



Deep Learning Optimised on Jean Zay

Distribution – Data parallelism



IDRIS



DLO-JZ course
Commented slides
Author: Myriam Peyrounette
June 2023

Distributed training

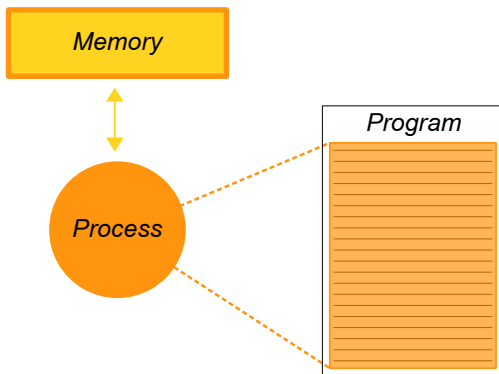
General knowledge about parallel computing ◀

Data parallelism to distribute your training ◀

2

This chapter begins by laying the foundations of parallel computing. In terms of distributed training, the high-level libraries such as TensorFlow and PyTorch offer ready-to-use tools which handle the parallelization but mask the problematics of distribution. The idea here is to understand a little better what is happening under the hood.

This introduction enables us to subsequently consider the distribution problematics of a neural network training on multiple GPUs and compute nodes. Here, we present the "data parallelism" technique which is the most approachable and most used.

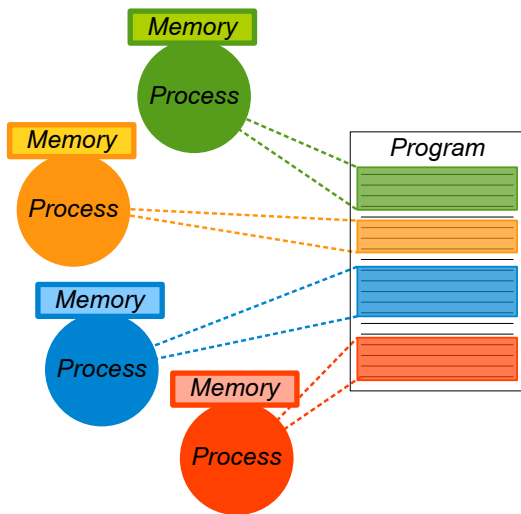


Sequential execution

- Only one process executes the program.
- The variables defined in the program are stored in the memory allocated to the process.
- One process executes the code on one physical compute unit (CPU core or GPU).

When a program is launched, a group of instructions is enacted. The sequence of instructions is stored in memory and executed by a physical computation unit (CPU core or GPU graphics card). This computation instance is called “process”.

A code is sequentially executed when a single process executes the instructions, one after another.



Parallel execution with distributed memory

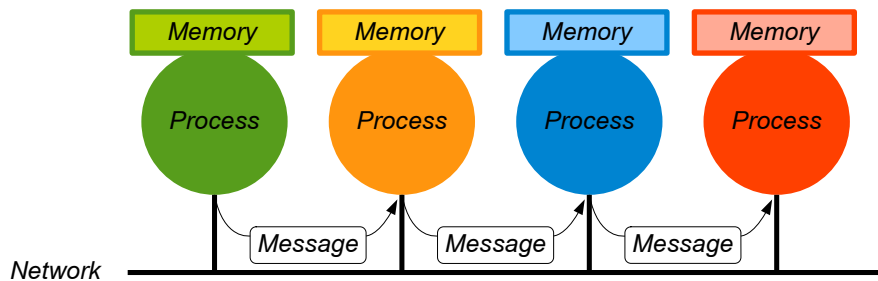
- Several processes execute the code at the same time (multi-processing).
- The variables defined in the program are private, they are stored in the local memory allocated to each process.
- It is possible that the processes execute separate parts of the code.

Multiple processes can be solicited to execute a program. The processes can take charge of different parts of the program in order to accelerate its global execution time. This is parallel execution.

In the case of parallel execution on a distributed memory system, an independent memory space is associated with each process. A process does not have access to variables stored in the memory which are allocated to another process.

Parallel execution with distributed memory

- To share information, the processes can send each other **messages** through the interconnection network.
- These **communications** are managed by libraries such as **MPI** or **NCCL**.

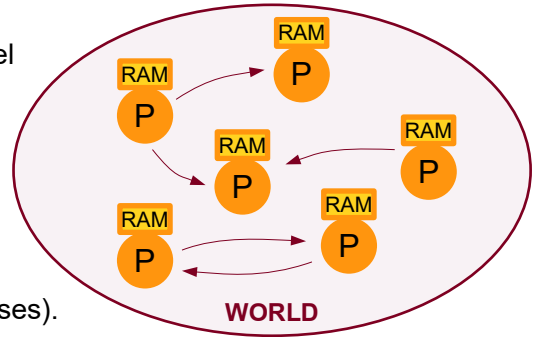


In distributed memory, process synchronization or information sharing steps are inevitable. For example : IO, overall error calculation, etc.

Processes can communicate with each other by sending messages via the interconnection network. These communications are handled by specialized libraries such as MPI (CPU) or NCCL (GPU, NVIDIA).

In a parallel code based on the MPI or NCCL library,

- The set of processes exists in a common parallel **environment** initialized at the beginning of the execution.
- From the initialization to the destruction of this parallel environment, all the processes read and execute the program.
- During the initialization of the environment, a **communicator WORLD** is created to allow the set of processes to communicate with each other.
- The communicator has a **size** (the number of processes).
- Within a communicator, each process can be identified by its **rank**.



At the beginning of the execution, the library handling the communications initializes a parallel environment by creating an overall communicator: the WORLD communicator. This contains all of the processes allocated to the execution. Each of these processes is identified by its rank.

The inter-process communication mechanisms are also initialized and managed by the libraries: communication pipelines (ports, addresses), communication buffers (temporary memory management when a communication is initiated), etc.

