Deep Learning Optimized on Jean Zay

Distribution – Data parallelism

cnrs

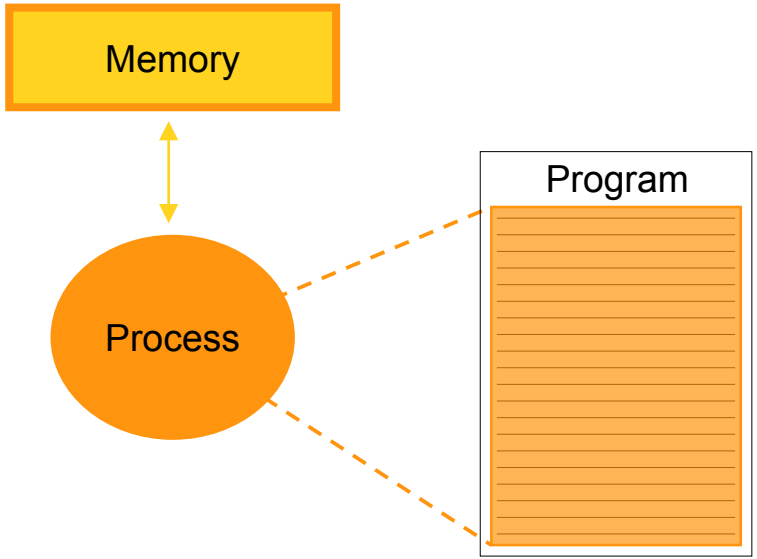
IDRIS

Distributed training

General knowledge about parallel computing ◂

Data parallelism to distribute your training ◂

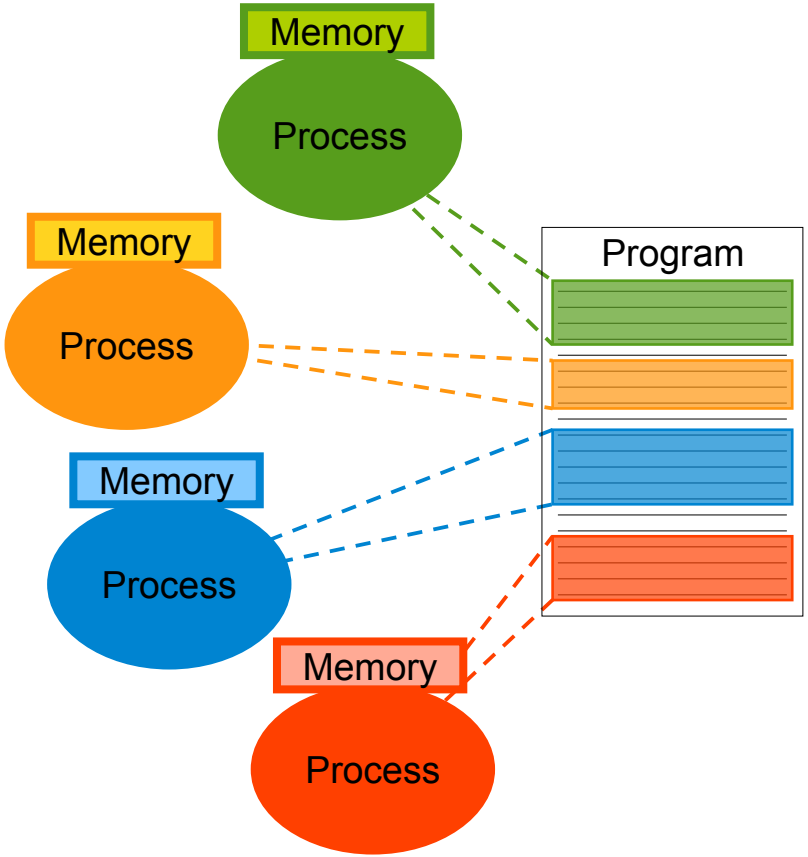
Distribution: General knowledge about parallel computing



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).

Distribution: General knowledge about parallel computing



Parallel execution with distributed memory

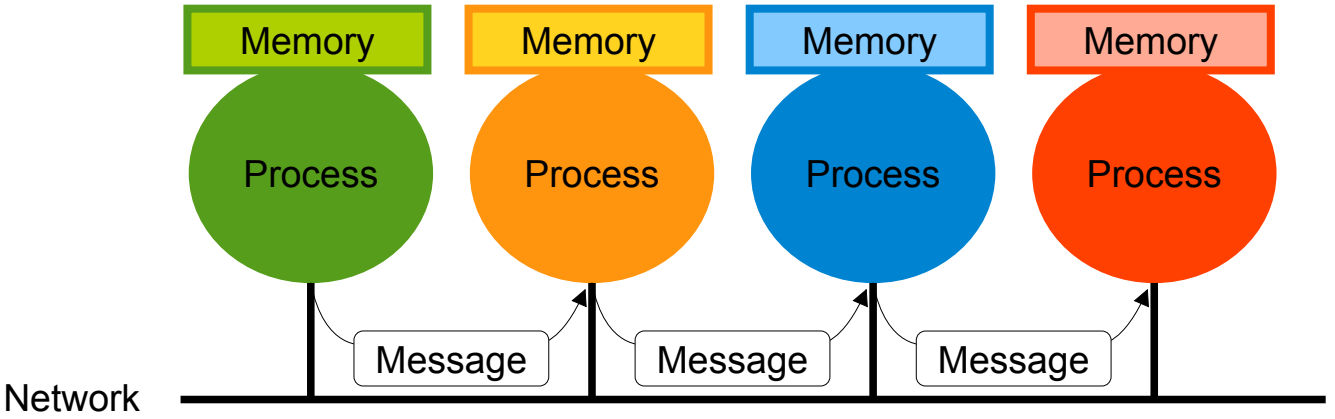
- Several processes execute the code at the same time (multiprocessing).
- The variables defined in the program are private, they are stored in the local memory allocated to each process.
- It is possible that the processes execute separate parts of the code.

