Distributed applications

02476 Machine Learning Operations
Nicki Skafte Detlefsen

What is distributed applications?

Computing on multiple threads/devices/nodes in parallel

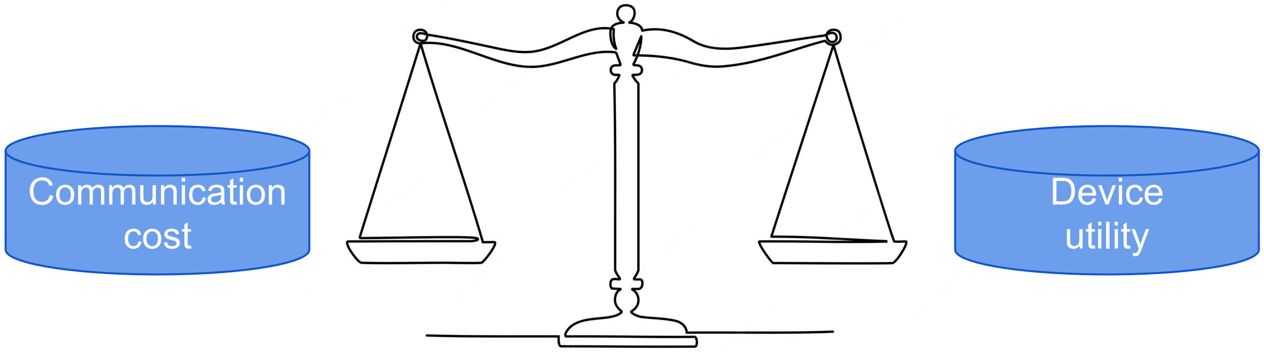
What can run in parallel

- Data loading
- Training
- Inference

This lecture focuses on training as it is the most computationally expensive

Key take away

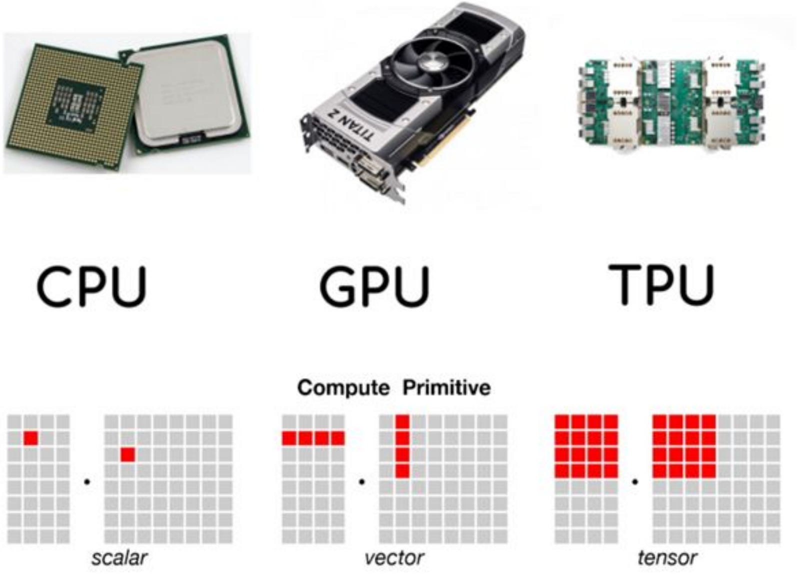
Distributed computing is not always beneficial, its an trade-off:



Devices

Three common types of devices

- CPU
  - General compute unit
  - 2-128 parallel operations
- GPU
  - Rendering unit
  - 1.000-10.000 parallel operations
- TPU
  - Specialized unit
  - 32.000 - 128.000 parallel operations



