

A photograph of a server room. In the foreground, a large server rack is covered in a white, complex geometric pattern. The rack is filled with various electronic components and cables. In the background, other server racks and yellow overhead cranes are visible. A semi-transparent white box with rounded corners is overlaid on the left side of the image, containing text.

# **HIGH PERFORMANCE MACHINE LEARNING**

Caspar van Leeuwen  
High Performance ML consultant  
SURF

**SURF**

<https://github.com/sara-nl/PraceHPML2022>

# Program, 2nd day

- 9:00 – 9:45 Introduction to Parallel Computing (for Deep Learning) (Caspar van Leeuwen)
- 9:45 – 10:30 Understanding your hardware and hardware bottlenecks (Caspar van Leeuwen)
- 10:30 – 11:00 Coffee break
- 11:00 – 12:00 Structure of Deep Learning Frameworks: computational graph, autodiff, and optimizers (Robert Jan Schlimbach)
- 12:00 – 13:00 Lunch Break
- 13:00 – 14:00 Hands-on: Profiling PyTorch with TensorBoard (Caspar van Leeuwen)
- 14:00 – 15:00 Hands-on: Data Parallelism with PyTorch Distributed (CIFAR10) (Bryan Cardenas Guevara)
- 15:00 – 15:30 Coffee break
- 15:30 – 16:00 Model sharding (Robert Jan Schlimbach)
- 16:00 – ... Open Discussion

# Parallel computing for Deep Learning

Goals: to understand...

- benefits of parallelization
- parallelization strategies
- parallel stochastic gradient descent (SGD)
- synchronous and asynchronous parallel SGD
- communication backends
- frameworks for distributed deep learning
- documentation of distributed DL frameworks

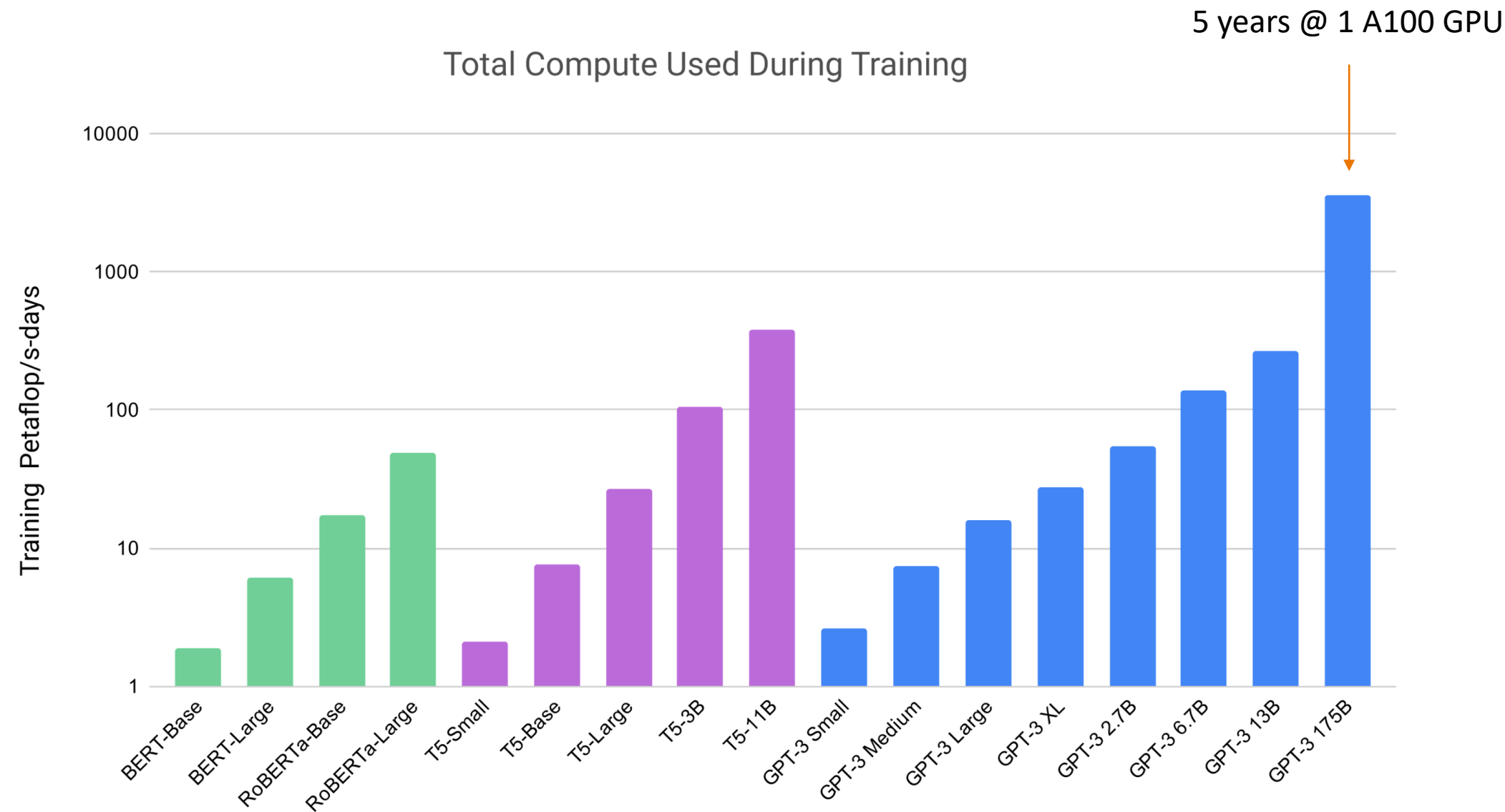
# Parallel computing for Deep Learning

Goal: understand the documentation of distributed DL frameworks.

From TensorFlow docs on “distribution strategy”:

- “tf.distribute.Strategy intends to cover a number of use cases along different axes...  
Synchronous vs asynchronous training: These are two common ways of distributing training with data parallelism. In sync training, all workers train over different slices of input data in sync, and aggregating gradients at each step. In async training, all workers are independently training over the input data and updating variables asynchronously. Typically sync training is supported via all-reduce and async through parameter server architecture.”
- “MultiWorkerMirroredStrategy currently allows you to choose between two different implementations of collective ops. CollectiveCommunication.RING implements ring-based collectives using gRPC as the communication layer. CollectiveCommunication.NCCL uses Nvidia's NCCL to implement collectives.”