Distribution: General knowledge about parallel computing

Parallel execution with distributed memory

- To share information, the processes can send each other **messages** through the interconnection network.

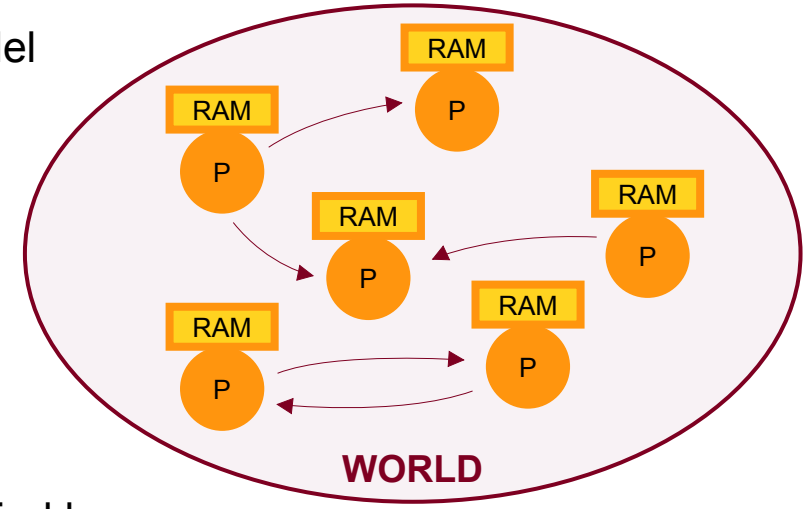
- These **communications** are managed by libraries such as **MPI** or **NCCL**.



Distribution: General knowledge about parallel computing

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**.



Distribution: General knowledge about parallel computing

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 ✓

Distribution: General knowledge about parallel computing

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
```

Distribution: General knowledge about parallel computing

Example 2: We identify the processes thanks to their ranks

- Parallelized code using Horovod:

```python
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
```

Distribution: General knowledge about parallel computing

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
```

Distribution: General knowledge about parallel computing

Example 3bis: 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
```

Distribution: General knowledge about parallel computing

Example 4: Parallelization of a compute loop

- Initial state of the memory
- Program
- Final 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
```

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

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

Distribution: General knowledge about parallel computing

Example 4: Parallelization of a compute loop

- Initial state of the memory
- Program
- Final 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
```

```
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]
```

```
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
```

Distribution: General knowledge about parallel computing

Example 4: Parallelization of a compute loop

- Initial state of the memory
- Program
- Final state of the memory

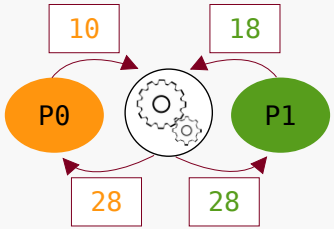
↓Process 0↓

```
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
```

```
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]
sum = hvd.allreduce(sum,op=hvd.Sum)
```

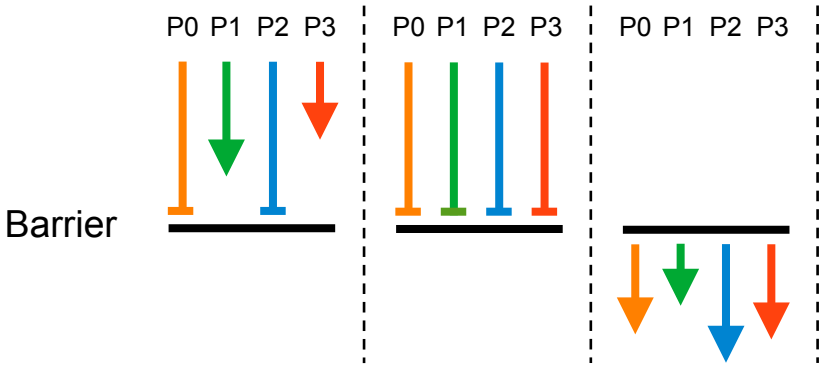


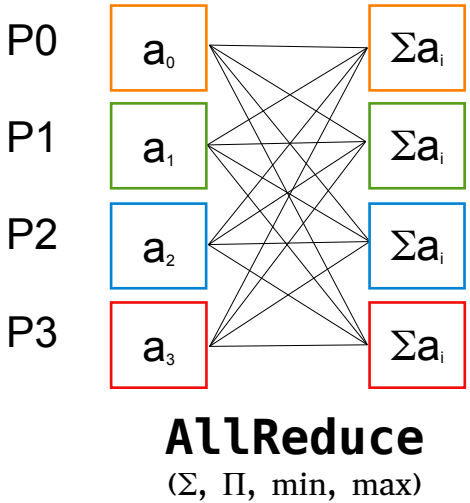