Device Memory

Equally important is the amount of memory you haven available

With more memory you get

- Faster data transfer
- Possibility of higher data modalities
- Larger models

	CPU	GPU	TPU
Standard	32-64 GiB	12 GiB	64 GiB
Maximum	2 TiB	80 GiB	32 TiB

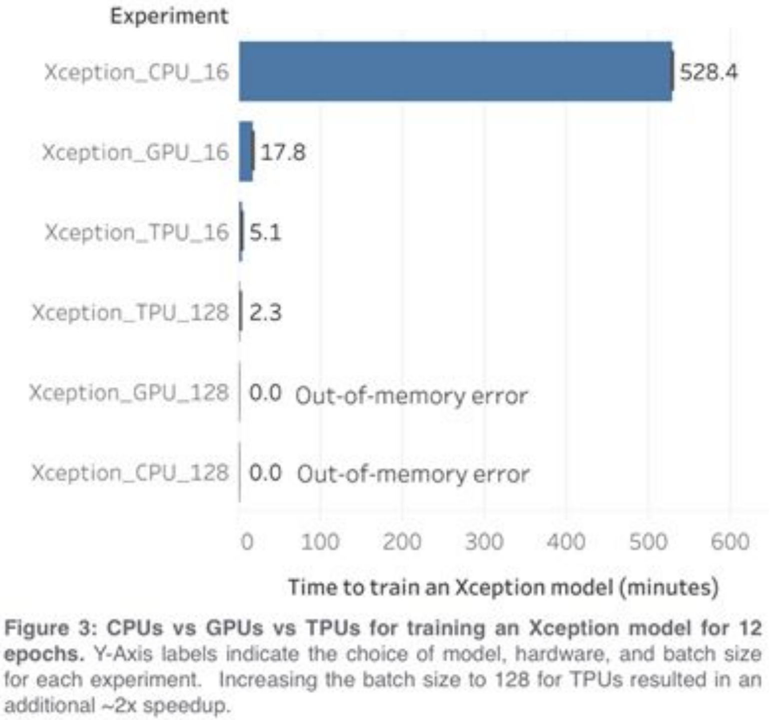


Image credit: https://towardsdatascience.com/when-to-use-cpus-vs-gpus-vs-tpus-in-a-kaggle-competition-9af708a8c3eb