# Parallelization: why?



# Parallelization: why?

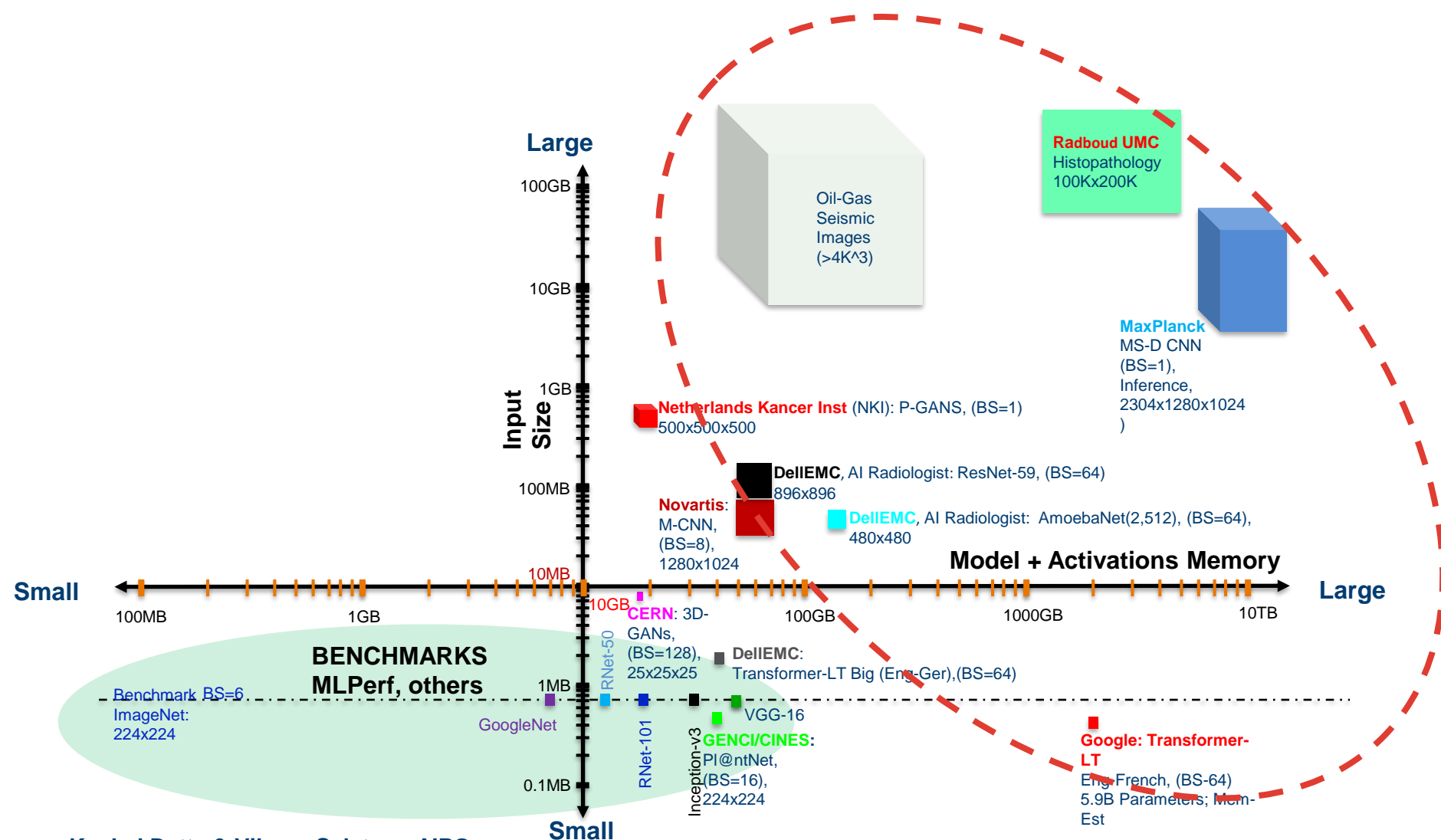
Faster trainings ...

- Enables learning on larger datasets
- Enables improved accuracy through better hyperparameter tuning
- Enables larger, more complex models
- ...

Bigger models (high memory requirement) ...

- Enables larger, more complex models

# Parallelization: when?



Source: Kushal Datta & Vikram Saletore, AIPG, Intel



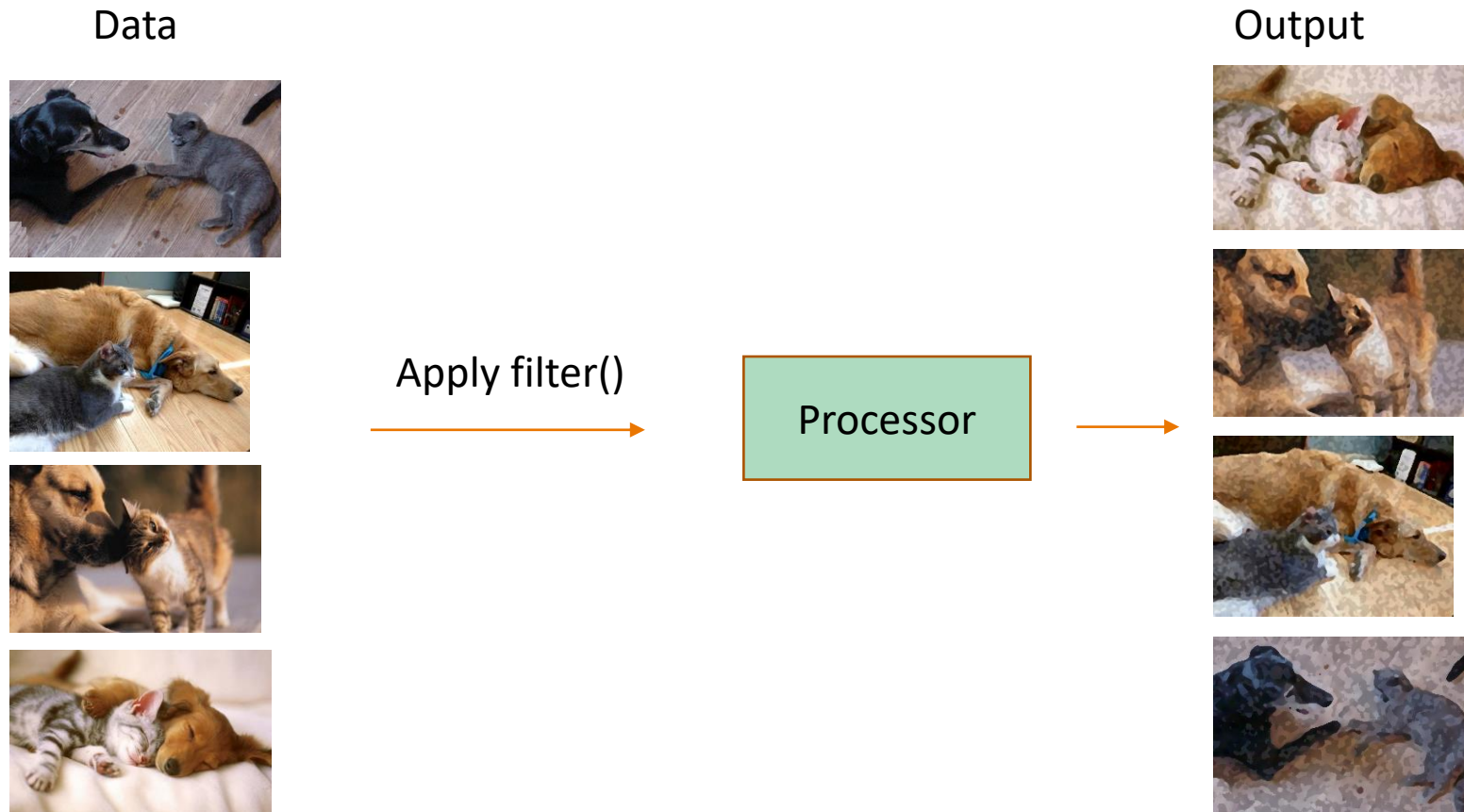
# Parallelization: the basics

What is parallel computing?

