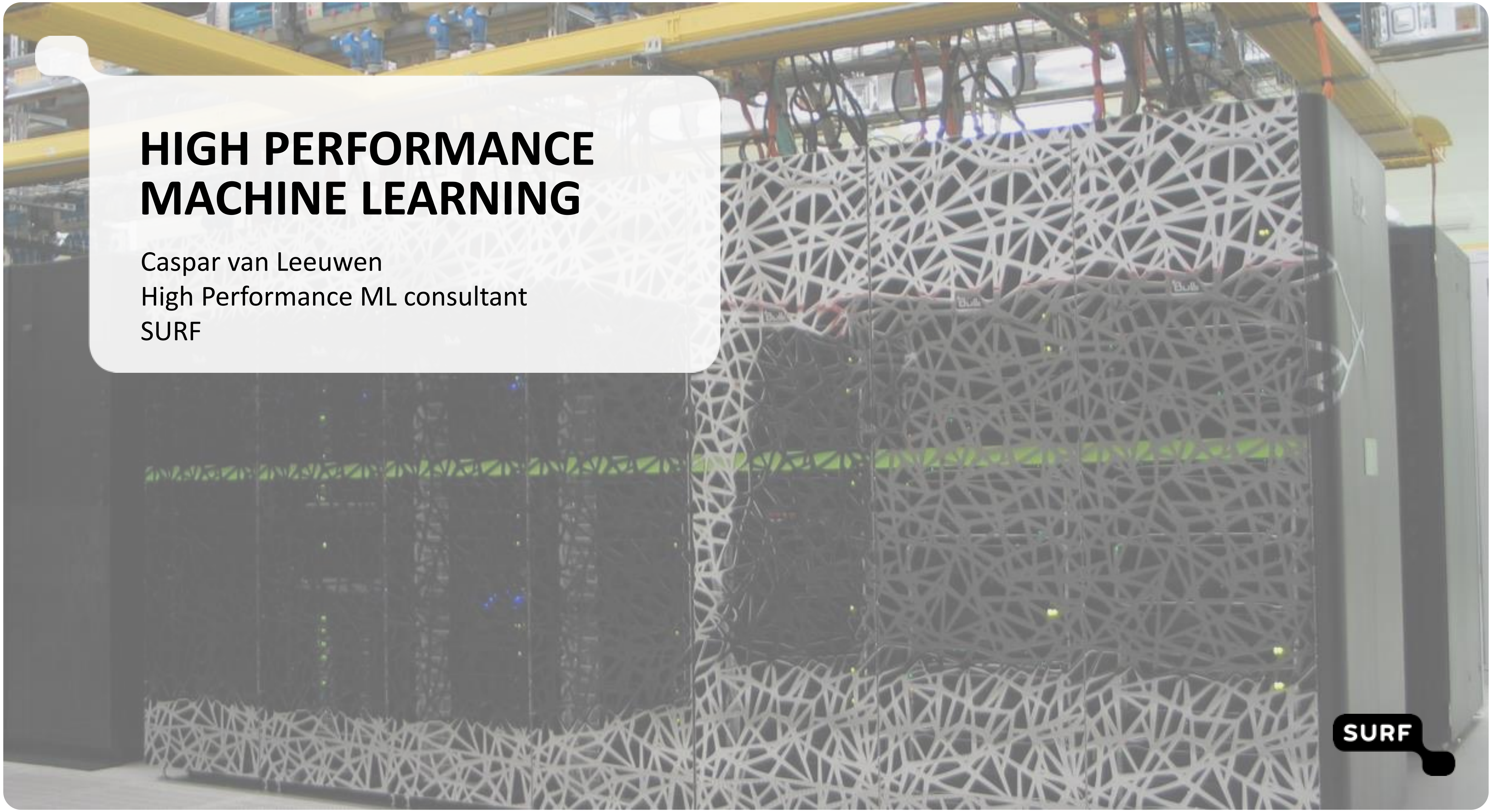


HIGH PERFORMANCE MACHINE LEARNING

Caspar van Leeuwen
High Performance ML consultant
SURF

Program, 2nd day

- 9:00 – 9:30 Introduction to Parallel Computing (Caspar van Leeuwen)
- 9:30 – 10:30 Parallel Computing for Deep Learning: ideas, frameworks, and hardware bottlenecks (Caspar van Leeuwen)
- 10:30 – 11:00 Coffee break
- 11:00 – 11:30 Structure of Deep Learning Frameworks: computational graph, autodiff, and optimizers (Joris Mollinga)
- 11:30 – 12:30 Hands-on: Profiling TensorFlow with TensorBoard (Caspar van Leeuwen)
- 12:30 – 14:00 Lunch Break
- 14:00 – 15:00 Hands-on: Data Parallelism with Horovod (CIFAR10) (Joris/Maxwell)
- 15:00 – 15:30 Coffee break
- 15:30 – 16:15 Introduction to hybrid parallelism (Caspar van Leeuwen)
- 16:15 – 17:00 Open Discussion

Parallel computing for Deep Learning

Goals: to understand…

- benefits of parallellization
- different parallellization strategies (pro’s and con’s)
- concepts of parallel stochastic gradient descent (SGD)
- synchronous and asynchronous SGD
- how communication backends are used by distributed DL frameworks for gradient aggregation
- the 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."