Many layers of distributed computations

Data

Load data in parallel
Inference on data in parallel

Model

Stack models in parallel

Experiment

Multiple experiments in parallel

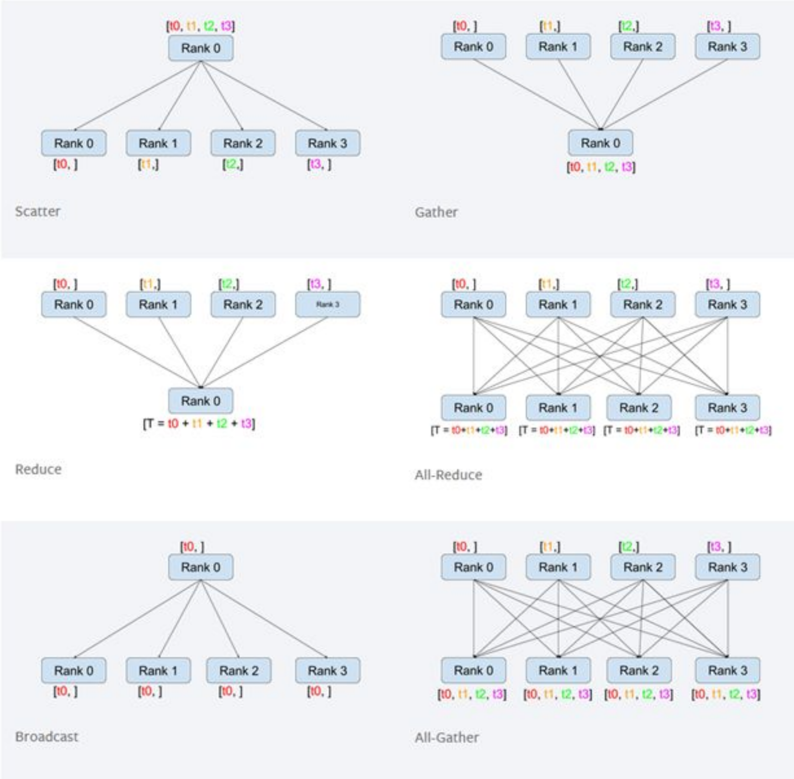
Use a combination

Communication operations

- Scatter
- Gather
- Reduce
- Broadcast
- All-gather
- All-reduce

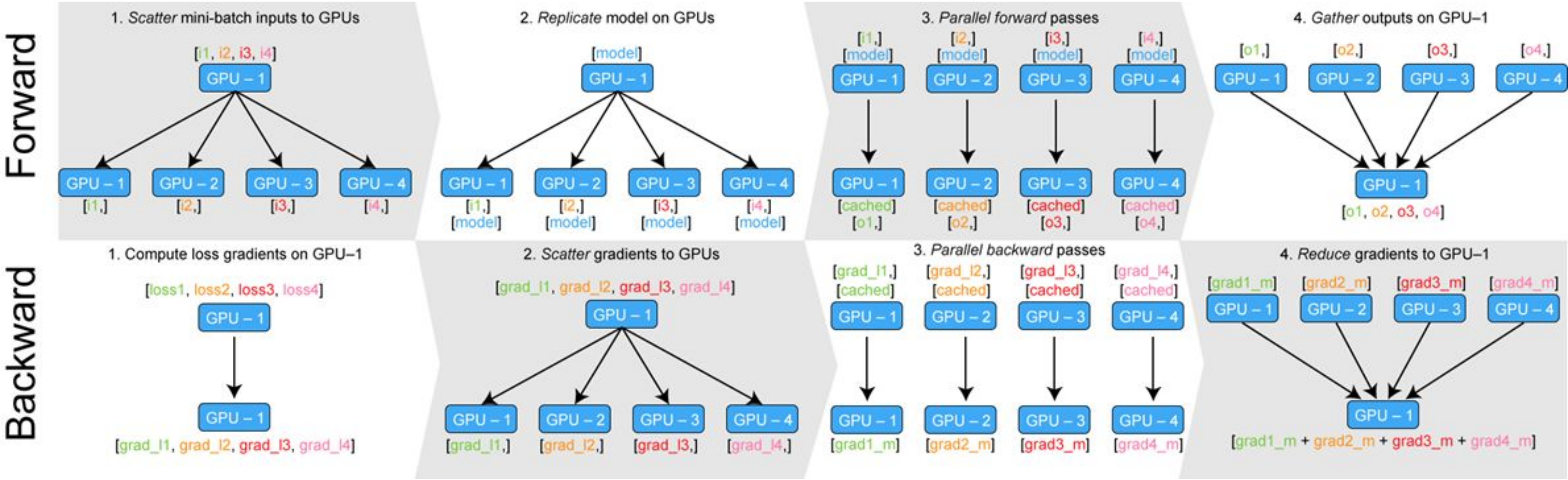
Rank 0: main

Rank >0: worker



Data parallel

Simple as `parallel_model = torch.nn.DataParallel(model)`



Only GPU-1 is updated, replicas are destroyed

Distributed data parallel

Distributed Data Parallel