```
N = 4
a = [0, 1, 2, 3]
b = [4, 5, 6, 7]
c = [4, 6, 0, 0]
sum = 28
```

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

Distribution: General knowledge about parallel computing

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

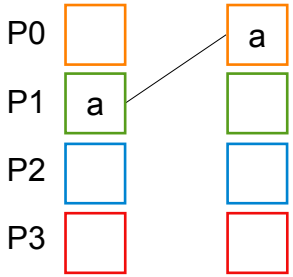




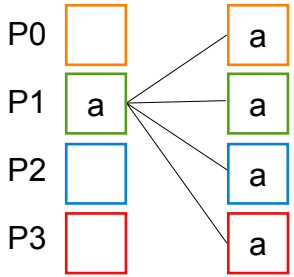
costly communication

Distribution: General knowledge about parallel computing

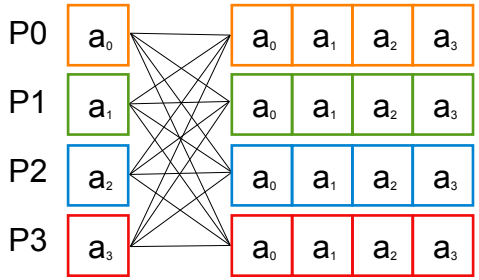
- NCCL communications



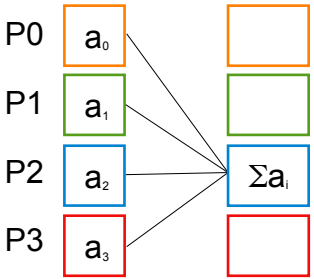
Send/Receive



Broadcast

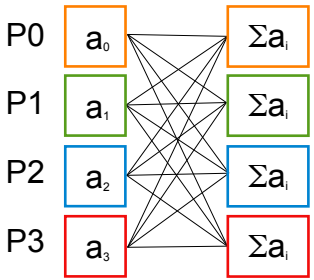


AllGather



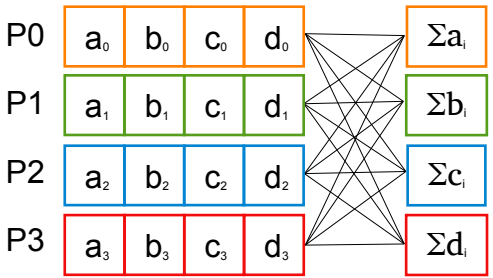
Reduce

(Σ, Π, min, max)



AllReduce

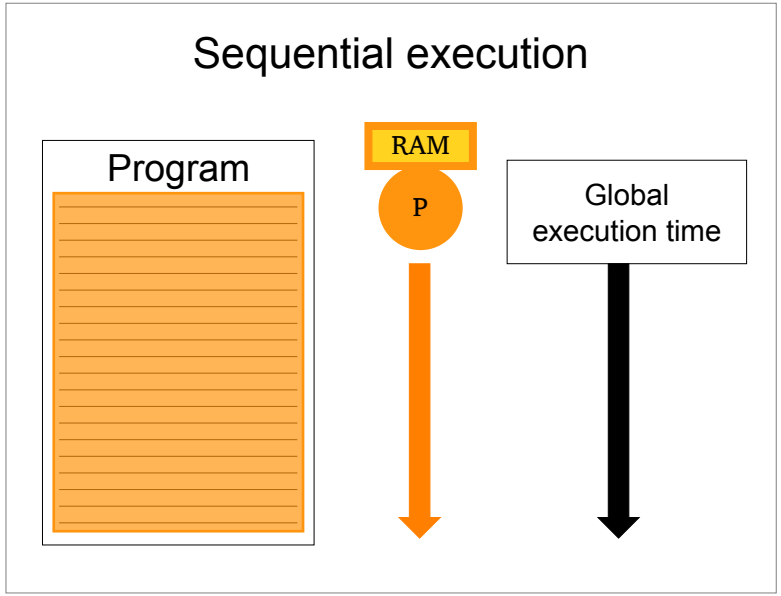
(Σ, Π, min, max)

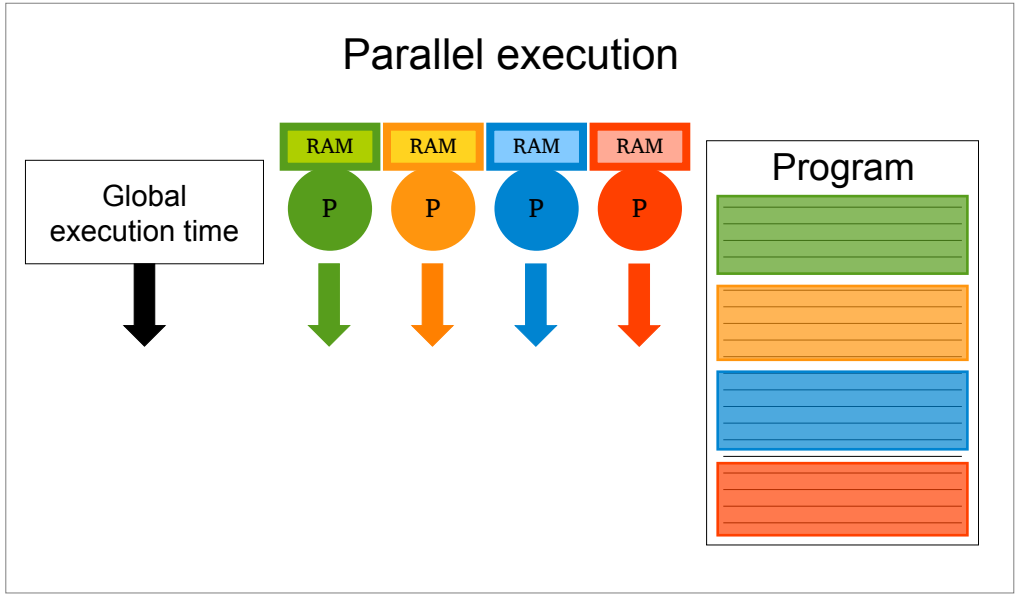


ReduceScatter

(Σ, Π, min, max)

Distribution: General knowledge about parallel computing