Parallellization: why?

Time to train ResNet-50 on Imagenet 1k (top-1 Accuracy +/-75%)

-	1 × Tesla P100	1-2 weeks
He et al [2016]	8 × Tesla P100	29 h
Goyal et al [2017]	256 × Tesla P100	1 h
Cho et al [2017]	256 × Tesla P100	50 min
Codreanu et al [2017]	1024 × KNL	42 min
You et al [2017]	2048 × KNL	20 min
Akiba et al [2017]	1024 × P100	15 min
Jia et al [2018]	1024 × P40	8.7 min
Jia et al [2018]	2048 × P40	6.6 min
Mikami et al [2018]	2176 × V100	3.7 min

Source: Jia et al https://arxiv.org/pdf/1807.11205.pdf ;
Mikami et al https://nnabla.org/paper/imagenet_in_224sec.pdf

Parallellization: why?

Faster trainings …

- Enables learning on larger datasets
- Enables improved accuracy through better hyperparemeter tuning
- Enables larger, more complex models
- …

Bigger models (high memory requirement) …

- Enables larger, more complex models

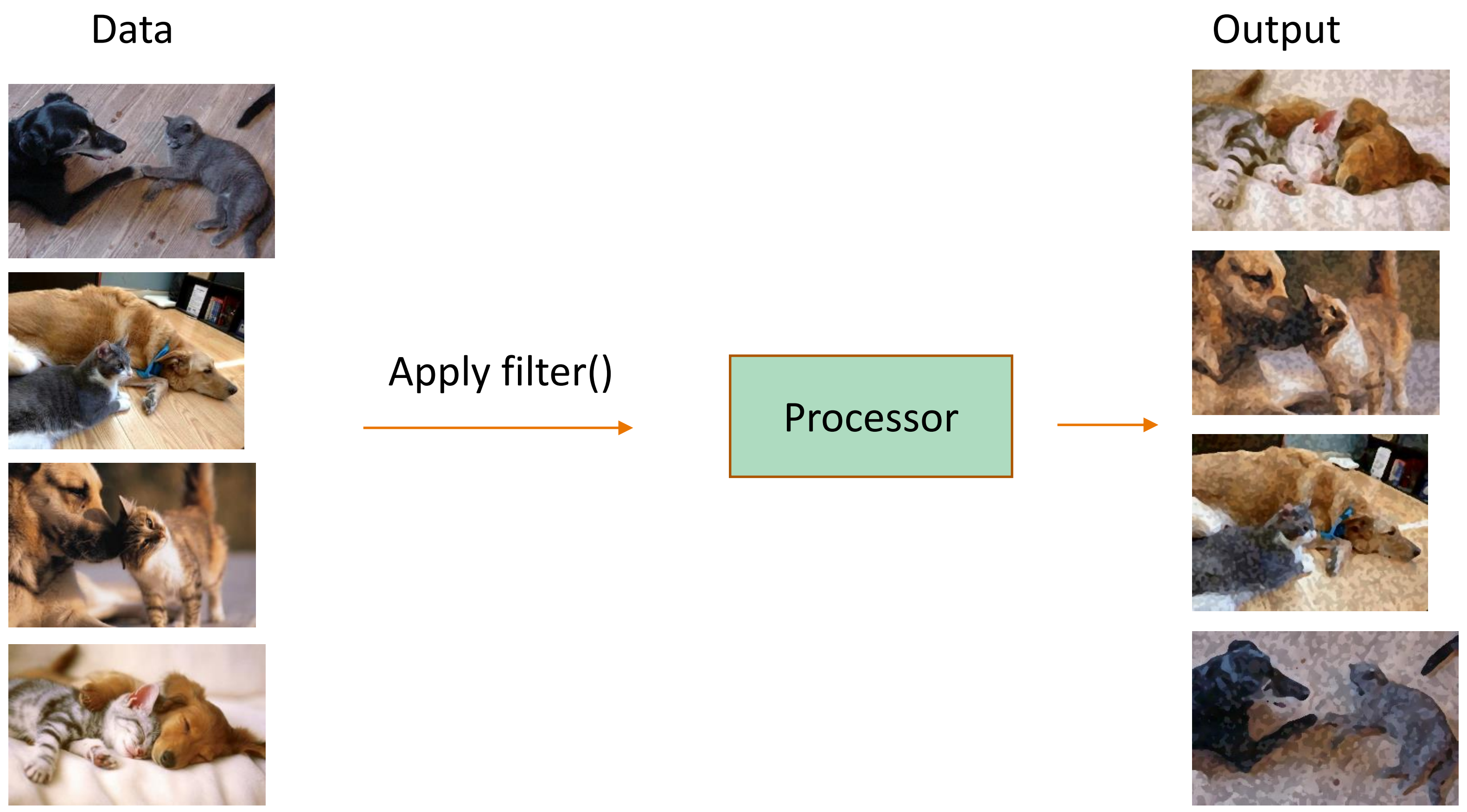
Parallelization: the basics

What is parallel computing?

- Multiple processors or computers working on a single computational problem

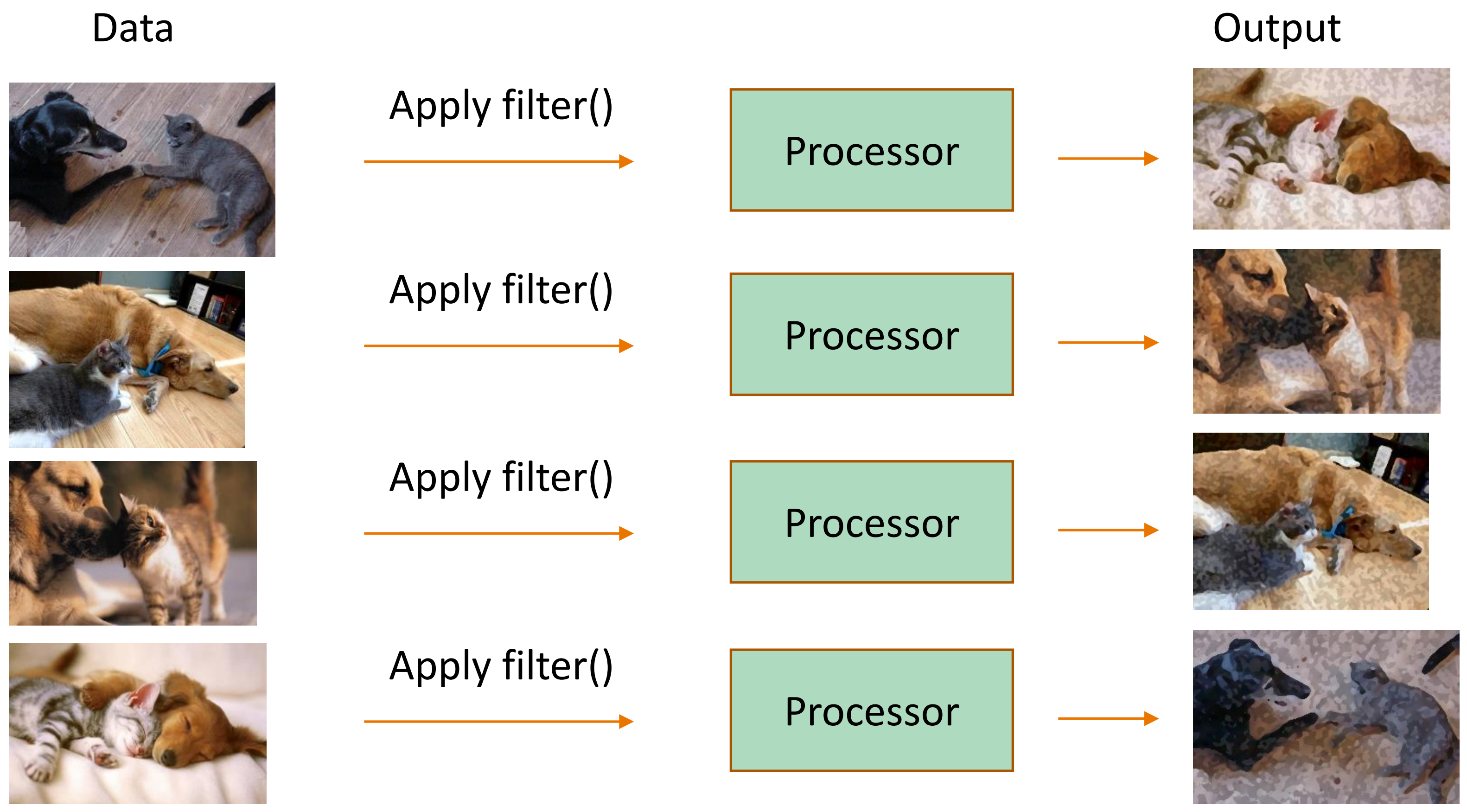
Parallelization: the basics

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?