- Concrete examples based on the Horovod library
- The **Horovod** library is designed to ease the implementation of Deep Learning distributed training

Communication libraries	
• MPI (on CPU)	✓
• NCCL (on GPU)	✓

Deep learning frameworks	
• TensorFlow	✓
• Keras	✓
• PyTorch	✓
• Apache MXNet	✓

To understand the behavior of a program executed in parallel, we will present some simple examples.

These examples are based on the use of Horovod, a library developed to facilitate the distribution of Deep Learning training.

Horovod uses the MPI or NCCL communication libraries in backend: MPI for the inter-CPU communications, and NCCL for the inter-GPU communications.

In the examples which follow, the instructions are on CPU for the sake of simplicity and easy reading but all the concepts covered are transposable on GPU.

Example 1: Each process reads and executes the code lines

- Parallelized code using Horovod:

```
import horovod.torch as hvd
hvd.init()
size = hvd.size()
print(f'The communicator size is {size}')
```

- Execution on 4 processes (#SBATCH --ntasks=4):

```
$ srun --ntasks=4 <...> python script.py
The communicator size is 4
The communicator size is 4
The communicator size is 4
The communicator size is 4
```

Example 1 :

The `hvd.init()` initializes the parallel environment at the start of execution. A WORLD communicator is created.

To launch a script in parallel on Jean Zay, a number of processes > 1 must be defined via Slurm using the `--ntasks` option and the script must be launched via the `srun` command.

The example message: Each process reads and executes the program independently.

Example 2: We identify the processes thanks to their ranks

- Parallelized code using Horovod:

```
import horovod.torch as hvd
hvd.init()
size = hvd.size()
rank = hvd.rank()
print(f'I am proc {rank} among {size}')
```

- Execution on 4 processes (#SBATCH --ntasks=4):

```
$ srun --ntasks=4 <...> python script.py
I am rank 1 among 4
I am rank 3 among 4
I am rank 2 among 4
I am rank 0 among 4
```

Example 2 :

Example message: Each process is identified by its rank within the communicator.

We see here that each process contains a private variable, rank, a distinct value per process.

Example 3: The processes can be assigned different tasks according to their ranks

- Parallelized code using Horovod:

```
import horovod.torch as hvd
hvd.init()
size = hvd.size()
rank = hvd.rank()
print(f'I am proc {rank}, my rank is {"even" if rank%2==0 else "odd"}')
```

- Execution on 4 processes (#SBATCH --ntasks=4):

```
$ srun --ntasks=4 <...> python script.py
I am proc 2, my rank is even
I am proc 0, my rank is even
I am proc 1, my rank is odd
I am proc 3, my rank is odd
```

10

Example 3 :

Example message: We can make a line of code do different things on each process depending on the local variables that each process has.

Example 4: The processes can be assigned different tasks according to their ranks

- Parallelized code using Horovod:

```
import horovod.torch as hvd

hvd.init()

size = hvd.size()
rank = hvd.rank()

if rank%2==0:
    print(f'I am proc {rank}, my rank is even')
else:
    print(f'I am proc {rank}, my rank is odd')
```

- Execution on 4 processes (#SBATCH --ntasks=4):

```
$ srun --ntasks=4 <...> python script.py
I am proc 2, my rank is even
I am proc 3, my rank is odd
I am proc 0, my rank is even
I am proc 1, my rank is odd
```

11

Example 4 :

Example message: We can attribute different tasks per process by making certain instructions dependent on the rank number of the process.

Example 4: Parallelization of a compute loop

↓Process 0↓

- Initial state of the memory

```
N = 4
a = [0, 1, 2, 3]
b = [4, 5, 6, 7]
c = [0, 0, 0, 0]
sum = 0
```

- Program

```
for i in range(N) :
    c[i] = a[i] + b[i]
    sum += c[i]
```

- Final state of the memory

```
N = 4
a = [0, 1, 2, 3]
b = [4, 5, 6, 7]
c = [4, 6, 8, 10]
sum = 28
```

Example 4 :

Sequential execution of a computation loop.

Example 4: Parallelization of a compute loop

- Initial state of the memory

↓Process 0↓

```
N = 4
a = [0, 1, 2, 3]
b = [4, 5, 6, 7]
c = [0, 0, 0, 0]
sum = 0
```

↓Process 1↓

```
N = 4
a = [0, 1, 2, 3]
b = [4, 5, 6, 7]
c = [0, 0, 0, 0]
sum = 0
```

- Program

```
size = hvd.size()
rank = hvd.rank()

istart = rank * N / size
iend = (rank+1) * N / size

for i in range(istart, iend) :
    c[i] = a[i] + b[i]
    sum += c[i]
```

- Final state of the memory

```
N = 4
a = [0, 1, 2, 3]
b = [4, 5, 6, 7]
c = [4, 6, 0, 0]
sum = 10
```

```
N = 4
a = [0, 1, 2, 3]
b = [4, 5, 6, 7]
c = [0, 0, 8, 10]
sum = 18
```

13

Parallelization of the computation loop + vision of a parallel execution on 2 processes.

1/ To parallelize a loop, we distribute the loop indexes on the different processes. It is important for this to be the most equitable distribution possible. ($iend - istart$ ideally identical on each process) in order to balance the computation load on all the processes and to optimize the computation acceleration.