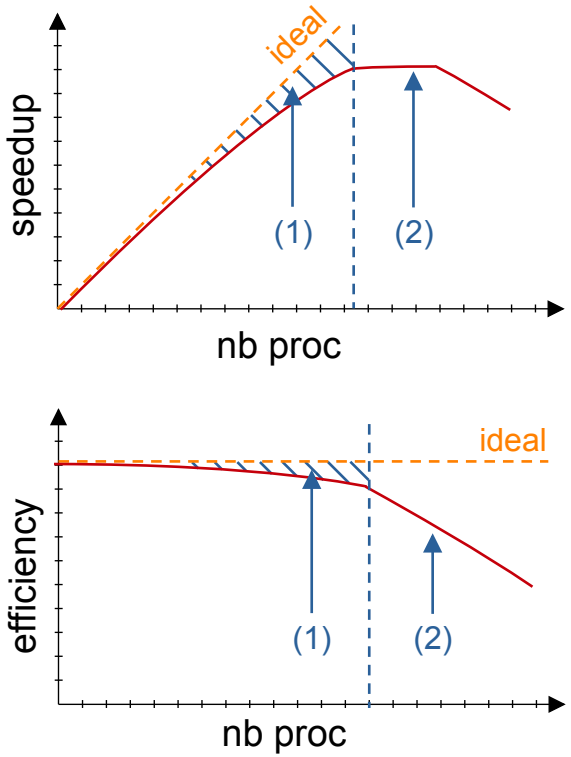
- Global execution time: $$T(Nprocs) = \frac{T_{parall}(1proc)}{N} + T_{seq} + T_{comm}$$

Distribution: General knowledge about parallel computing

Scalability study: $\frac{T(1proc)}{T(Nprocs)}$

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

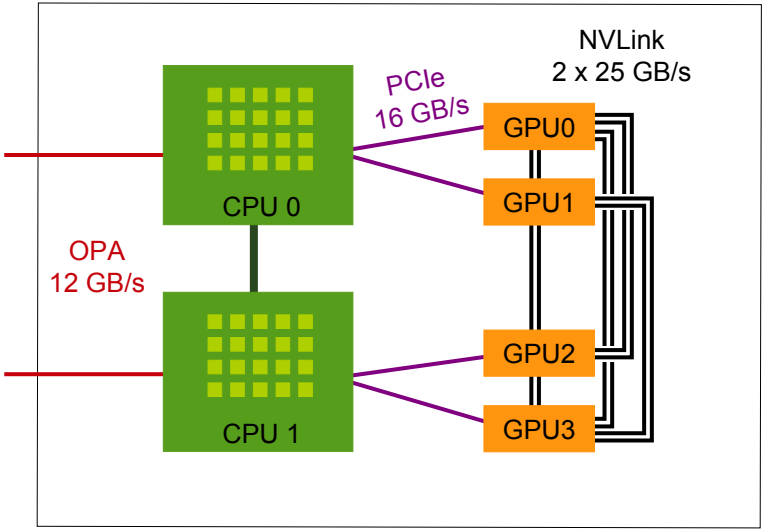
- **Weak scaling**
  (problem size is proportional to the number of processes)



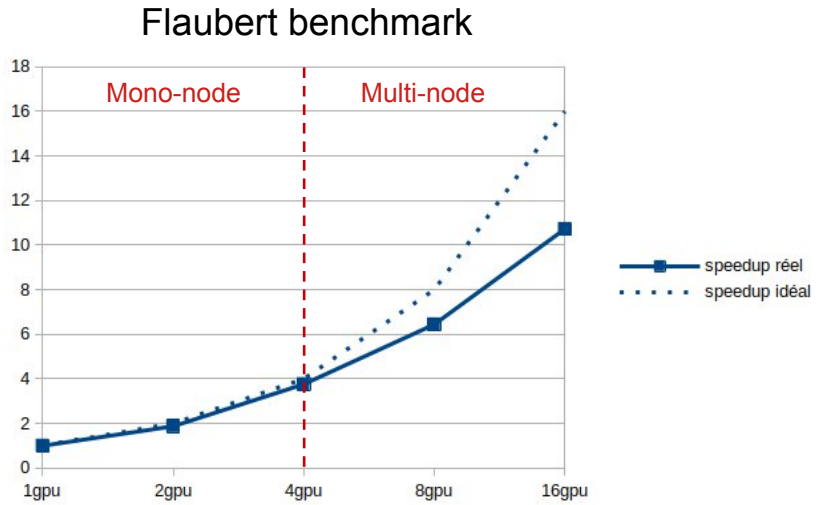
(1) impact of the sequential parts of the code
(2) cost of the inter-process communications

Distribution: General knowledge about parallel computing

- Bandwidths of the interconnection networks on Jean Zay:



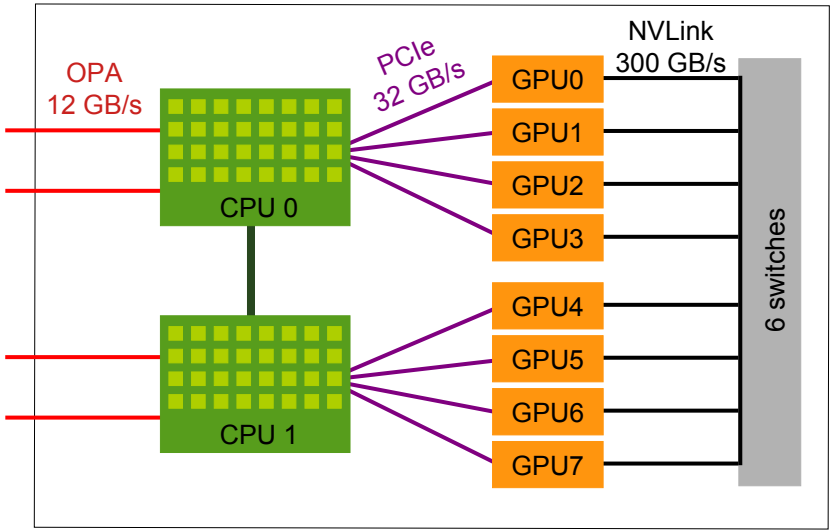
Node 4 × V100 16GB



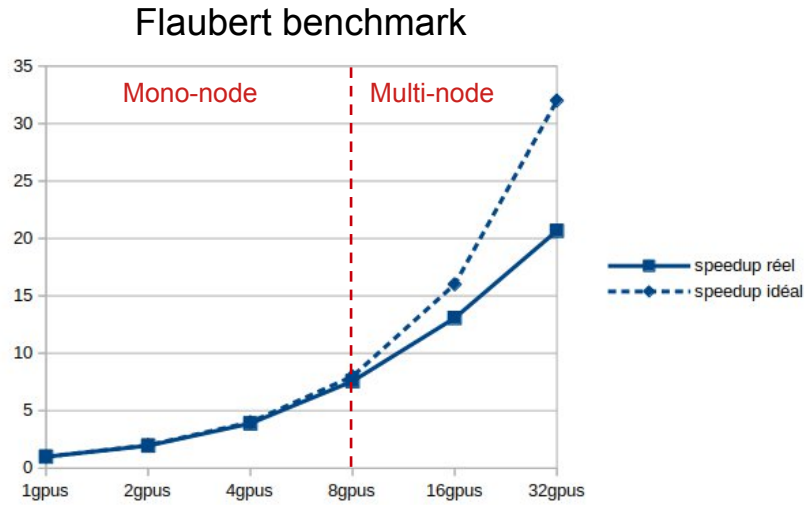
Strong scaling

Distribution: General knowledge about parallel computing