- Multiple processors or computers working on a single computational problem

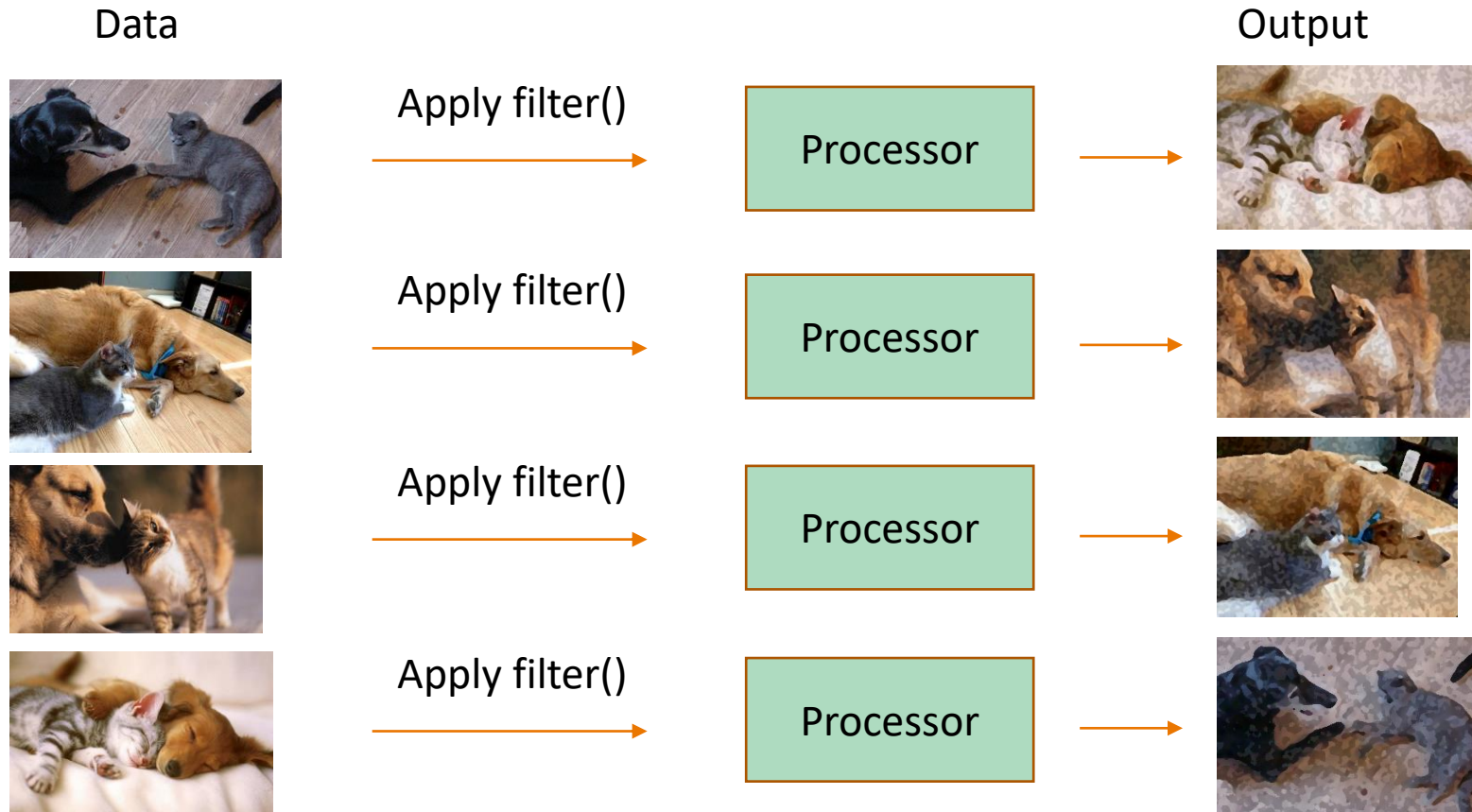
# Parallelization: the basics

Serial computing



# Parallelization: the basics

## Parallel computing



# Parallelization: the basics

Benefits:

- Solve computationally intensive problems (speedup)
- Solve problems that don't fit a single memory (multiple computers)


Requirements:

- Problem should be divisible in smaller tasks

# Parallelization for deep learning

How does one 'divide' DL tasks?

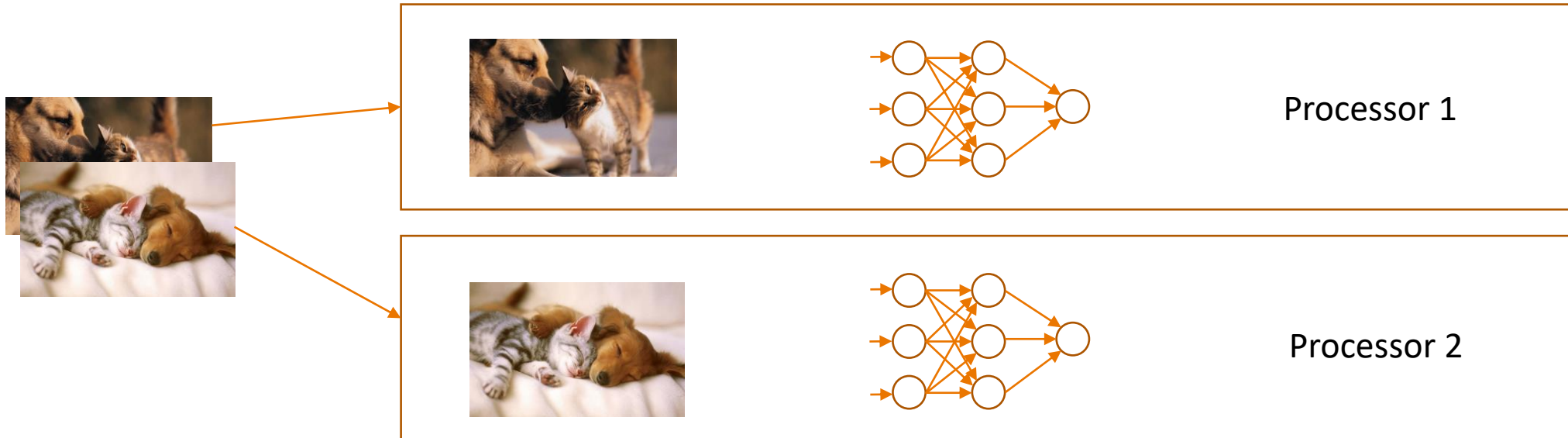
- Data parallelism
- Model parallelism
- Hybrid data/model parallelism
- Pipeline parallelism



Increasing complexity

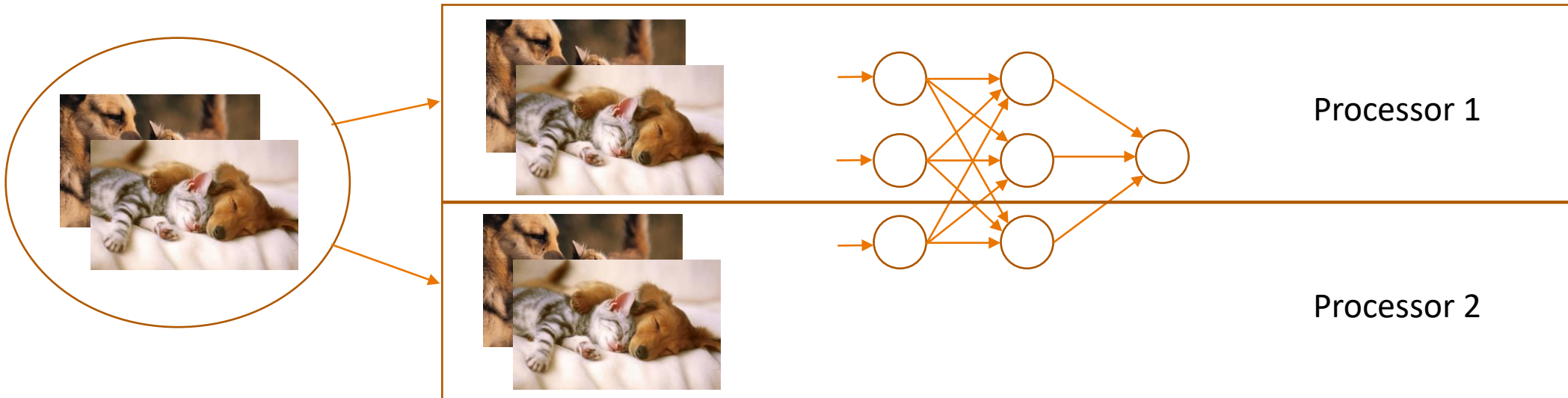
# Data Parallelism

- Split the batch over multiple processors (CPUs/GPUs)
- Each processor holds a copy of the model
- Forward pass: calculated by each of the workers
- Backward pass: gradients computed (per worker), communicated and aggregated