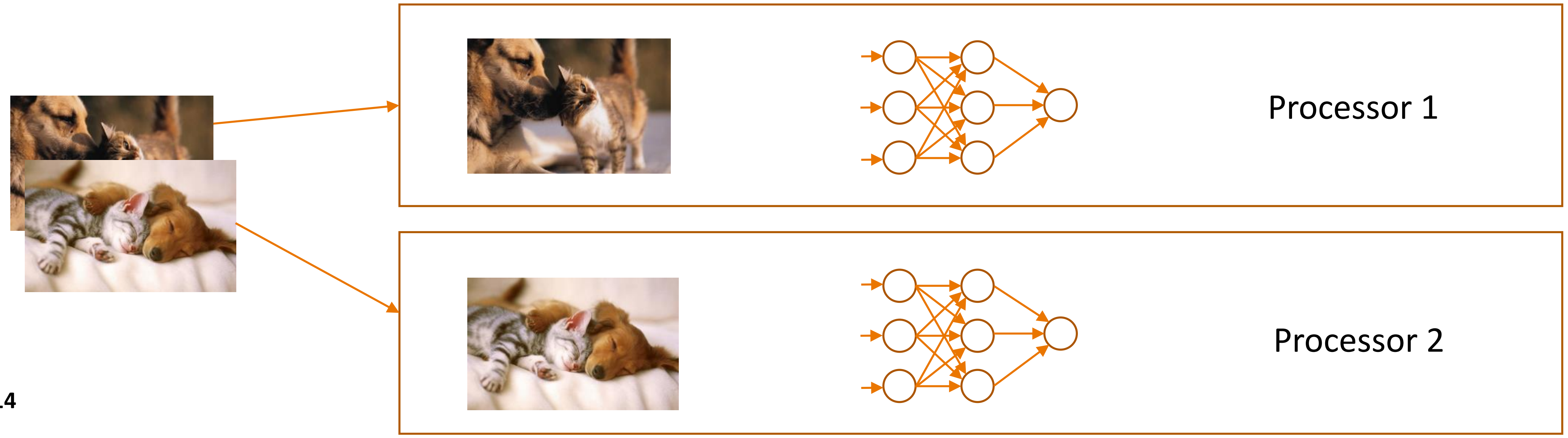
- Data parallelism
- Model parallelism
- Hybrid data/model parallelism
- Pipeline parallelism

$\downarrow$ Increasing complexity

Parallelization for deep learning

How does one 'divide' DL tasks?

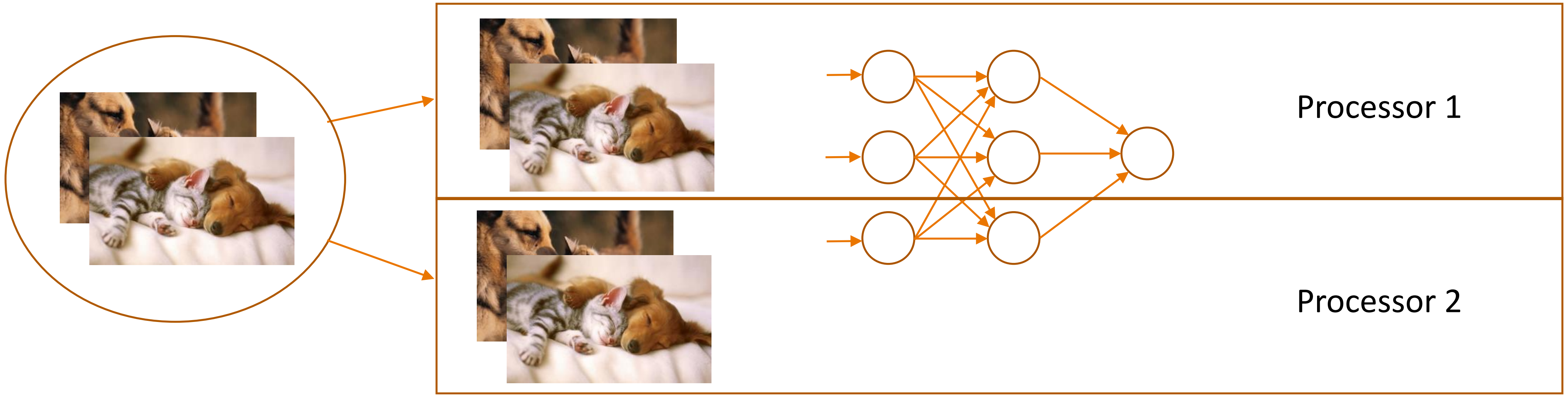
- Data parallelism
  - Split the batch over multiple processors (CPUs/GPUs)
  - Each processor holds a copy of the model
  - Forward pass: calculated by each of the workers
  - Backward pass: gradients computed (per worker), communicated and aggregated



Parallelization for deep learning

How does one ‘divide’ DL tasks?