- Bandwidths of the interconnection networks on Jean Zay:



Node 8 × A100 80GB



Strong scaling

Distributed training

General knowledge about parallel computing ◄

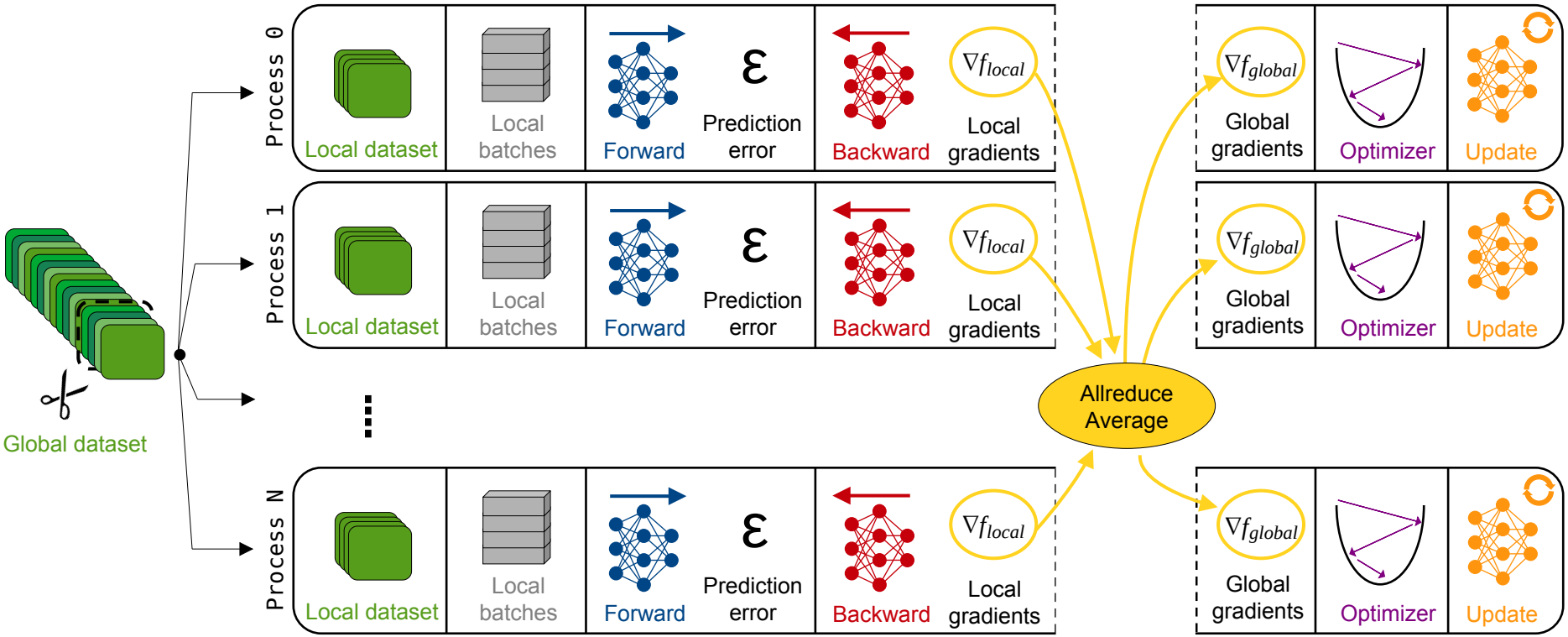
Data parallelism to distribute your training ◄

Distribution: Data parallelism

Data parallelism

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

Distribution: Data parallelism

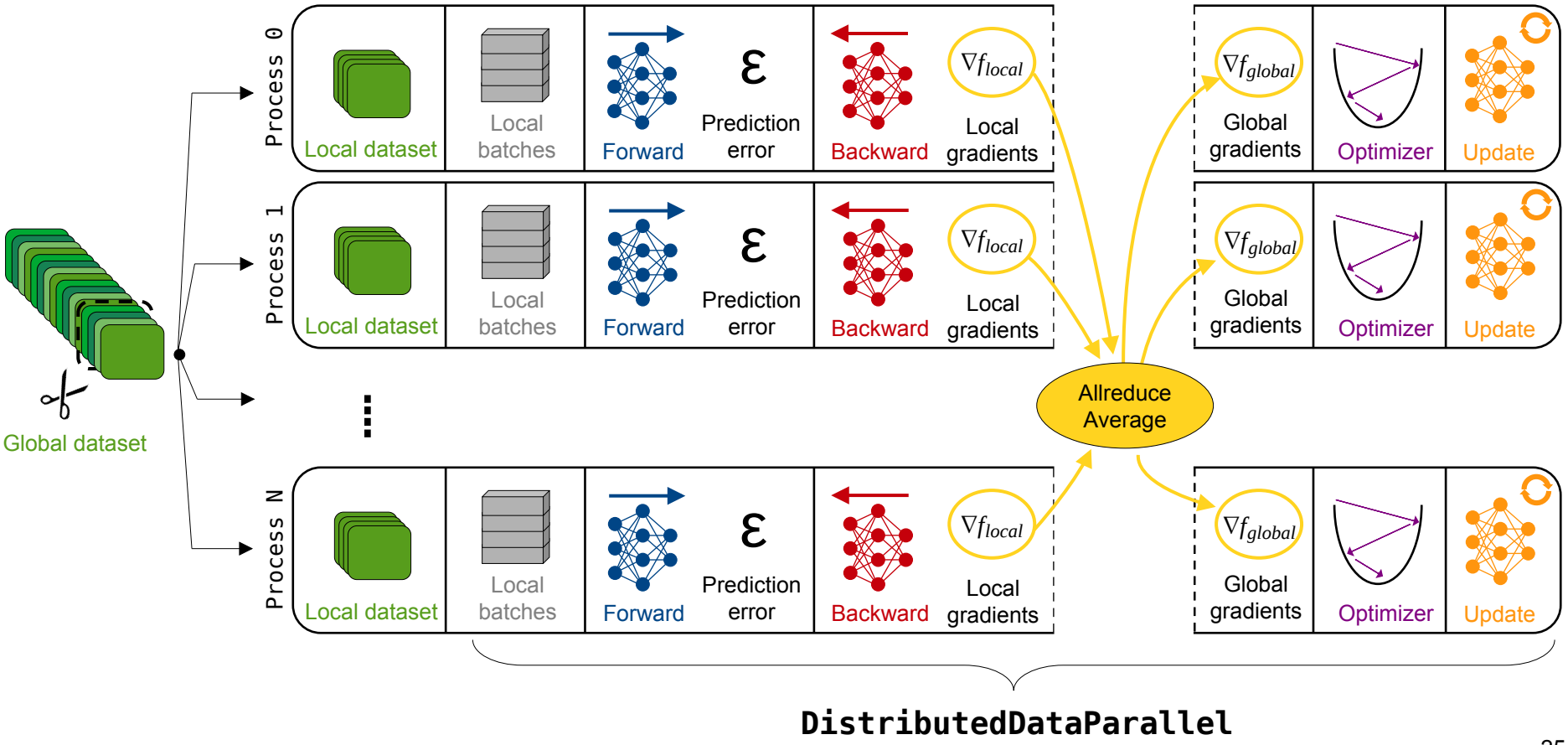


Distribution: Data parallelism

Implementation of the data parallelism

- PyTorch → **`DistributedDataParallel`** (integrated solution)
- TensorFlow → `MultiWorkerMirroredStrategy` (integrated solution)
- Horovod (external librairy)

Distribution: Data parallelism

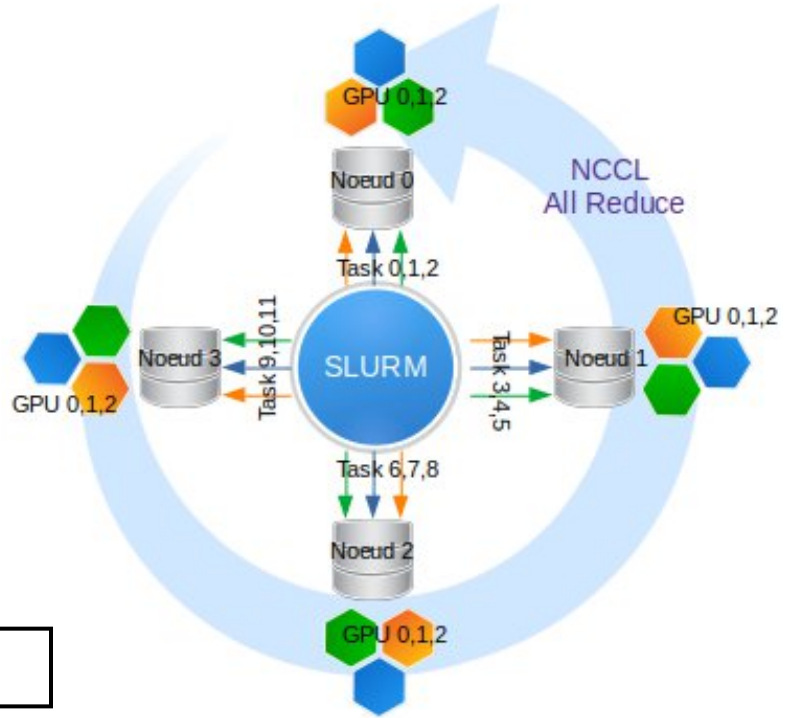


Distribution: Data parallelism

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
```