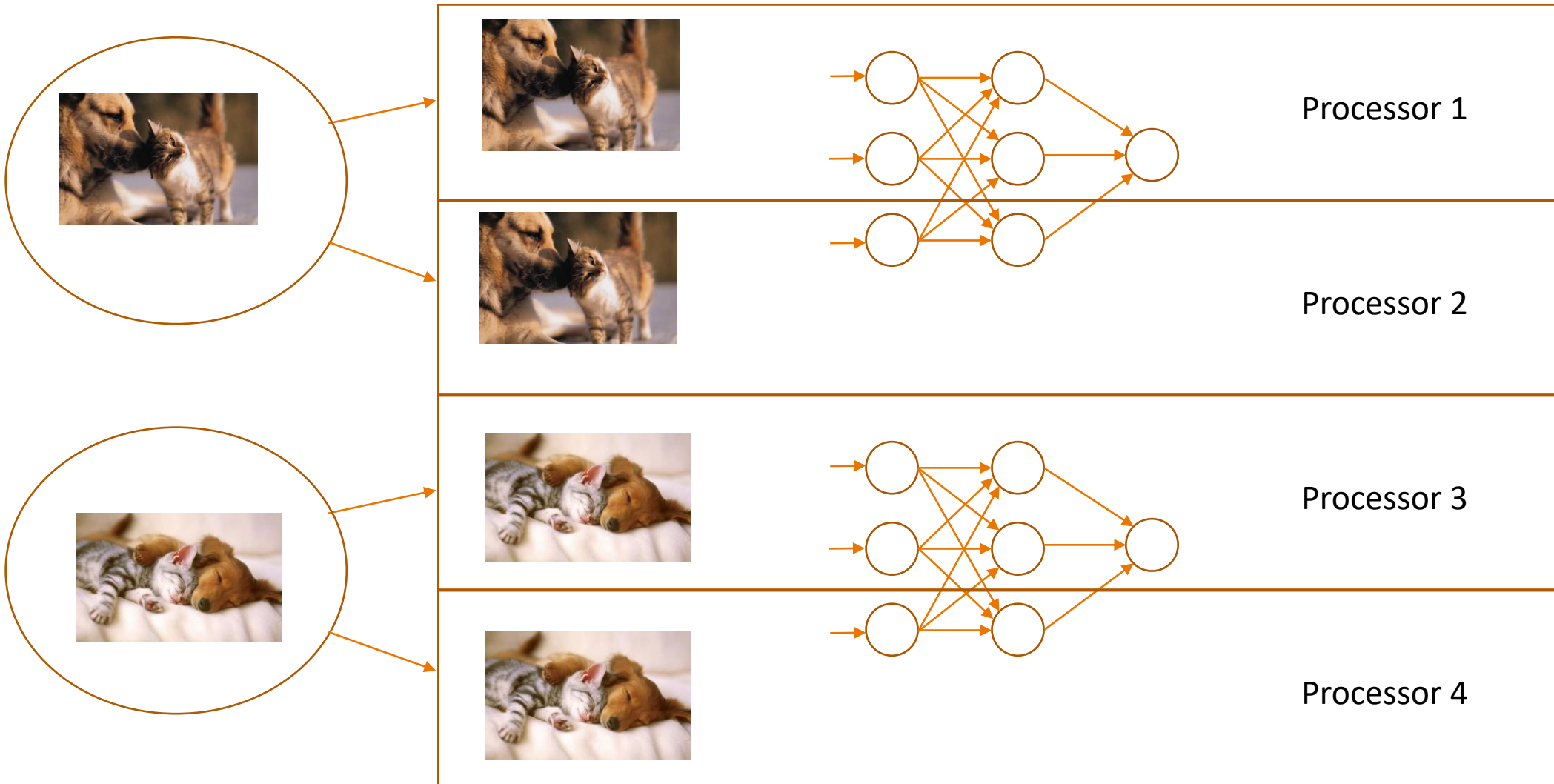


# Model parallelism

- Split the model over multiple processors (CPUs/GPUs)
- Each processor sees all the data
- Communication needed both during forward and backward pass!



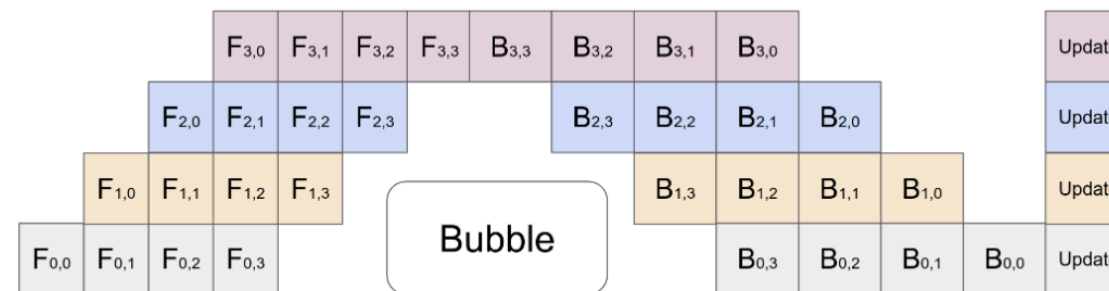
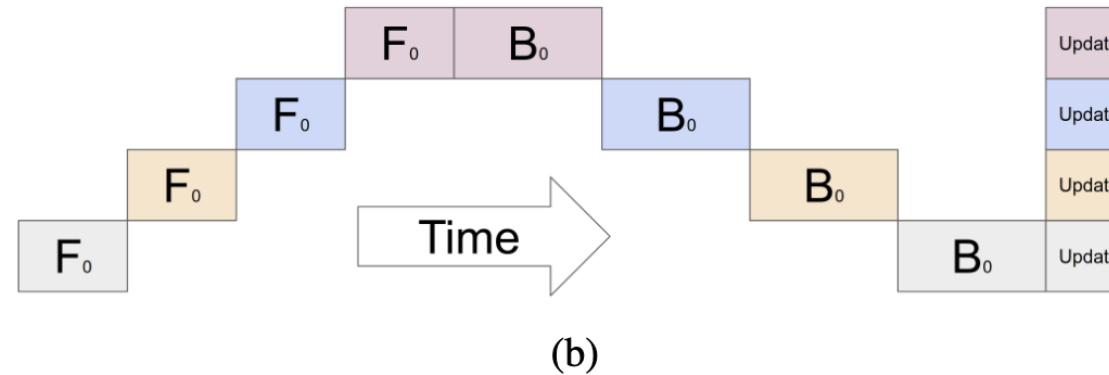
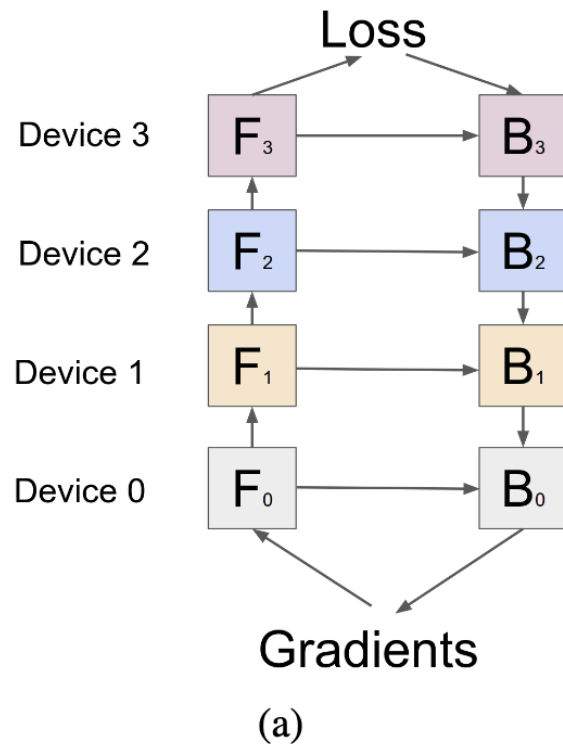
# Hybrid model/data parallelism