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



Parallelization for deep learning

Hybrid parallelism

- Mix of model & data parallelism
- Model parallelism to fit larger models; data parallelism to speed up
- Some support in frameworks, see e.g. Mesh Tensorflow
  https://github.com/tensorflow/mesh

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)
- More this afternoon!

Parallelization for deep learning

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

Data parallel	Model parallel
Increases throughput compared to serial	Decreases throughput compared to serial
Relatively easy to implement	More difficult to implement
Some communication overhead	High communication overhead

In general, only use model parallel if you *really* need to (and even then, consider pruning your model to fit memory of a single processor, or use a different architecture with more memory…).

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

Most networks are trained based on stochastic gradient descent (SGD):

$$w = w - \eta \nabla \mathrm{Q}(\mathrm{w})$$

w = weights, η = learning rate, $\nabla \mathrm{Q}(\mathrm{w})$ = gradient for current batch.

In data parallel SGD, each worker (j) computes the gradients ($\nabla \mathrm{Q}_{\mathrm{j}}(\mathrm{w})$) based on its own batch!

Data parallel synchronous SGD

Gradients are aggregated

$$\nabla \mathrm{Q}(\mathrm{w}) = \sum_j \nabla \mathrm{Q}_{\mathrm{j}}(\mathrm{w})$$