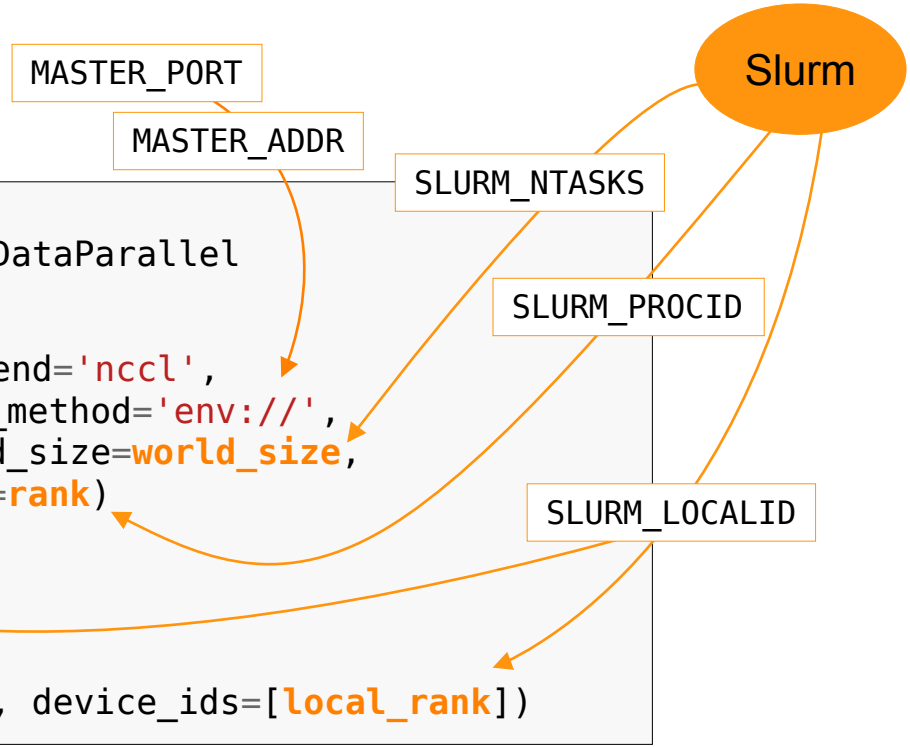


Each GPU must be binded with one process.