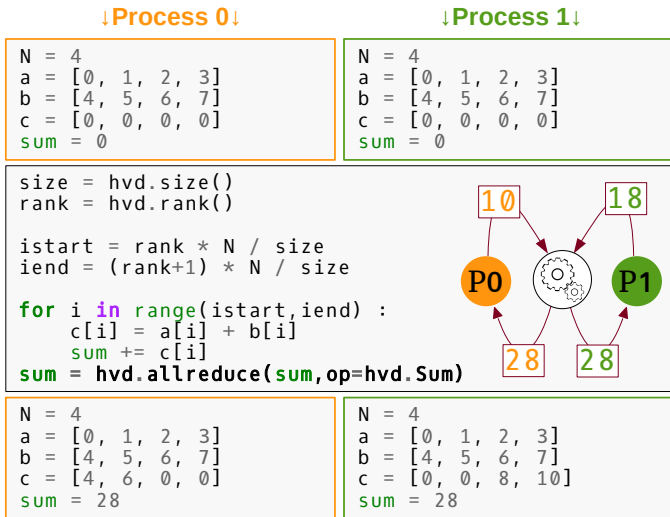
2/ We see the impact of the private variables:

- Vector `c` is only partially filled by processes. If we consider that the interest of this program is to compute the value of `sum`, we can ignore it. A memory gain is even possible by setting the size of vector `c` for $iend - istart$ on each process.

- The `sum` value is incorrect, and differs depending on the process, because each process only reads a part of vector `c`. An inter-process communication is necessary.

Example 4: Parallelization of a compute loop

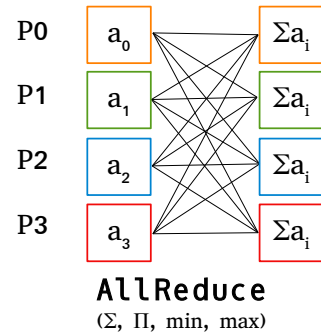
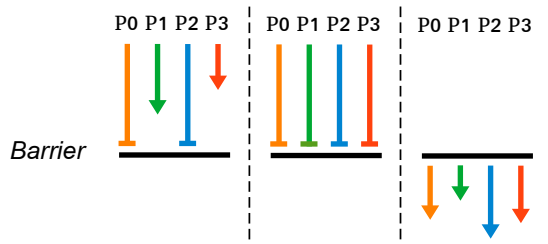
- Initial state of the memory



Parallelization of a computation loop + vision of a parallel execution on 2 processes + inter-GPU communication

The AllReduce communication operation enables performing a reduction operation (sum, product, minimum, maximum) on a value held by all of the communicator processes. The result of this operation is recuperated by all of the processes.

- Inter-process communication `AllReduce()`
 - Collective communication
 - Synchronization barrier



costly communication

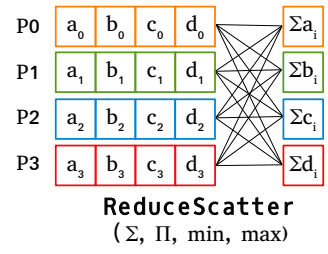
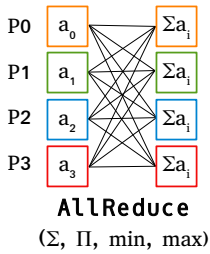
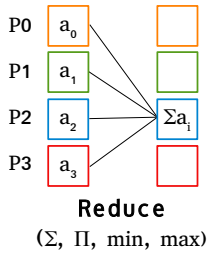
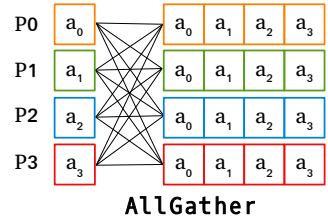
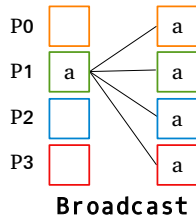
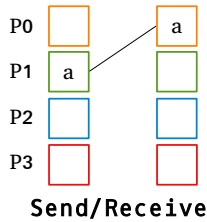
AllReduce is a collective communication: It solicits all the processes.

Here, all the processes must both send and receive a piece of information (value sent = a_i , value received = Σa_i).

This communication represents a synchronization barrier: All the processes must wait before continuing their computation.

This synchronization can represent an important loss of time if the computing load is unevenly distributed between the processes.

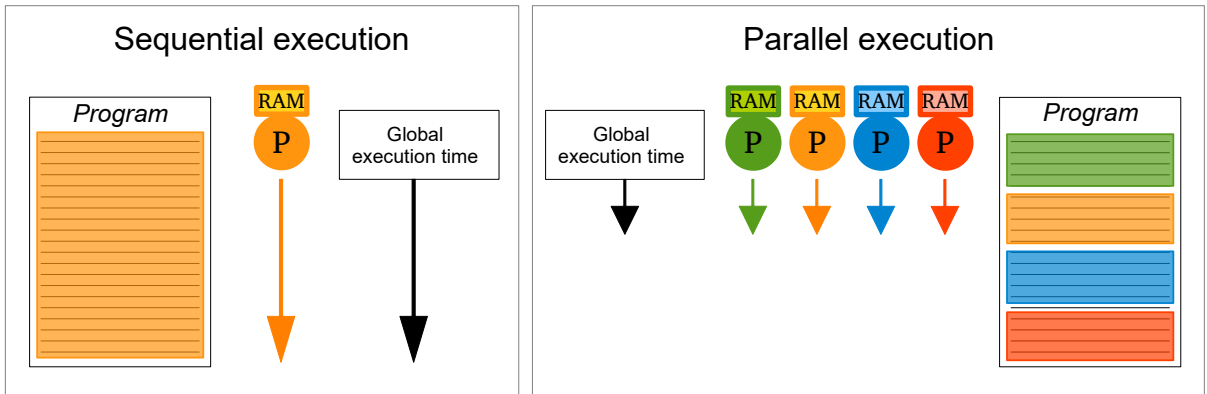
- NCCL communications



A quick overview of the inter-process communications offered by the NCCL library.

SEND / RECV = point-to-point operations soliciting only two processes.

The other operations are collective.



- Global execution time:
$$T(N \text{ procs}) = \frac{T_{\text{paral}}(1 \text{ proc})}{N} + T_{\text{seq}} + T_{\text{comm}}$$

How much time gain to expect from the parallelization of a program?

Ideally, to solve the same problem, a parallelized code goes N times faster on N processes than a sequential.