No master GPUs

Implemented in PyTorch DistributedDataParallel module

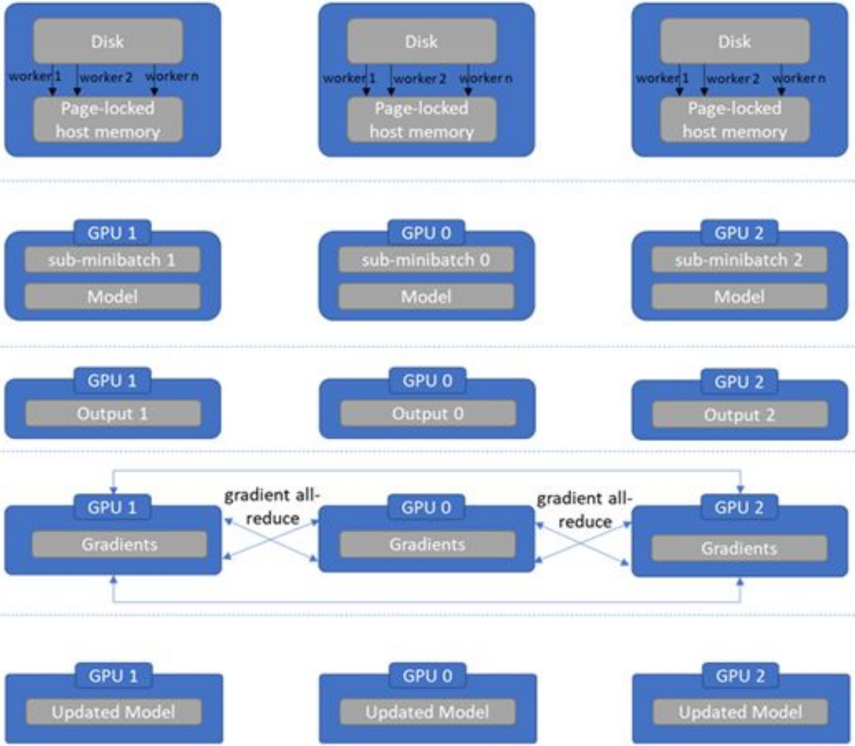
1. Load data from disk into page-locked memory on the host. Use multiple worker processes to parallelize data load. Distributed minibatch sampler ensures that each process loads non-overlapping data

2. Transfer minibatch data from page-locked memory to each GPU concurrently. No data broadcast is needed. Each GPU has an identical copy of the model and no model broadcast is needed either

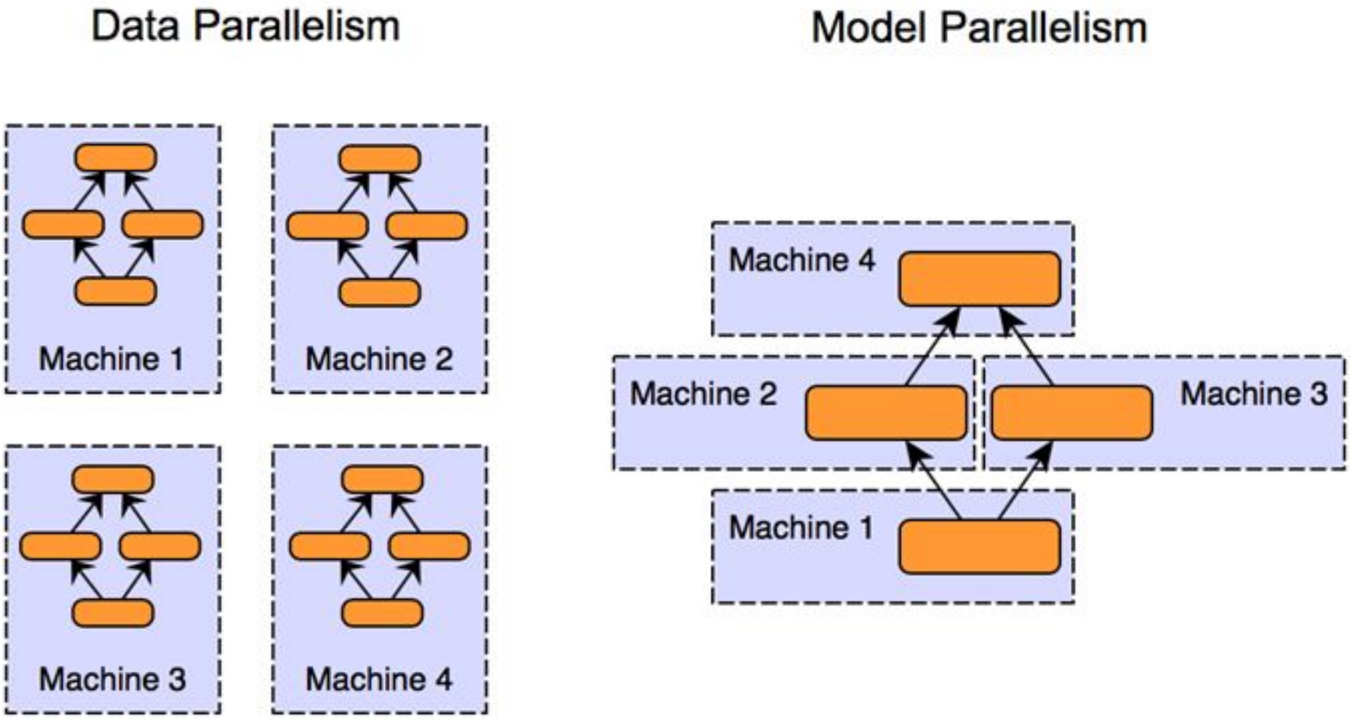
3. Run forward pass on each GPU, compute output