Distribution: Data parallelism

- **`DistributedDataParallel`** (training distribution)

```python
import torch.distributed
from torch.nn.parallel import DistributedDataParallel

# initialize the parallel environment
torch.distributed.init_process_group(backend='nccl',
                                     init_method='env://',
                                     world_size=world_size,
                                     rank=rank)

# bind one GPU per process
torch.cuda.set_device(local_rank)

# duplicate the model
ddp_model = DistributedDataParallel(model, device_ids=[local_rank])
```



Distribution: Data parallelism

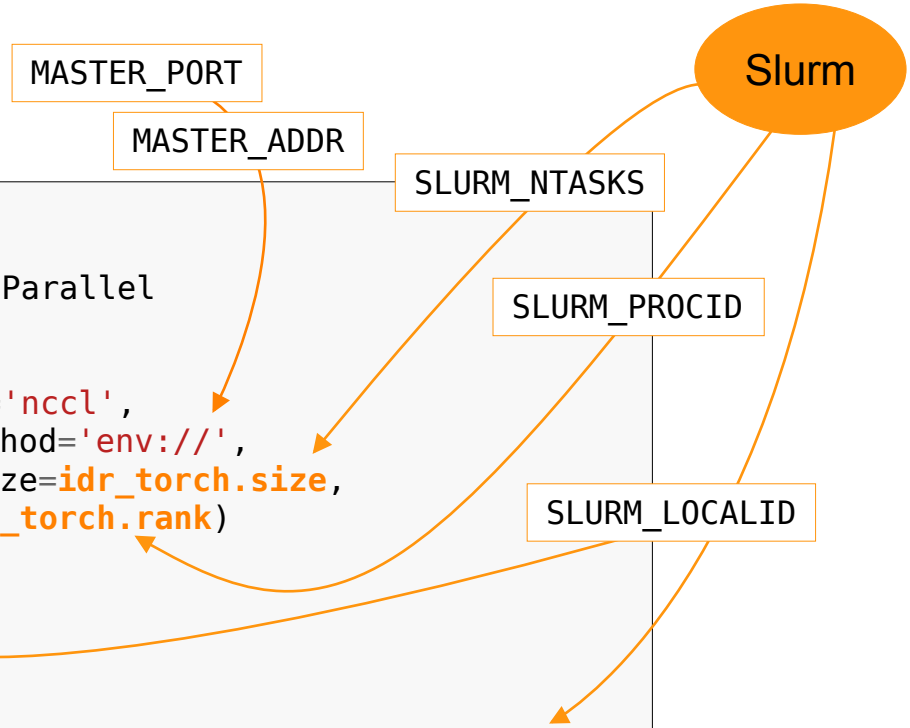
- **`DistributedDataParallel`** (training distribution)
  - *idr_torch.py* script from IDRIS

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

Distribution: Data parallelism

- **`DistributedDataParallel`** (training distribution)

```python
import idr_torch
import torch.distributed
from torch.nn.parallel import DistributedDataParallel

# initialize the parallel environment
torch.distributed.init_process_group(backend='nccl',
                                     init_method='env://',
                                     world_size=idr_torch.size,
                                     rank=idr_torch.rank)

# bind one GPU per process
torch.cuda.set_device(idr_torch.local_rank)

# duplicate the model
ddp_model = DistributedDataParallel(model, device_ids=[idr_torch.local_rank])
```



Distribution: Data parallelism

- What about `torchrun`? → Possible but cumbersome.

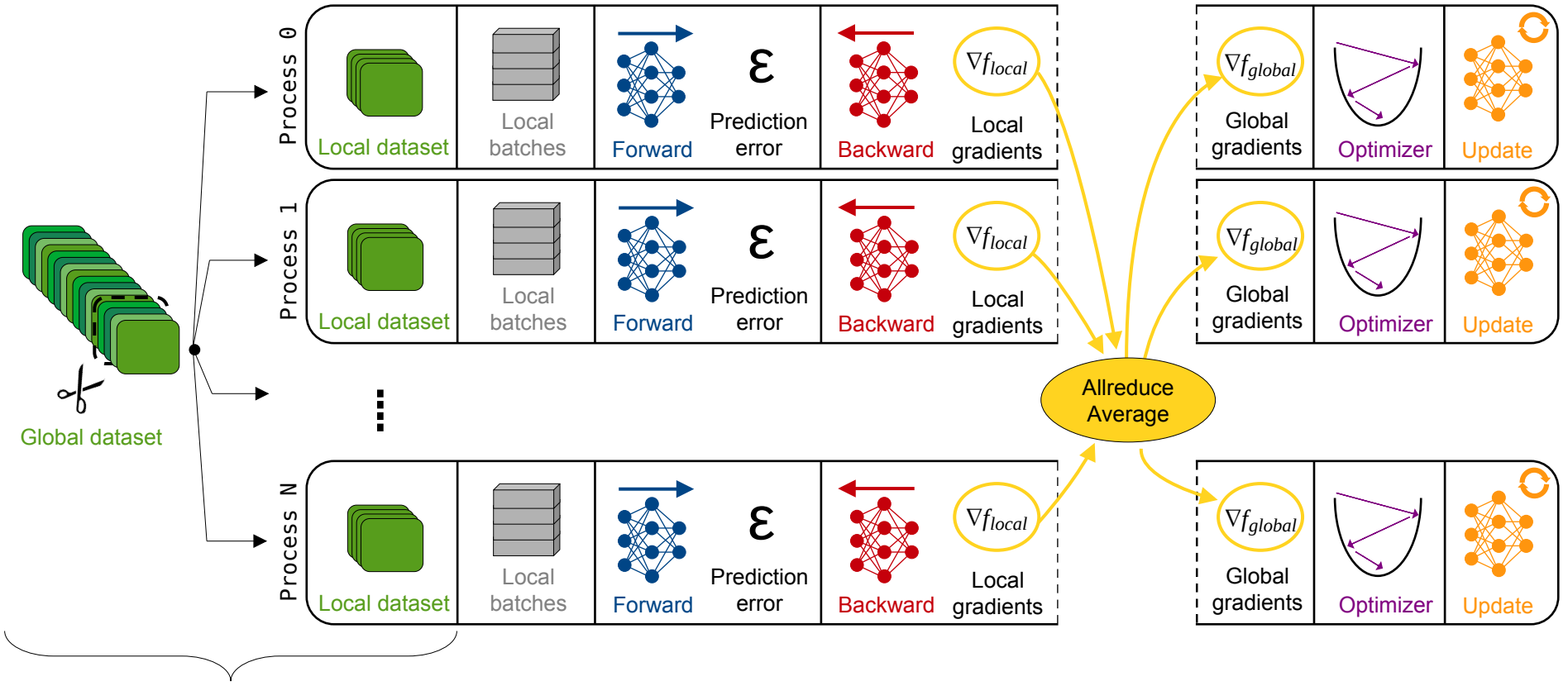
slurm.sh

```
[…]
#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"
```