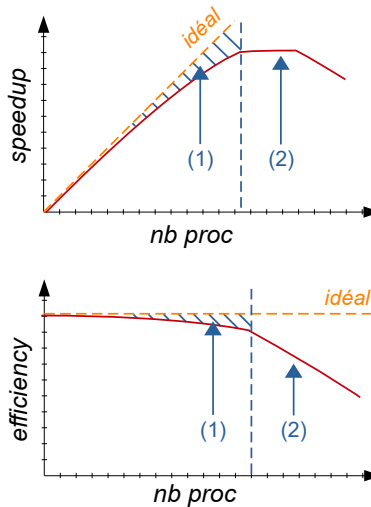
In practice:

- Certain code parts are not parallelizable and stay sequential. This cost is inherent to the code structure.
- Inter-process communications have a cost (initialization of buffers, physical time of message propagation on the interconnection network, ...). This cost becomes potentially prohibitive when the number of processes is too large for a given computation load.

- Scalability study: $\frac{T(1\text{ proc})}{T(N\text{ procs})}$

- **Strong scaling**
(problem size is constant)

- **Weak scaling**
(problem size is proportional to the number of processes)



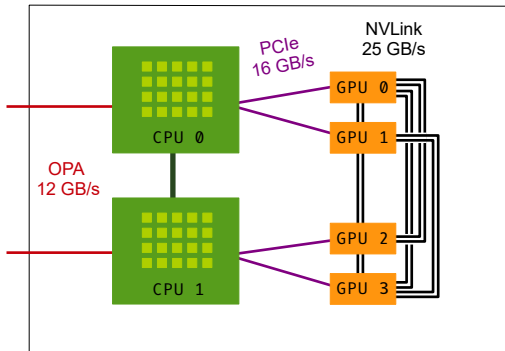
(1) impact of the sequential parts of the code
(2) cost of the inter-process communications

To estimate the impact of parallelization on a code, it is advised to conduct a scalability study. This study links the execution time of the code in parallel to the number of processes.

Strong scaling: We maintain a fixed problem size and we trace the speedup based on the number of processes. Tracing this scaling shows us the optimal number of processes to define for a given problem.

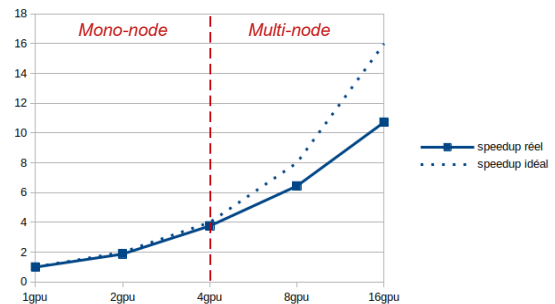
Weak scaling: We increase the problem size proportionally to the number of processes and we trace the efficiency based on the number of processes. Tracing this scaling shows us the maximum problem size which the code is capable of processing.

- Bandwidths of the interconnection networks on Jean Zay:



Node 4 × V100 16GB

Flaubert benchmark



Strong scaling

The cost of communications on Jean Zay.

Different types of interconnection networks quadri-GPU V100 nodes:

- NVLink ~ 25 GB/s → intra-node
- PCIe ~ 16 GB/s → CPU/GPU
- OPA ~ 12 GB/s → inter-nodes

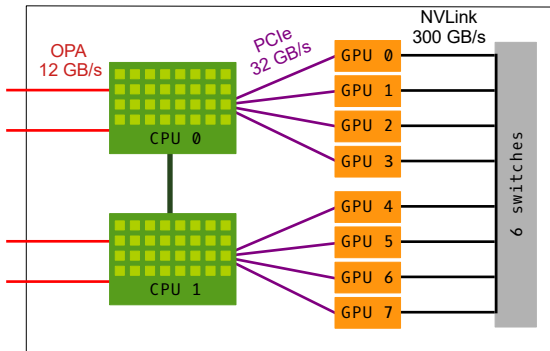
Conclusion: Good intra-node scalability expected; less good inter-node scalability.

On the right, example of strong scaling of the “Flaubert” classification bench (fine-tuning). Executed on Jean Zay. Illustration of the divergence at the moment of changing to multi-nodes.

Distribution: General knowledge about parallel computing

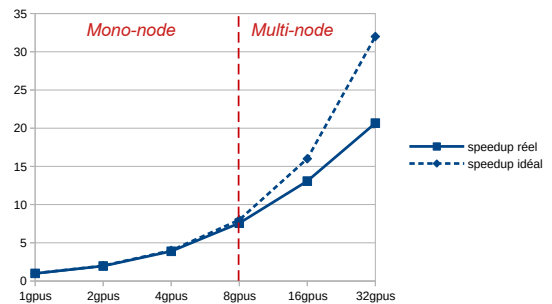


- Bandwidths of the interconnection networks on Jean Zay:



Node 8 × A100 80GB

Flaubert benchmark



Strong scaling

The cost of communications on Jean Zay.

Different types of interconnection networks of **octo-GPU 80GB A100 nodes**:

- NVLink ~ 300 GB/s → intra-node
- PCIe ~ 32 GB/s → CPU/GPU
- OPA ~ 12 GB/s → inter-nodes

Conclusion: Good intra-node scalability expected; less good inter-node scalability.

On the right, example of strong scaling of the “Flaubert” classification bench (fine-tuning). Executed on Jean Zay. Illustration of the divergence at the moment of changing to multi-nodes.