4. Compute loss, run backward pass to compute gradients. Perform gradient all-reduce in parallel with gradient computation

5. Update Model parameters. Because each GPU started with an identical copy of the model and gradients were all-reduced, weights updates on all GPUs are identical. Thus no model sync is required



Model parallelisme



Comparing methods

Method	Pros	Cons
Data parallel	Simple to use	Slow due to replicas being destroyed
Distributed data parallel	Fast	High memory requirement
Model parallel	Large models that other methods	Slow due to high communication cost

How to do this in practise

Dataparallel

- `parallel_model = torch.nn.DataParallel(model)`

Distributed data parallel (DDP)

- Set a environment `MASTER_ADDR` and `MASTER_PORT`
- Initialize a process group
- `parallel_model = nn.parallel.DistributedDataParallel(model, device_ids=[gpu])`
- Use mp.spawn to spawn multiple processes
- …

Model parallizeme

- A shit ton of `tensor.to(f“cuda:{i}”)` calls

How to do this in practise

Dataparallel

- `parallel_model = torch.nn.DataParallel(model)`

Distributed data parallel (DDP)

- Set
- Initi
- `para`
  `devi`
- Use
- …

Trust me, you do not want to do this yourself

Model parallelism

- A shit ton of `tensor.to(f"cuda:{i}")` calls

What you should focus on

Separating engineering code and research code

Research code

```
l1 = nn.Linear(...)
l2 = nn.Linear(...)
decoder = Decoder()

x1 = l1(x)
x2 = l2(x2)
out = decoder(features, x)

loss = perceptual_loss(x1, x2, x) + CE(out, x)
```

Engineering code

```
model.cuda(0)
x = x.cuda(0)

distributed = DistributedParallel(model)

with gpu_zero:
    download_data()

dist.barrier()
```

You should be spending time on Research code not Engineering code

Lets abstract away engineering code

Spend time on research code and not engineering code

=> This is the reason high-level frameworks exist!

- Reduce boilerplate
- Focus on what is important
- Reproducibility
- Share Ability
- Consistency
- Scalability
- ...