# Pipeline parallelism

- Model parallelism, executed in a pipelined fashion over multiple micro-batches.
- More efficient than model parallelism: hides communication with computation
- E.g. Gpipe (for PyTorch / TensorFlow), PipeDream (PyTorch)



(c) Huang et al. (2019, arXiv:[1811.06965](https://arxiv.org/abs/1811.06965))

# Parallelization for deep learning

Very generic statements (there are exceptions) on model vs data parallelism

	Data parallel	Model parallel
Throughput	Increases	Decreases
Implementation	Relatively straightforward	More difficult
Communication	Low to moderate	High
Typical use case	Speedup training	Models with large memory requirement

Use data parallel whenever you can. Use model parallel if you *really* need to. Even then, consider alternatives:

- Model pruning
- Use different hardware architecture (e.g. data parallel @ CPU)
- Use pipeline parallelism

# Data parallel stochastic gradient descent

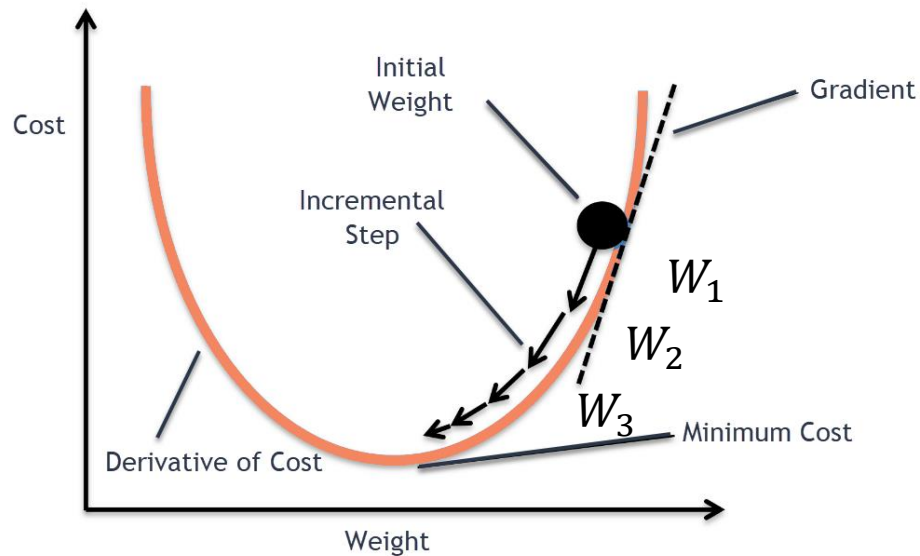
- Most networks are trained using stochastic gradient descent (SGD)
- Distributed stochastic gradient can be done in two ways
  - Synchronous SGD
  - Asynchronous SGD

# Stochastic gradient descent (SGD)

SGD: find optimum by following the slope

$$w = w - \eta \nabla Q(w)$$

$w$  = weights,  $\eta$  = learning rate,  $\nabla Q(w)$  = gradient for current batch.



# Data parallel synchronous SGD

- Each device (j) computes the gradients ( $\nabla Q_j(w)$ ) based on its own batch!
- Needs to be aggregated before updating weights



$$\nabla Q(w) = \sum_j \nabla Q_j(w)$$



$$w = w - \eta \nabla Q(w)$$

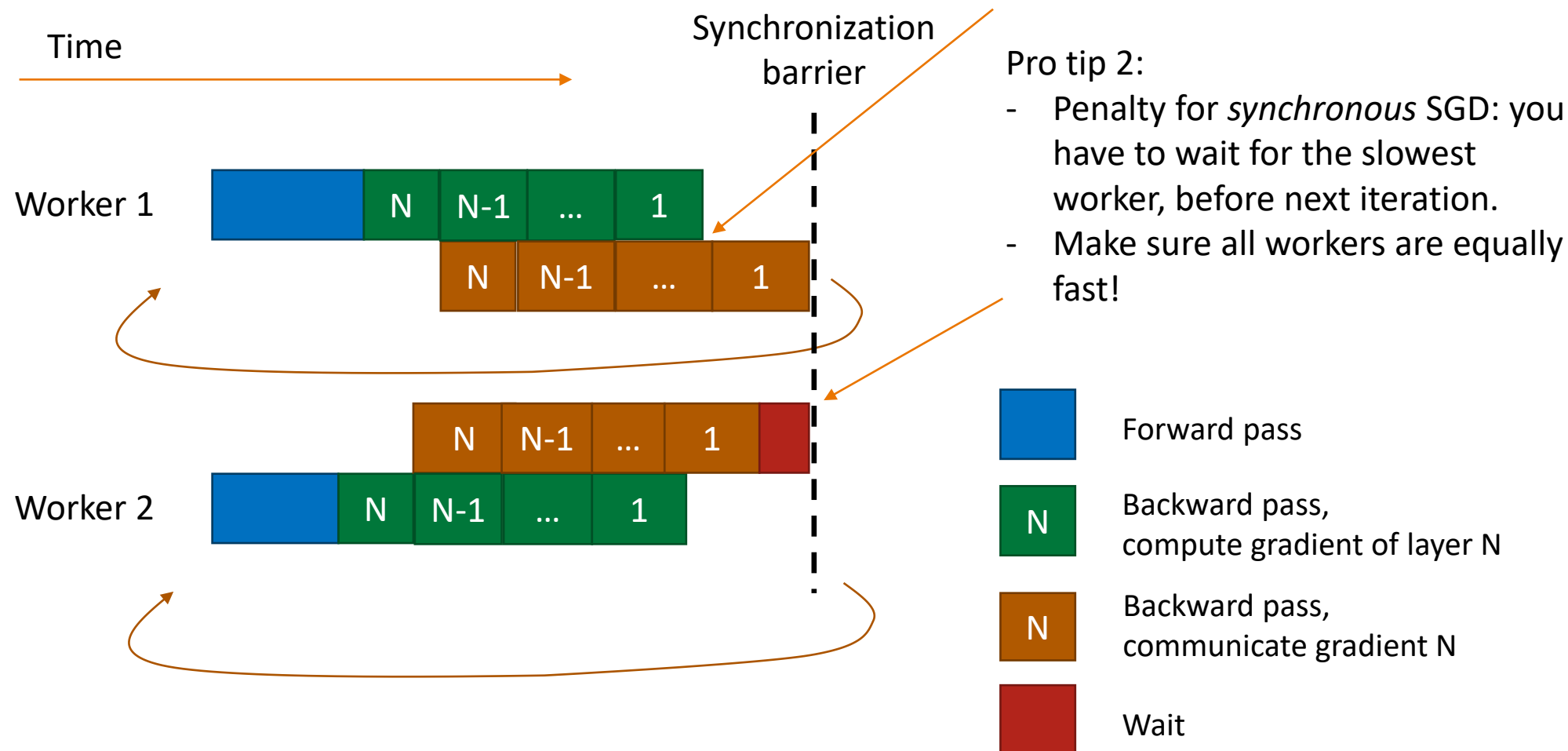
# Data parallel synchronous SGD

Effect on batch size:

- For  $N$  workers that each see  $n$  examples: batch size effectively  $n \times N$ .
- Larger batch => generally needs to be compensated by higher learning rate.
- No exact science!
  - Some use  $\eta_{distributed} = \eta_{serial} \cdot N$
  - Some use  $\eta_{distributed} = \eta_{serial} \cdot \sqrt{N}$
  - Experiment!