Distributed training

General knowledge about parallel computing ◀

Data parallelism to distribute your training ◀

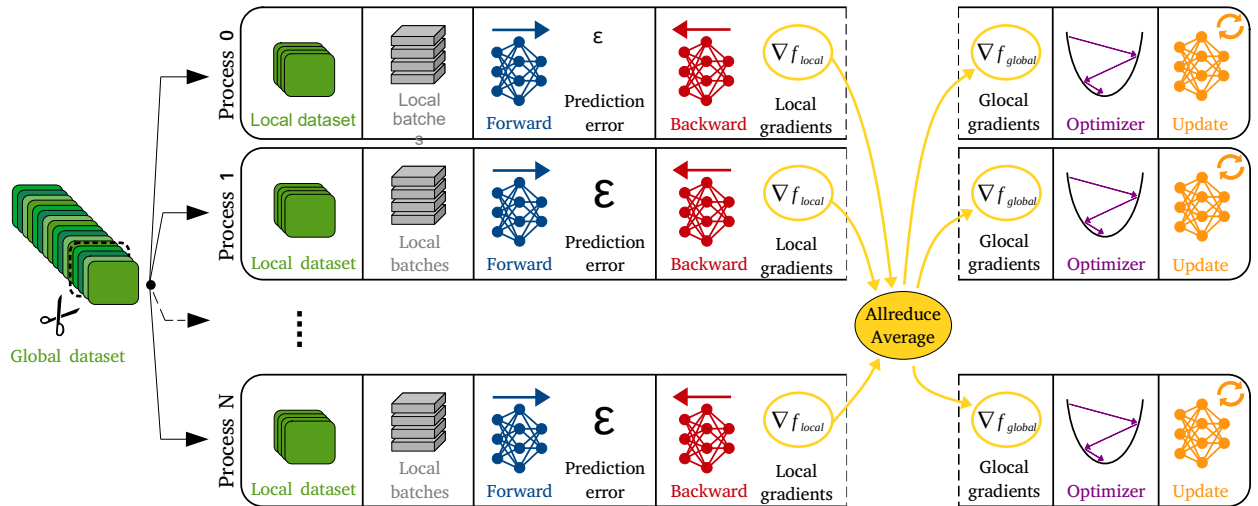
- **Data parallelism**
 - Training time speedup
 - Model small enough to be contained on one GPU in memory
 - Causes large batches
(consequences on the training quality)

The main objective of data parallelism is to speed up the training.

Here, the model will be replicated on each GPU. Therefore, the model must be contained in memory on 1 GPU.

The consequence of data parallelism is the augmentation of batch size in proportion to the number of GPUs used. This implies adjusting certain parameters of the model. This will be presented in the following chapters.

Distribution: Data parallelism



Data parallelism consists of equally distributing the dataset on all the available processes so that each process treats only one part of the data.

The batches created per process are called local batches here (i.e. contained in memory by only one process). The propagation and backpropagation steps are effectuated from these local batches, resulting in local gradient computation.

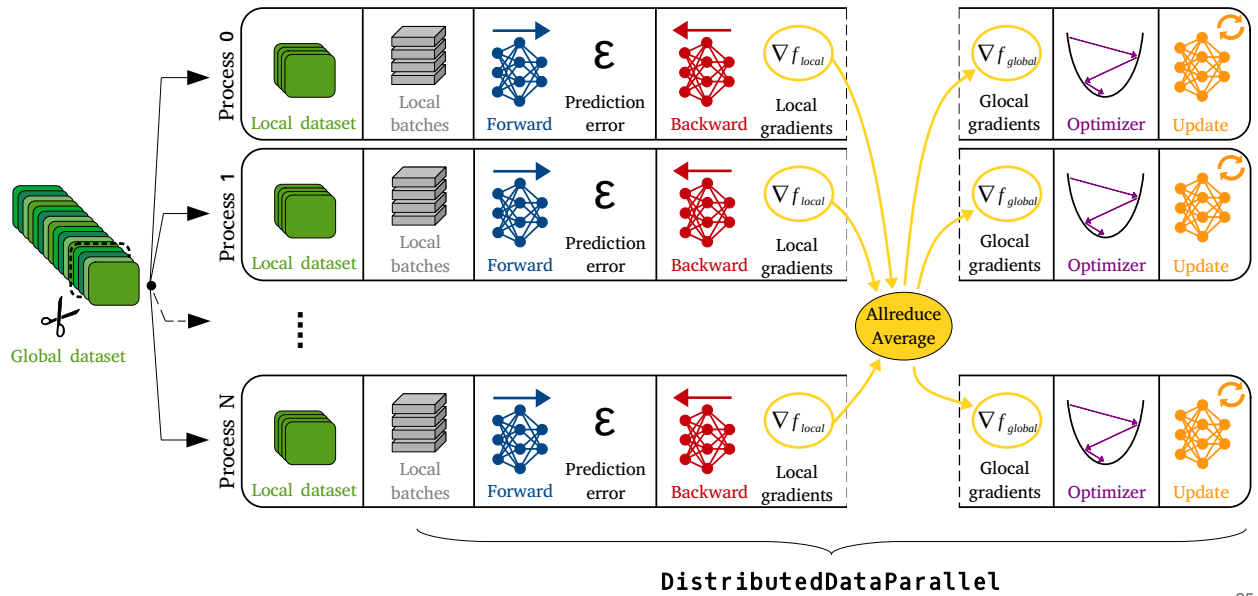
The local gradients are averaged on all the processes through AllReduce, a collective communication reduction. The model parameters are updated based on these overall gradients.

Important: In data parallelism, a training iteration implies N local batches in parallel with N representing the number of active processes. The overall batch size used for this training is, therefore, $N \times \text{batch_size_per_gpu}$.

- Implementation of the data parallelism
 - PyTorch → **DistributedDataParallel** (integrated solution)
 - TensorFlow → `MultiWorkerMirroredStrategy` (integrated solution)
 - Horovod (external library)

The data parallelism is implemented in classes within the PyTorch or TensorFlow libraries. It is also possible to use Horovod in these two frameworks.

Distribution: Data parallelism



We will focus on the PyTorch `DistributedDataParallel` class.