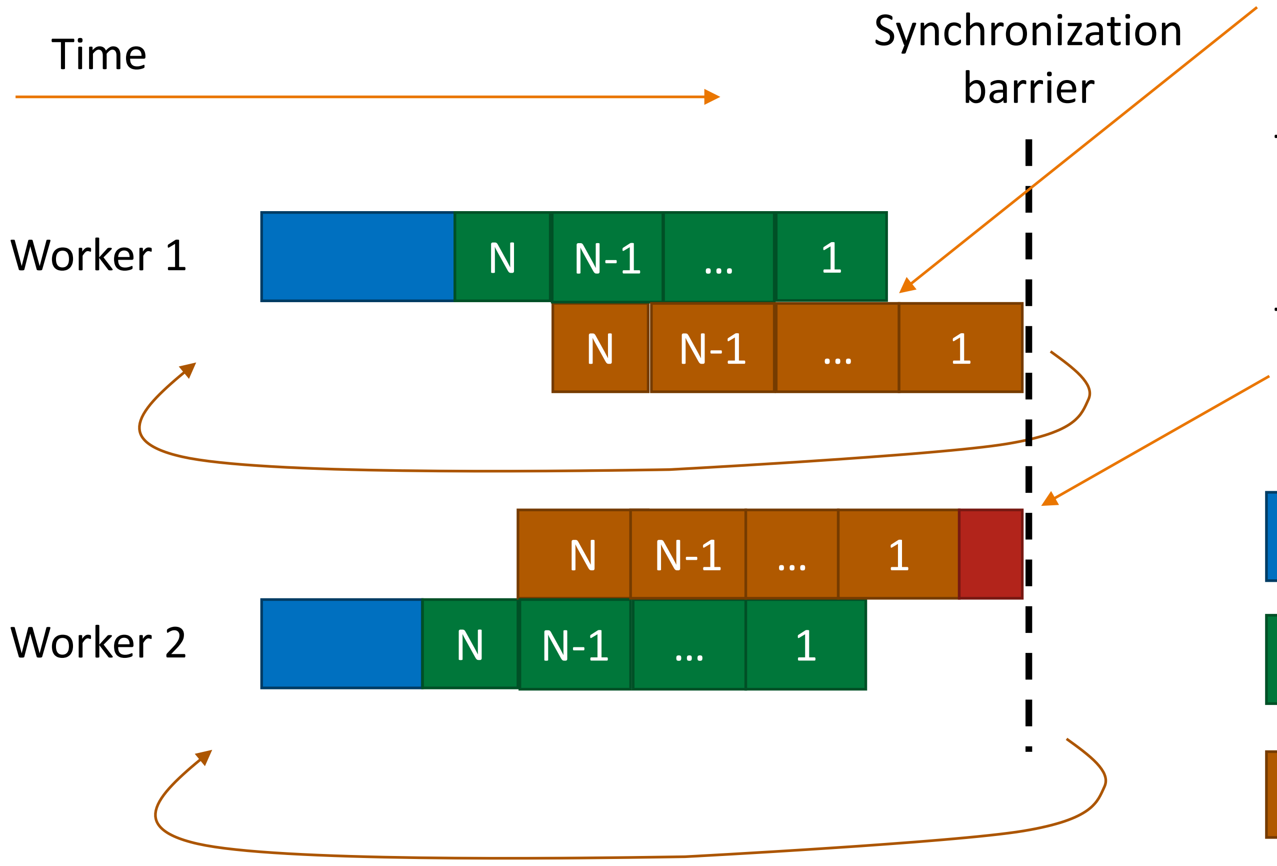
Before the model weights are updated

$$w = w - \eta \nabla \mathrm{Q}(\mathrm{w})$$

- For *N* workers that each see *n* examples: batch size effectively n × 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…



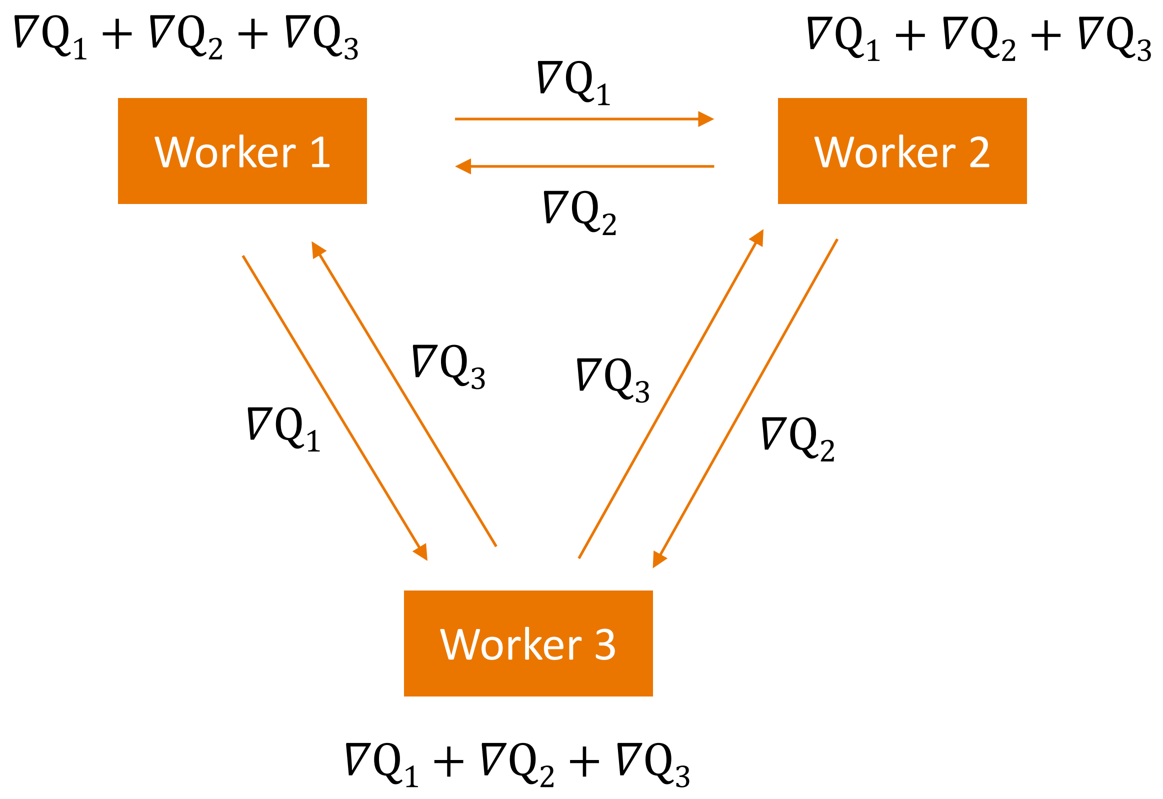
Pro tip:
- Overlap communication and computation (don't waste compute cycles waiting for communication!)
- *Most* (distributed) DL frameworks already take care of this for you ☺

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!

🟦	Forward pass
🟩 N	Backward pass, compute gradient of layer N
🟧 N	Backward pass, communicate gradient N
🟥	Wait