# Data parallel synchronous SGD

A different view...



Pro tip:

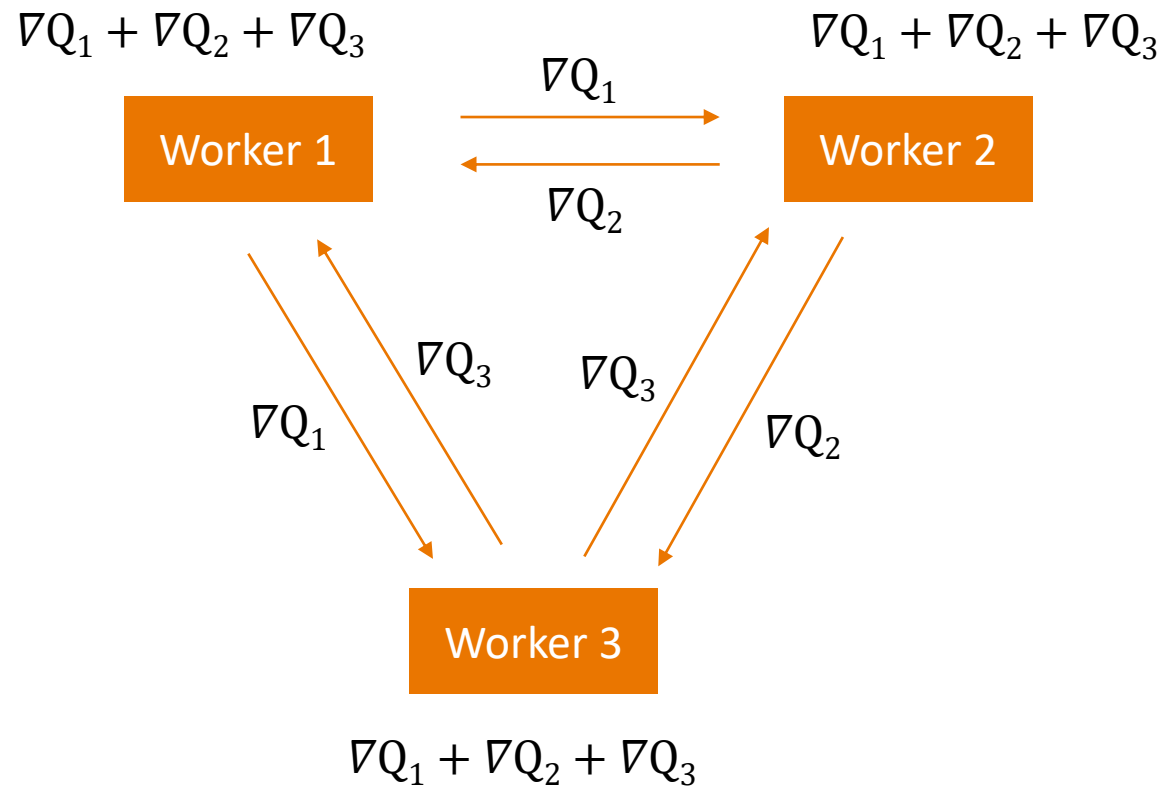
- Overlap communication and computation (don't waste compute cycles waiting for communication!)
- *Most* (distributed) DL frameworks already take care of this for you 😊

Pro tip 2:

- Penalty for *synchronous* SGD: you have to wait for the slowest worker, before next iteration.
- Make sure all workers are equally fast!

# Decentralized data parallel synchronous SGD

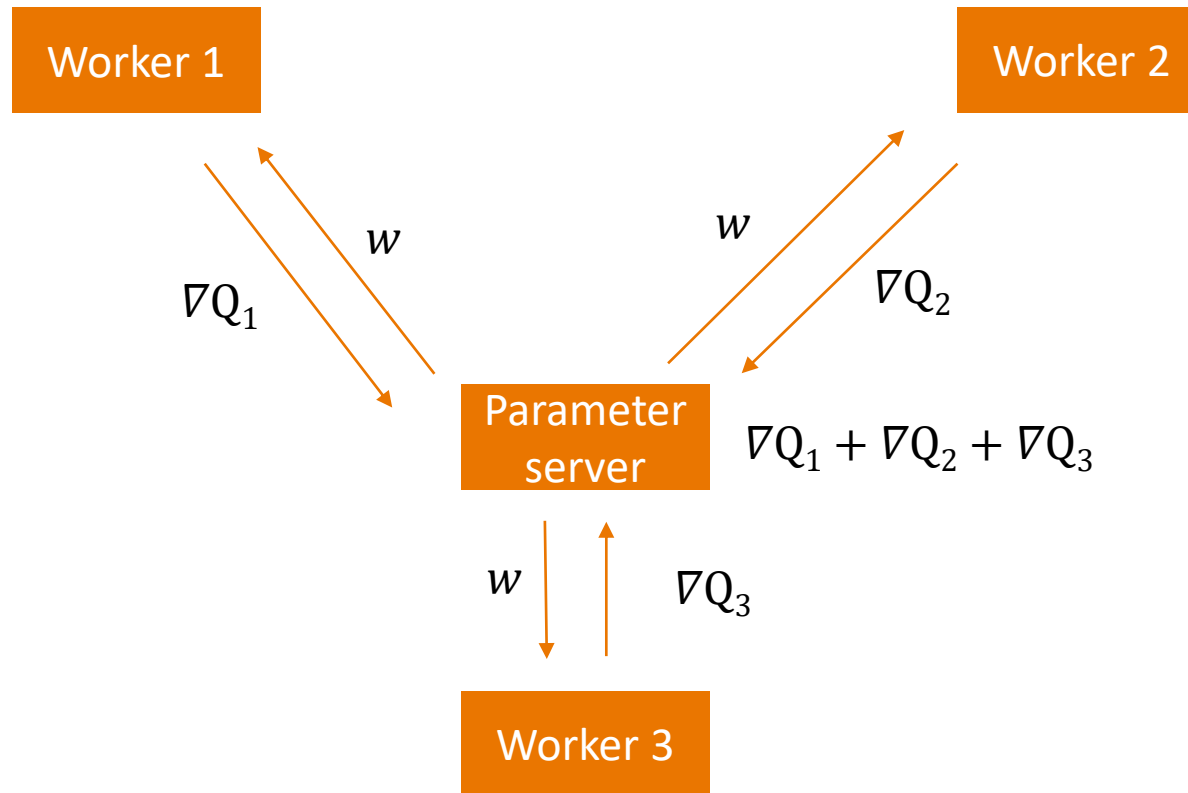
Gradients are communicated and aggregated by *all* workers





# Centralized data parallel synchronous SGD

There is also an alternative, where a parameter server is used to aggregate the gradients, and distribute the updated model:



# Centralized vs decentralized

- Centralized approach does not scale well: parameter servers create a communication bottleneck
- More info, see e.g. <https://arxiv.org/pdf/1705.09056.pdf>