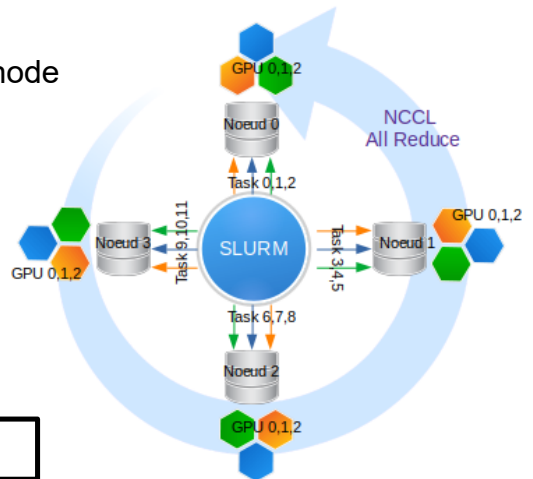
Distribution: Data parallelism

- Execution of the parallel code → Slurm environment

Distribution example on : 4 nodes
 3 GPUs per node

```
## Slurm script
#SBATCH --nodes=4           # nb nodes
#SBATCH --ntasks=12        # nb proc
#SBATCH --ntasks-per-node=3 # nb proc / node
#SBATCH --gres=gpu:3        # nb GPUs / node

srun python script.py
```

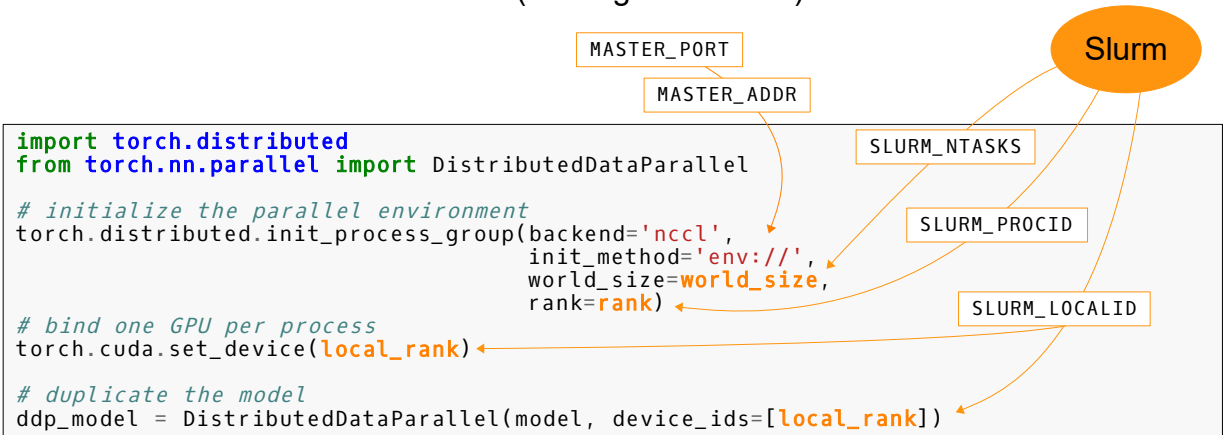


Each GPU must be binded with one process.

Example of a Slurm configuration for the parallel execution of a training.

Important: In data parallelism, one process is attached to each GPU and the script is launched with the `srun` command.

- **DistributedDataParallel** (training distribution)



Implementing data parallelism with `DistributedDataParallel`.

The parallel environment is initialized by the call to the `init_process_group()` function.

NCCL is the backend to use on a GPU architecture.

The `MASTER_ADDR` (name of the compute node which will be designated as “master” during the inter-node communications) and `MASTER_PORT` (port number of the “master” node chosen between 10000 and 20000 arbitrarily) variables must also be defined in the computing environment.

The characteristic values of the parallel environment (`world_size`, `rank` and `local_rank`) are recovered from the Slurm environment by the corresponding environment variables.

- `DistributedDataParallel` (training distribution)
- `idr_torch.py` script from IDRIS

```
# idr_torch.py
import os
import hostlist

# get SLURM variables
size = int(os.environ['SLURM_NTASKS'])
rank = int(os.environ['SLURM_PROCID'])
local_rank = int(os.environ['SLURM_LOCALID'])
cpus_per_task = int(os.environ['SLURM_CPUS_PER_TASK'])

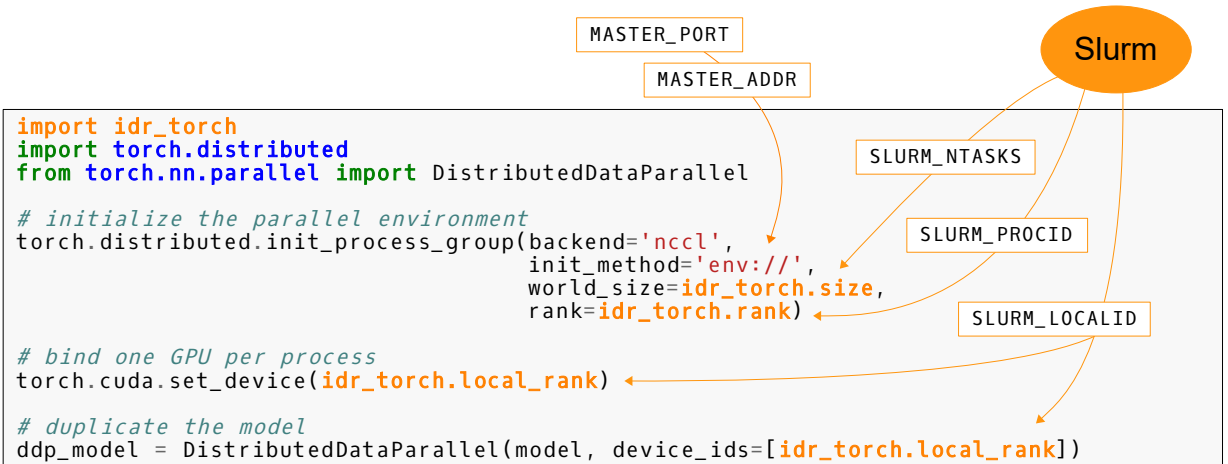
# get node list from slurm
hostnames = hostlist.expand_hostlist(os.environ['SLURM_JOB_NODELIST'])

# get IDs of reserved GPU
gpu_ids = os.environ['SLURM_STEP_GPUS'].split(",")

# define MASTER_ADDR & MASTER_PORT
os.environ['MASTER_ADDR'] = hostnames[0]
os.environ['MASTER_PORT'] = str(12345 + int(min(gpu_ids)))
```

IDRIS developed the `idr_torch` model to facilitate the interfacing with Slurm. It enables recovering the values of the `size`, `rank` and `local_rank` environment variables and to define the `MASTER_PORT` and `MASTER_ADDR` environment variables.