Decentralized data parallel synchronous SGD

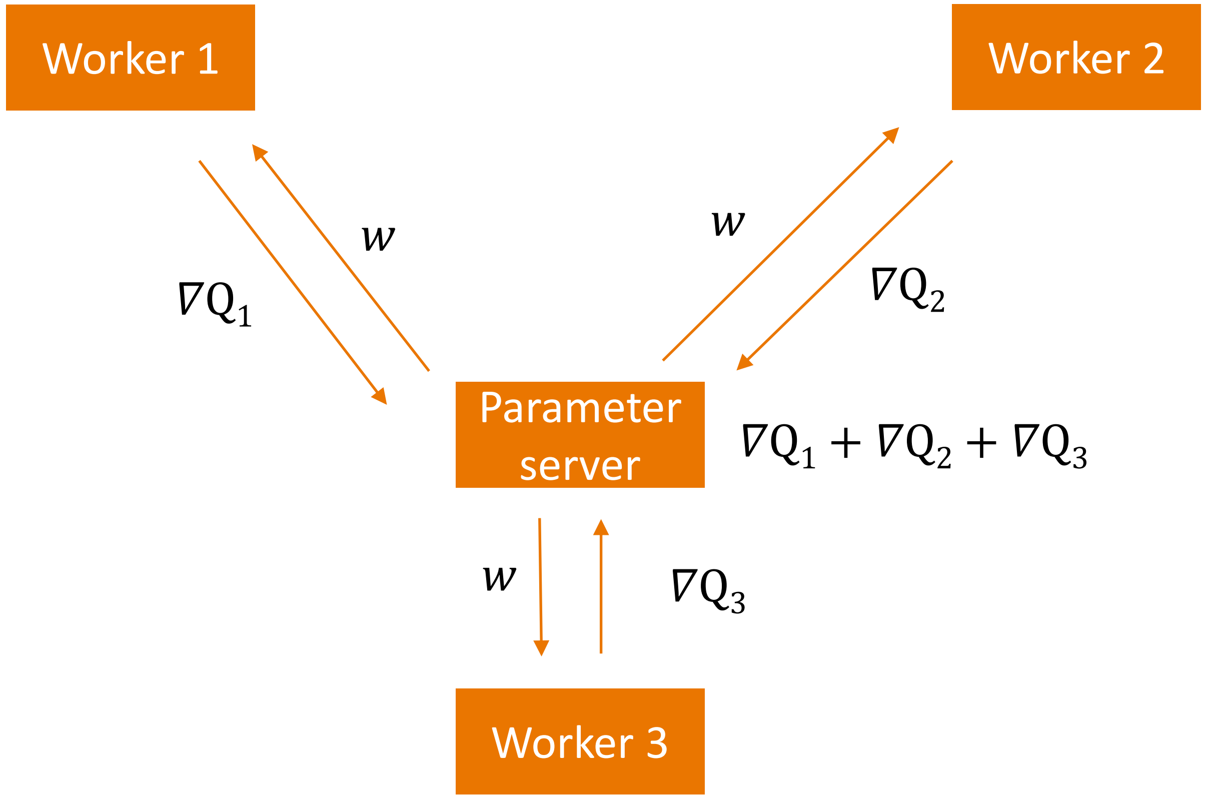
Gradients are communicated and aggregated by *all* workers



$\nabla Q_1 + \nabla Q_2 + \nabla Q_3$
$\nabla Q_1 + \nabla Q_2 + \nabla Q_3$
∇Q_1
Worker 1
Worker 2
∇Q_2
∇Q_3
∇Q_3
∇Q_1
∇Q_2
Worker 3
$\nabla Q_1 + \nabla Q_2 + \nabla Q_3$

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 more ‘traditional’, and implemented in many DL frameworks
- Centralized approach does not scale well: parameter servers create a bottleneck
- More info, see e.g. https://arxiv.org/pdf/1705.09056.pdf