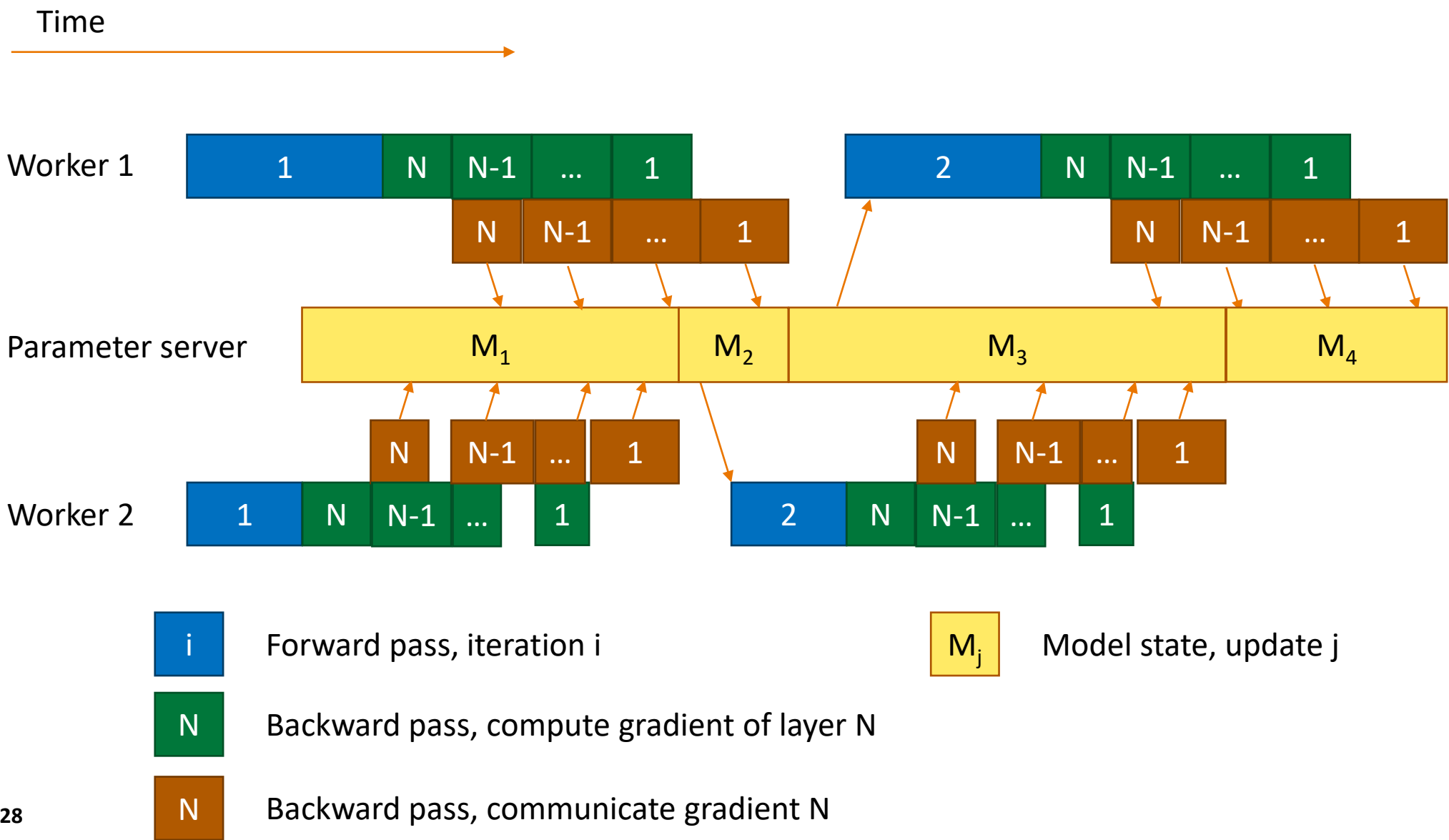
# Data parallel asynchronous SGD

Gradients are not aggregated, weights are updated as soon as *one* worker ( $j$ ) finishes, with the gradient of that worker:

$$w = w - \eta \nabla Q_j(w)$$

- If another worker finishes, it does an update on the *current* set of weights (even though the gradient may have been computed based on an earlier version of the weights)
- Asynchronous SGD does not have the same convergence guarantees as synchronous SGD
- Generally scales well, because there is no ‘barrier’ that induces wait time
- More info, see e.g. <https://ai.google/research/pubs/pub45187>

# Data parallel asynchronous SGD



# Data parallel SGD

Word of warning:

- Data parallel *asynchronous* SGD is an active area of research
- Data parallel *synchronous* SGD is well understood and is currently the *accepted* and *advised* approach

# Communicating gradients...

Ok, so the most widely accepted approach is

distributed...

data parallel...

synchronous...

SGD...

... but how do distributed deep learning frameworks aggregate their gradients in such a setup?

# A bit of history

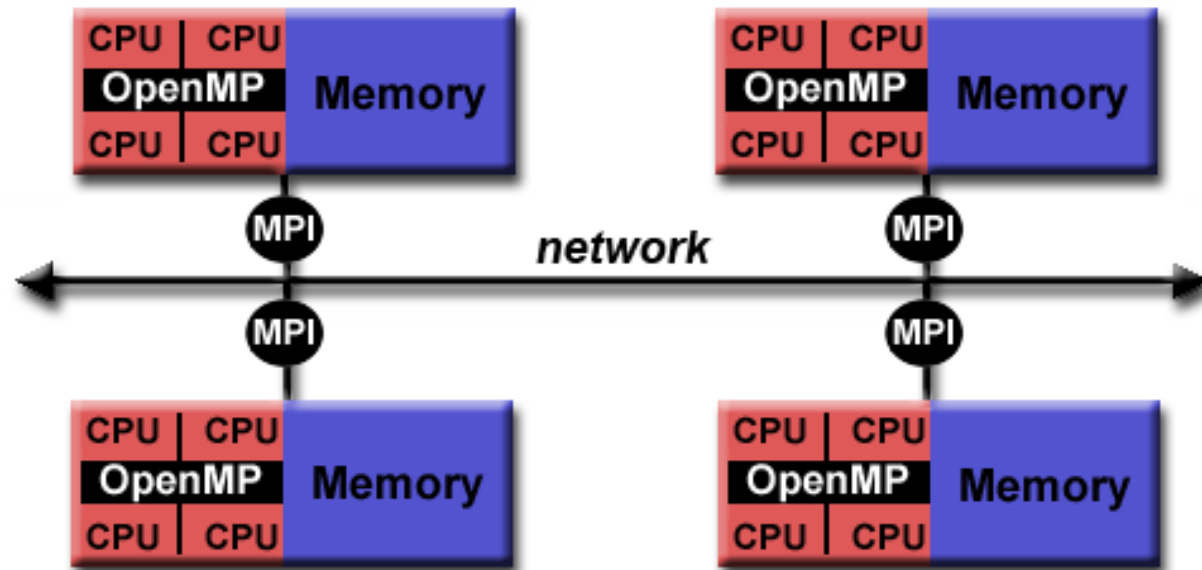
‘Traditionally’ a lot of machine learning was *not* done in an HPC context. As a result:

- Most frameworks had little focus on distributed learning
- Most frameworks that offered distributed learning were based on parameter servers
- Most AI experts probably never heard of MPI...

# The Message Passing Interface (MPI)

MPI is a standard for *parallelization on a distributed memory system*

- Distributed memory system: processors can't access each other's memory
- Explicit communication (over a network) is required between one memory and another to work on the same task
- MPI is the 'language' of this communication
- MPI is the *de facto* standard for traditional HPC applications





# The Message Passing Interface (MPI)

MPI has routines to send data between individual workers...



# The Message Passing Interface (MPI)

But also to broadcast data to other workers...