- **DistributedDataParallel** (training distribution)



Using the module `idr_torch` module.

- What about torchrun? → Possible but cumbersome.

```
[...]
#SBATCH --ntasks-per-node=1
[...]

GPUS_PER_NODE=8

MASTER_ADDR=$(scontrol show hostnames $SLURM_JOB_NODELIST | head -n 1)
MASTER_PORT=15000

CMD="train.py --arg1 1 --arg2 2"

export LAUNCHER="torchrun --nproc_per_node $GPUS_PER_NODE \
                --nnodes $SLURM_NNODES \
                --rdzv_backend c10d \
                --rdzv_endpoint $MASTER_ADDR:$MASTER_PORT"

srun bash -c "$LAUNCHER --node_rank $SLURM_PROCID $CMD"
```

slurm.sh

```
[...]
parser.add_argument("--local_rank", type=int, help="Local rank. Necessary for using torchrun.")
[...]
WORLD_RANK = int(os.environ['RANK'])
LOCAL_RANK = int(os.environ['LOCAL_RANK'])
WORLD_SIZE = int(os.environ['WORLD_SIZE'])
```

train.py

30

Using the launcher torchrun is possible on Jean Zay but cumbersome.

"#SBATCH --ntasks-per-node=1"

Only one task must be launched per node, then torchrun spawns the processes on each node for you (1 process per GPU).

"GPUS_PER_NODE=8"

The number of GPUs per node must correspond to the current compute configuration.

"--nproc_per_node \$GPUS_PER_NODE --nnodes \$SLURM_NNODES"

Each node must be aware of the number of GPUs per node and the total number of nodes involved.

"--rdzv_backend c10d --rdzv_endpoint \$MASTER_ADDR:\$MASTER_PORT"

Each node must use the same MASTER_ADDR and MASTER_PORT to be able to communicate.

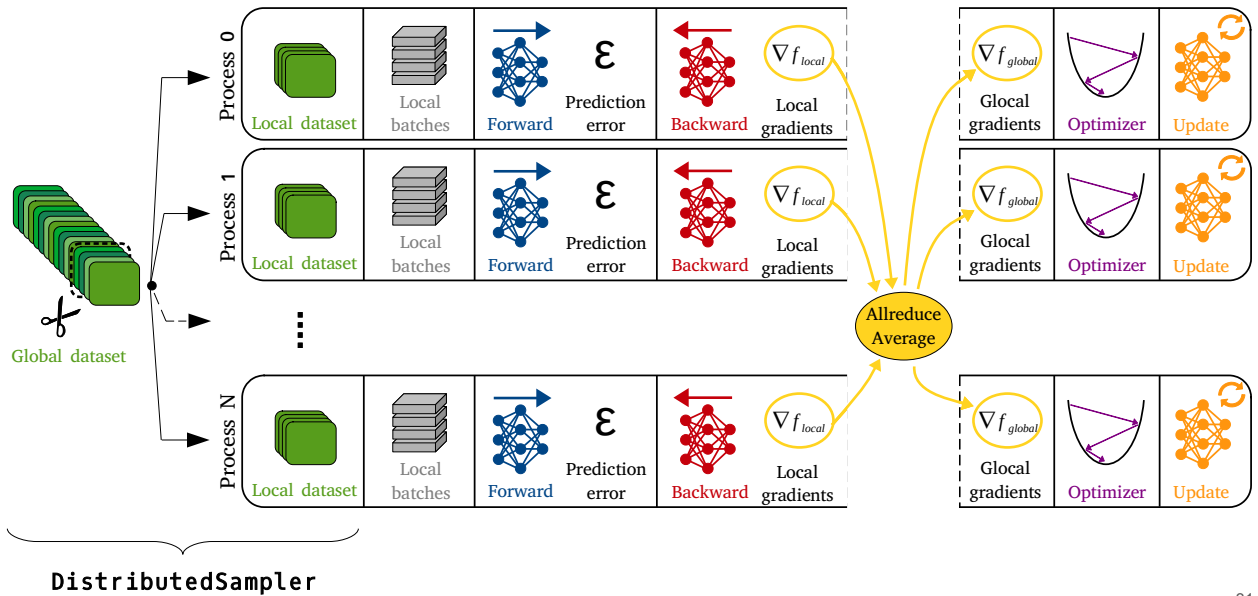
"--node_rank \$SLURM_PROCID"

Each node must have a unique rank.

A new argument `--local_rank` must be defined in your training script.

New environment variables `WORLD_SIZE`, `RANK` and `LOCAL_RANK` are created by torchrun and can be used in the training script.

Distribution: Data parallelism



Distributing the dataset on all the available processes is managed by the `DistributedSampler` class.

- `DistributedSampler` (distributing the input data)

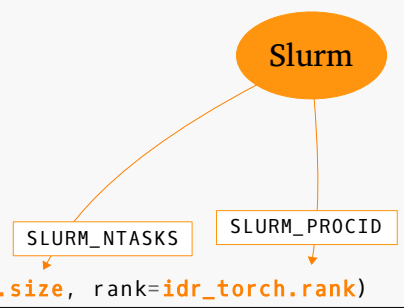
```
import idr_torch
import torch.distributed
from torch.nn.parallel import DistributedDataParallel
from torch.utils.data.distributed import DistributedSampler

# initialize the parallel environment
[...]
```

bind one GPU per process
[...]

duplicate the model
[...]

distribute the input data
`data_sampler = DistributedSampler(dataset, shuffle=True,`
 `num_replicas=idr_torch.size, rank=idr_torch.rank)`



```
graph TD
    Slurm((Slurm)) --> SLURM_NTASKS[SLURM_NTASKS]
    Slurm --> SLURM_PROCID[SLURM_PROCID]
```

The shuffling step is assigned to the `Sampler`.

