blocking join on all processes. If set to `false`, a `ProcessContext` is returned by the method and the joining has to be done manually. Note that the last argument `start_method` is deprecated, you should not change its default value.

4.3.2 Initiating and destroying the group

Each process has to know which distributed process group it is a part of. For that, a good practice is to create a setup function to prepare the environment and initiate the group. This function is run for each process. For instance:

```
def setup(rank, world_size):
  os.environ['MASTER_ADDR'] = 'localhost'
  os.environ['MASTER_PORT'] = '12355'
  dist.init_process_group("gloo", rank=rank, world_size=world_size)
```

This very simple example sets the master's IP address and port, and prepares the process giving it the backend used, its rank and the world size. The method used is the following:

```
torch.distributed.init_process_group(backend, init_method, timeout, world_size, rank, store,
  group_name, pg_options)
```

In the example above, we initialize a process group explicitly, but it is also possible to specify an `init_method` which indicates how to discover the peers. The `store` parameter, mutually exclusive with the `init_method` serves as the key/value store accessible to all workers and is used to interchange connection and/or address information.

At the end of the training, it is good practice to destroy the process group and deinitialize the distributed package;

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

4.3.3 Complete (simple) example

Let's go through a complete example of a model that uses the `DistributedDataParallel` package:

```
# Imports and hyperparameters ...

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

    dist.init_process_group("gloo", rank=rank, world_size=world_size)

def cleanup():
    dist.destroy_process_group()

class Net(nn.Module):
    # Model ...

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

    model = Net().cuda(rank)
    ddp_model = DDP(model, device_ids=[rank])
    torch.cuda.set_device(rank)

    train_dataset = datasets.MNIST('./data/', download=True, train=True, transform=transform)
    train_sampler = torch.utils.data.distributed.DistributedSampler(train_dataset, num_replicas=
  world_size, rank=rank, shuffle=False)
    train_loader = torch.utils.data.DataLoader(dataset=train_dataset, batch_size=batch_size,
  shuffle=False, pin_memory=True, sampler=train_sampler)

    criterion = nn.NLLLoss()
    optimizer = optim.SGD(ddp_model.parameters(), lr=learning_rate)

    for _ in range(epochs):
        # Training process ...
        # Don't forget here to move the tensors to the right process using the .cuda(rank) method!

    cleanup()

def run(fn, world_size):
    mp.spawn(fn, args=(world_size,), nprocs=world_size, join=True)

if __name__ == "__main__":
    world_size = torch.cuda.device_count()
    run(main, world_size)
```

4.4 Let's simplify our work with DeepSpeed

Glossary

GIL Lock that allows only one thread to hold the control of the interpreter. 7

multiprocessing Multiple processors run concurrently, where each processor can run one or more threads. 8

multithreading A single processor execute multiple threads concurrently. 8

SOTA for State Of The Art, refers to the best models , in a specific subdomain or generally. 3

5 Annex

In this section, I will write and briefly describe codes that can be useful in order to attain a better understanding of the backends and the parallelization in PyTorch. I recommend the reader to take some time to experiment and play with them.

5.1 MPI

5.1.1 Simple send-receive peer-to-peer exchange

This first example consists of a simple exchange from a sender to a receiver. We send a number across the devices: rank 0 (root) is the sender and rank 1 will receive and print the result.

```c
#include <mpi.h>
#include <stdio.h>

int main(int argc, char **argv) {
    MPI_Init(NULL, NULL);
    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    int size;
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    int number;
    if(rank == 0) {
        number = 43523;
        MPI_Send(&number, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
    } else if(rank == 1) {
        MPI_Recv(&number, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        printf("Process 1 received number %d from process 0\n", number);
    }
    MPI_Finalize();
}
```

It is important here to note that the node with rank 1 has not seen what the number is initialized to: one must be careful about what each process knows from the code itself or from what it receives from other nodes.

5.1.2 Approximating π using multiple processes

Let's go through a quite fun example. We are here trying to approximate π using the Taylor series expansion for arctan(1), using the fact that $\pi = 4 \cdot \frac{\pi}{4} = 4 \cdot \arctan(1)$.

$$\arctan(1) = \sum_{n=0}^{\infty} \frac{(-1)^n}{2n+1}$$

The idea is that each process will compute one element of this sum. We then use `MPI_Reduce` with the sum operation to compute $\frac{\pi}{4}$ and multiply it by 4. The result is sent to the root process defined in the head of the code, and the root process is asked to print the result.

```c
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>
#include <math.h>

#define ROOT 0

double taylor(const int i, const double x, const double a) {
    int sign = pow(-1, i);
    double num = pow(x, 2 * i + 1);
    double den = a * (2 * i + 1);
    return (sign * num / den);
}

int main(int argc, char *argv[]) {
    int nodes, rank;
    double* partial;
```

```
18          double res;
19          double total = 0;
20
21          MPI_Init(&argc, &argv);
22          MPI_Comm_size(MPI_COMM_WORLD, &nodes);
23          MPI_Comm_rank(MPI_COMM_WORLD, &rank);
24
25          res = taylor(rank, 1, 1);
26          printf("rank=%d total=%f\n", rank, res);
27
28          MPI_Reduce(&res, &total, 1, MPI_DOUBLE, MPI_SUM, ROOT, MPI_COMM_WORLD);
29
30          if(rank == ROOT)
31              printf("Total is = %f\n", 4*total);
32
33          MPI_Finalize();
34      }
```

5.2 NCCL

TODO: NCCL code is not working on my computer (GPUs crashing)

5.3 PyTorch DDP

6 Sources

TODO: make that readable and not ugly

- Huggingface parallelization
- PyTorch distributed overview
- PyTorch DDP tutorial
- PyTorch distributed tutorial
- MPI lesson
- NVIDIA NCCL introduction slides