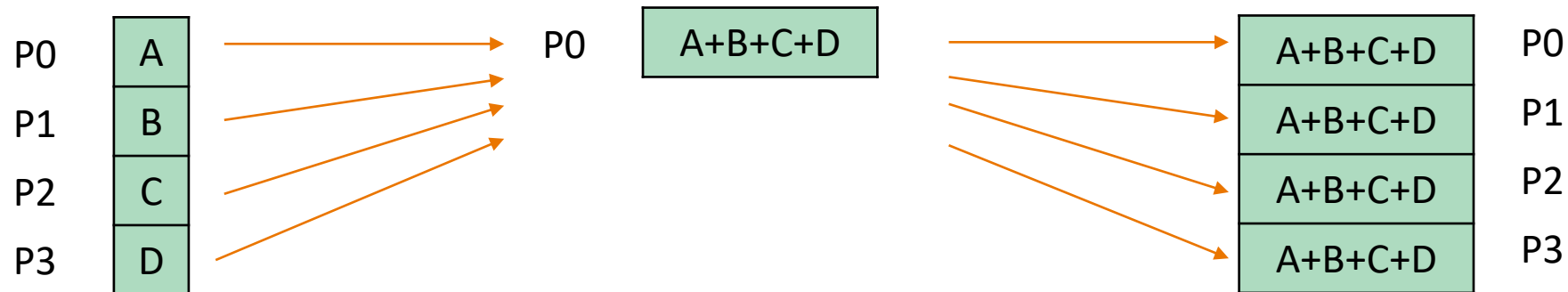
# The Message Passing Interface (MPI)

And most importantly (for deep learning): apply *collective* operations, such as 'allreduce'. This operation is widely used in distributed deep learning to aggregate gradients!



# The Message Passing Interface (MPI)

- MPI is a standard: it defines what AllReduce should *do*, not *how it should be done*.
- MPI libraries implement MPI functions. These libraries decide *how it should be done*.
- Example: an inefficient allreduce operation could be implemented like this:



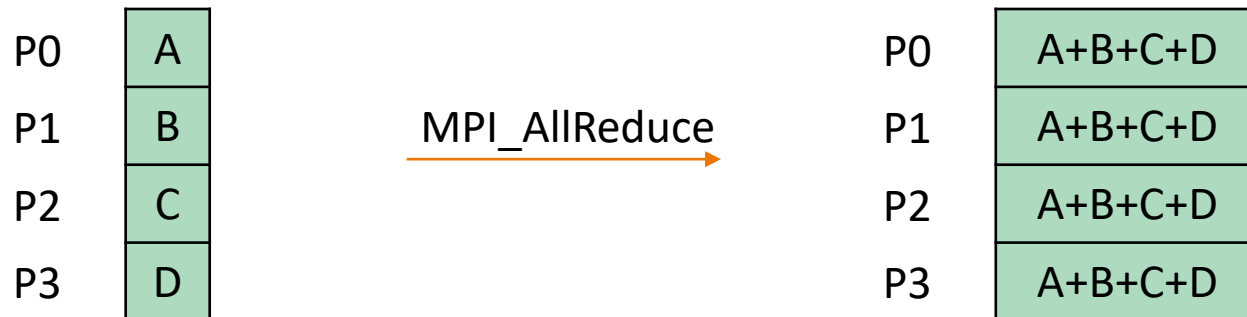
# Relevance of MPI for distributed deep learning

Why is this relevant?

- Distributed DL frameworks often support multiple communication backends for their collective allreduce operations
- These backends often either implement (part of) the MPI API or something similar
- It is important to pick a communication backend with an efficient implementation.
- The most efficient implementation may vary per hardware.
- Example: Nvidia's NCCL library implements a subset of MPI collective operations. These implementations are highly optimized for Nvidia GPUs.

# MPI / NCCL

- MPI often used to communicate gradients
- MPI\_AllReduce aggregates and sums gradients (remember:  $\nabla Q(w) = \sum_j \nabla Q_j(w)$ )
- NVIDIA's NCCL library contains an implementation of MPI routines optimized for GPU  $\Leftrightarrow$  GPU communication



# Frameworks for distributed learning

- TensorFlow's tf.distribute: quite tricky to program. Lot's of code changes needed from serial to distributed ([https://www.tensorflow.org/guide/distributed\\_training](https://www.tensorflow.org/guide/distributed_training))
- TensorFlow + Horovod: serial => distributed with minimal code changes (<https://horovod.readthedocs.io/en/stable/tensorflow.html>)
- PyTorch's torch.distributed ([https://pytorch.org/tutorials/intermediate/dist\\_tuto.html](https://pytorch.org/tutorials/intermediate/dist_tuto.html))
- PyTorch + Horovod: serial => distributed with minimal code changes (<https://horovod.readthedocs.io/en/stable/pytorch.html>)
- PyTorch Lightning: hides a lot of boiler plate code (also nice for serial training). Very little changes needed between serial & parallel execution, especially on a SLURM cluster (<https://pytorch-lightning.readthedocs.io/en/latest/clouds/cluster.html#slurm-managed-cluster>)

# Performance considerations

Bottlenecks can be:

- I/O, especially with distributed training (many processes reading the same files)
  - Tip: don't read many small files, pack samples together into a few large files
- Communication
  - Don't use parameter servers. Use NCCL backend on NVIDIA GPUs. Don't try to parallelize very light workloads – communication overhead too large
- Memory bandwidth
  - High level frameworks (PyTorch, TensorFlow) use optimized low level libraries (cuDNN, MKL-DNN, etc) for optimized



# Recap of goal: understand docs of DL frameworks

From TensorFlow docs on “distribution strategy”:

- “tf.distribute.Strategy intends to cover a number of use cases along different axes...  
Synchronous vs asynchronous training: These are two common ways of distributing training with data parallelism. In sync training, all workers train over different slices of input data in sync, and aggregating gradients at each step. In async training, all workers are independently training over the input data and updating variables asynchronously. Typically sync training is supported via all-reduce and async through parameter server architecture.”
- “MultiWorkerMirroredStrategy currently allows you to choose between two different implementations of collective ops. CollectiveCommunication.RING implements ring-based collectives using gRPC as the communication layer. CollectiveCommunication.NCCL uses Nvidia's NCCL to implement collectives.”

# Practical tips & take home messages

- If increased throughput is the goal, use data parallelism
- If a large model is the goal, use model (or hybrid or pipeline) parallelism, but consider the consequences (slower training) and alternatives (model pruning, CPU-based training, etc)
- Account for the difference in convergence behavior of data parallel SGD, e.g. by adjusting & experimenting with the learning rate.
- *Synchronous* parallel SGD is the most common approach for distributed learning, because it is well understood. *Asynchronous* parallel SGD can scale very well, but convergence behavior is less clear.
- Use an efficient backend for collective communications (e.g. NCCL)

# Further reading

- Distributed TensorFlow using Horovod: <https://towardsdatascience.com/distributed-tensorflow-using-horovod-6d572f8790c4>
- Demystifying Parallel and Distributed Deep Learning: An In-Depth Concurrency Analysis: <https://arxiv.org/pdf/1802.09941.pdf>
- Prace best practice guide for Deep Learning: <http://www.prace-ri.eu/IMG/pdf/Best-Practice-Guide-Deep-Learning.pdf>
- Technologies behind Distributed Deep Learning: <https://preferredresearch.jp/2018/07/10/technologies-behind-distributed-deep-learning-allreduce/>
- PyTorch Distributed: [https://pytorch.org/tutorials/beginner/dist\\_overview.html](https://pytorch.org/tutorials/beginner/dist_overview.html) and <https://pytorch.org/docs/stable/distributed.html>