train.py

```
[…]
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'])
```

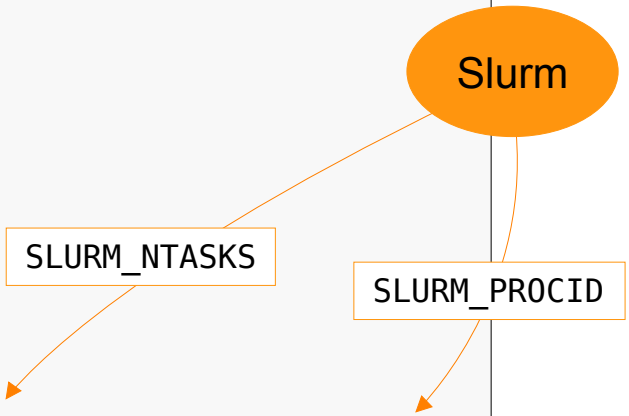
Distribution: Data parallelism



Distribution: Data parallelism

- **`DistributedSampler`** (distributing the input data)

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

# initialize the parallel environmenet
[...]

# 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)
```



The shuffling step is assigned to the Sampler.

Distribution: Data parallelism

- **`DistributedSampler`** (distributing the input data)

  - Parallel execution on 4 processes using the `DistributedSampler`:

```
dataset = [0, 1, ..., 99]
batch_size_per_gpu = 5
ntasks = 4
batch_size = 20
```

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

Distribution: Data parallelism

- **`DistributedSampler`** (distributing the input data) **+ shuffling**
  - **The index shuffling is performed by each GPU from a common seed.**

  - Parallel execution on 4 processes using the `DistributedSampler`:

```
dataset = [0, 1, ..., 99]
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])
```

Distribution: Data parallelism

- **`DistributedSampler`** (distributing the input data) **+ shuffling**
  - **The index shuffling is performed by each GPU from a common seed.**

```
for epoch in range(1,30):

        for i, batch in enumerate(dataloader):
                ...
```

```
>>> 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])
```

Distribution: Data parallelism

- **`DistributedSampler`** (distributing the input data) **+ 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(dataloader):
                ...
```

```
>>> 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])
```

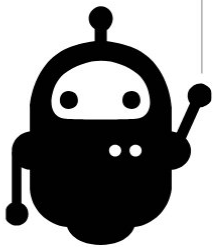
Distribution: Data parallelism

- Custom Sampler (inspired by `DistributedSampler`)

```python
class MyCustomDistributedSampler(Sampler):

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

        def __len__(self):
                return self.datalen

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


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

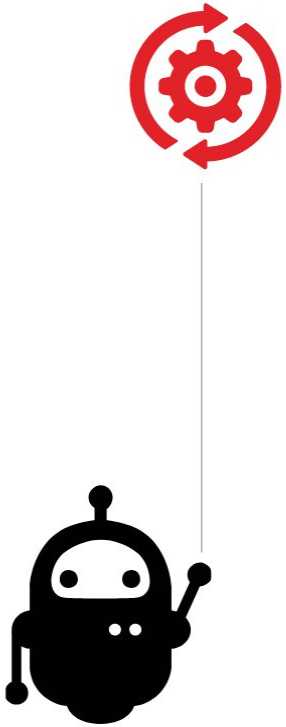
```
$ srun --ntasks=4 <...> script.py
Rank 0: Batch 0 = tensor([ 0,  4,  8, 12, 16])
Rank 1: Batch 0 = tensor([ 1,  5,  9, 13, 17])
Rank 2: Batch 0 = tensor([ 2,  6, 10, 14, 18])
Rank 3: Batch 0 = tensor([ 3,  7, 11, 15, 19])
```

TP_PI: Parallel computation of PI using torch.distributed



- Go into the directory `tp_pi/`
- Follow instructions in the notebook **`DLO-JZ_Compute_pi.ipynb`**
- Parallelize the code `compute_pi.py` using `torch.distributed`
- Compute π on 4 GPUs

TP2_1: Implement data parallelism in dlojz.py



- Follow instructions in the notebook **`DLO-JZ_Jour2.ipynb`**
- Implement data parallelism in the script `dlojz.py`
- Measure the gain in time when using 4 GPUs