Training frameworks

Many frameworks exist for reducing boilerplate

Many frameworks for accelerating training

Training frameworks

Many frame late

Many fram

I am affiliated with these guys

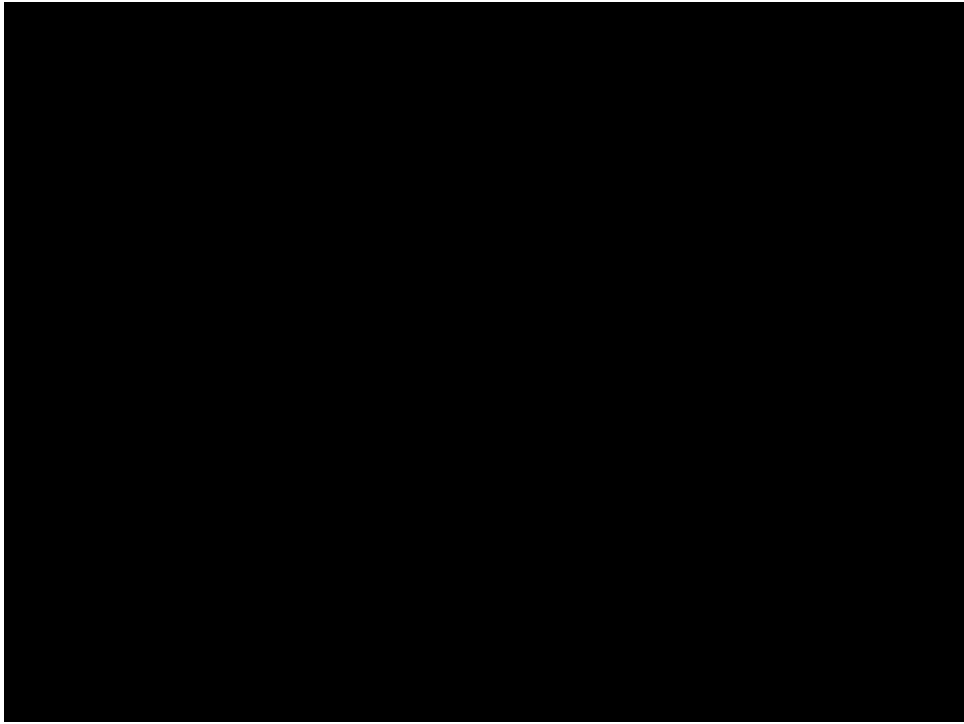
Pytorch lightning

Its just reorganized Pytorch code!

Two core objects

- Lightning Module
  - Training, validation, test logic
  - Optimizer
- Trainer
  - The "rest"

`trainer.fit(model)` does the heavy lifting



Device agnostic

```
# run on cpu, gpu, tpu, ipu
# with no code changes needed
trainer = Trainer(devices=8, accelerator='cpu')
trainer = Trainer(devices=8, accelerator='gpu')
trainer = Trainer(devices=8, accelerator='tpu')
trainer = Trainer(devices=8, accelerator='ipu')

# or just let lightning auto detect
trainer = Trainer(devices=8, accelerator='auto')
```

```
# for gpu, you can also do multiple nodes
# 32 nodes * 8 gpus per node = 256 gpus!
trainer = Trainer(devices=8, accelerator='gpu', num_nodes=32)
```

Scaling matters

Scale does matter in deep learning!
Scaling to such ridiculous number of parameters requires specialized training strategies


```python
# Sharded training using fairscale
trainer = Trainer(devices=4, strategy='ddp_sharded')

# sharded training using deepspeed
trainer = Trainer(devices=4, strategy="deepspeed_stage_1", precision=16)
trainer = Trainer(devices=4, strategy="deepspeed_stage_2", precision=16)
trainer = Trainer(devices=4, strategy="deepspeed_stage_3", precision=16)
```

Image credit: https://www.microsoft.com/en-us/research/blog/using-deepspeed-and-megatron-to-train-megatron-turing-nlg-530b-the-worlds-largest-and-most-powerful-generative-language-model/

“Free” features

The trade-off

Upfront cost of refactor

Vs

Long term gain of scaling + featureset

```python
# The trainer has 50+ flags to use
trainer = Trainer(
     # 1. accumulate over multiple batches
    accumulate_grad_batches=10,
     # 2. automatically determines best lr
    auto_lr_find=True,
     # 3. gradient clipping
    gradient_clip_val=1.0,
     # 4. 16 bit precision for memory
    precision=16,
     # 5. automatically profile your code
    profiler="simple",
)
```

The ecosystem

lightning Public

Build and train PyTorch models and connect them to the ML lifecycle using Lightning App templates, without handling DIY infrastructure, cost management, scaling, and other headaches.

Python 20.9k 2.7k

lightning-flash Public

Your PyTorch AI Factory - Flash enables you to easily configure and run complex AI recipes for over 15 tasks across 7 data domains

Python 1.6k 187

lightning-bolts Public

Toolbox of models, callbacks, and datasets for AI/ML researchers.

Python 1.4k 294

metrics Public

Machine learning metrics for distributed, scalable PyTorch applications.

Python 1.2k 246

Today's session

- Distributed dataloading
  - Using Pytorchs dataloader to load data in parallel
- Distributed training
  - Data parallel
  - Distributed data parallel
- Scalable inference
  - Architecture choice
  - Quantization
  - Pruning
  - Knowledge distillation

Meme of the day