Implementing `DistributedSampler`.

`DistributedSampler` will distribute the dataset to all of the processes according to the size and rank parameters.

Important: The shuffling step (random reorganization of the data indexes) is attributed to the `Sampler` and not to the `DataLoader`.

- **DistributedSampler** (distributing the input data)
- Parallel execution on 4 processes using the `DistributedSampler`:

```
dataset = [1, 2, ..., 100]  
batch_size_per_gpu = 5  
ntasks = 4  
batch_size=20
```

```
$ srun --ntasks=4 <...> python script.py  
Rank 0: Batch 0 = tensor([ 1,  5,  9, 13, 17])  
Rank 1: Batch 0 = tensor([ 2,  6, 10, 14, 18])  
Rank 2: Batch 0 = tensor([ 3,  7, 11, 15, 19])  
Rank 3: Batch 0 = tensor([ 4,  8, 12, 16, 20])
```

Example of using `DistributedSampler`.

- **DistributedSampler** + shuffling
 - The index shuffling is performed by each GPU from a common seed.
 - Parallel execution on 4 processes using the `DistributedSampler`:

```
dataset = [1, 2, ..., 100]
batch_size_per_gpu = 5
ntasks = 4
batch_size=20
```

```
$ srun --ntasks=4 <...> python script.py
Rank 0: Batch 0 = tensor([46, 36, 80, 17, 23])
Rank 1: Batch 0 = tensor([16, 64, 97, 12, 59])
Rank 2: Batch 0 = tensor([91, 18, 49, 24, 4])
Rank 3: Batch 0 = tensor([33, 73, 37, 81, 63])
```

Example of using `DistributedSampler` with random reorganization of the indexes.

- **DistributedSampler** + shuffling
- The index shuffling is performed by each GPU from a common seed.

```
for epoch in range(1,30):  
    for i, batch in enumerate(data_loader):  
        ...
```



```
>>> Epoch 1  
Rank 0: Batch 0 = tensor([46, 36, 80, 17, 23])  
Rank 1: Batch 0 = tensor([16, 64, 97, 12, 59])  
Rank 2: Batch 0 = tensor([91, 18, 49, 24, 4])  
Rank 3: Batch 0 = tensor([33, 73, 37, 81, 63])  
>>> Epoch 2  
Rank 0: Batch 0 = tensor([46, 36, 80, 17, 23])  
Rank 1: Batch 0 = tensor([16, 64, 97, 12, 59])  
Rank 2: Batch 0 = tensor([91, 18, 49, 24, 4])  
Rank 3: Batch 0 = tensor([33, 73, 37, 81, 63])
```

Important: The seed used by `DistributedSampler` for the shuffling step is calculated from the epoch number but this number is not updated automatically during the training.

- **DistributedSampler** + shuffling
- The index shuffling is performed by each GPU from a common seed.

```
for epoch in range(1,30):  
    data_sampler.set_epoch(epoch)  
    for i, batch in enumerate(data_loader):  
        ...
```



```
>>> Epoch 1  
Rank 0: Batch 0 = tensor([46, 36, 80, 17, 23])  
Rank 1: Batch 0 = tensor([16, 64, 97, 12, 59])  
Rank 2: Batch 0 = tensor([91, 18, 49, 24, 4])  
Rank 3: Batch 0 = tensor([33, 73, 37, 81, 63])  
>>> Epoch 2  
Rank 0: Batch 0 = tensor([49, 91, 8, 76, 48])  
Rank 1: Batch 0 = tensor([98, 50, 21, 15, 22])  
Rank 2: Batch 0 = tensor([ 2, 11, 71, 92, 75])  
Rank 3: Batch 0 = tensor([82, 9, 74, 39, 53])
```

To update the epoch number and, therefore, the seed used for shuffling in the `DistributedSampler` class, it is necessary to call the `set_epoch()` function at each epoch.

- Custom Sampler (inspired by DistributedSampler)

```
class MyCustomDistributedSampler(Sampler):
```

```
    def __init__(self, dataset, world_size, rank):
        self.data_len = len(dataset)
        self.world_size = world_size
        self.rank = rank
```

```
    def __len__(self):
        return self.data_len
```

```
    def __iter__(self):
        indices = list(range(self.data_len))
        # shuffle or not shuffle
        indices = indices[self.rank:self.data_len:self.world_size]
        return iter(indices)
```

```
$ srun --ntasks=4 <...> script.py
Rank 0: Batch 0 = tensor([ 1,  5,  9, 13, 17])
Rank 1: Batch 0 = tensor([ 2,  6, 10, 14, 18])
Rank 2: Batch 0 = tensor([ 3,  7, 11, 15, 19])
Rank 3: Batch 0 = tensor([ 4,  8, 12, 16, 20])
```

```
sampler = MyCustomDistributedSampler(dataset, idr_torch.size, idr_torch.rank)
```

37

If you use a custom Sampler, a distribution of indexes is possible as illustrated. This example is based on the `DistributedSampler` class.



- Go into the directory `tp_pi/`
- Follow instructions in the notebook `DL0-JZ_Compute_pi.ipynb`
- Parallelize the code `compute_pi.py`
- Compute **PI** on 4 GPUs

TP2_1: Implement data parallelism in dlojz.py



- Follow instructions in the notebook `DL0-JJ_Jour2.ipynb`
- Implement data parallelism in the script `dlojz.py`
- Measure the gain in time when using 4 GPUs