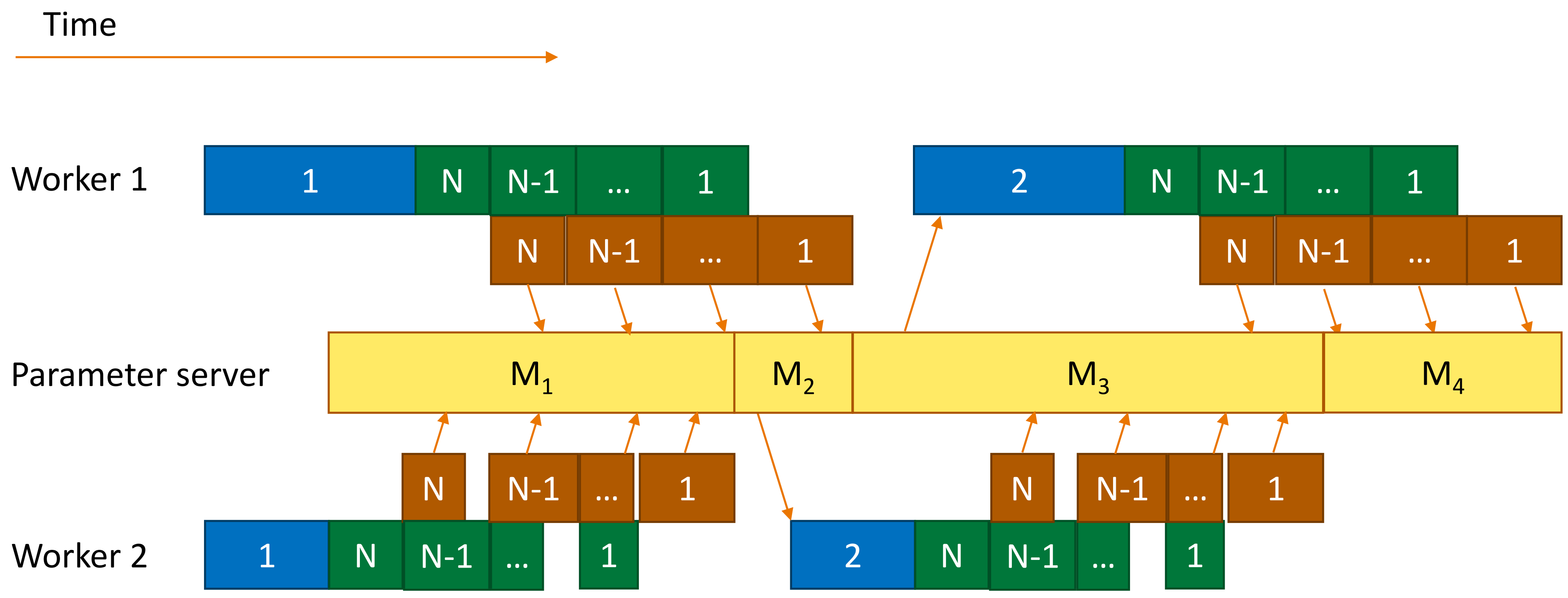
Data parallel asynchronous SGD

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

$$w = w - \eta \nabla \mathrm{Q_j}(\mathrm{w})$$

- If another worker finishes, it does an update on the *current* set of weights (even though the gradient may have be 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



🟦 i	Forward pass, iteration i
🟩 N	Backward pass, compute gradient of layer N
🟫 N	Backward pass, communicate gradient N
🟨 M_j	Model state, update j

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

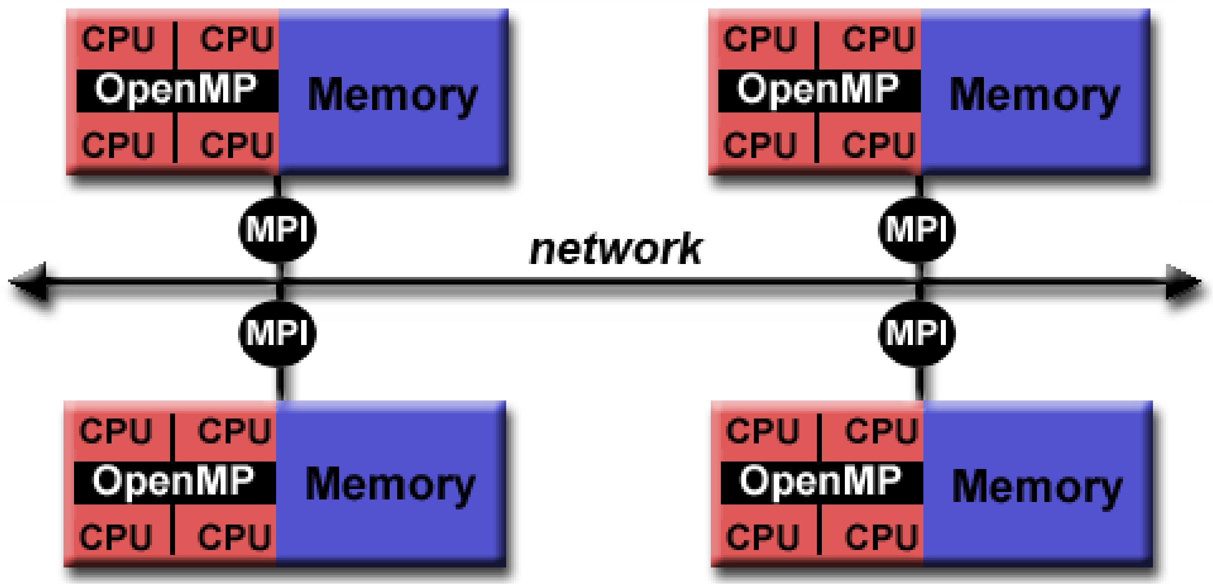
‘Traditionally’ a lot of machine learning was *not* done in an HPC context. AI experts would be very happy with one powerful GPU (and a lot of people still are). As a result:

- 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…

FROM: NODE 0
TO: NODE 1

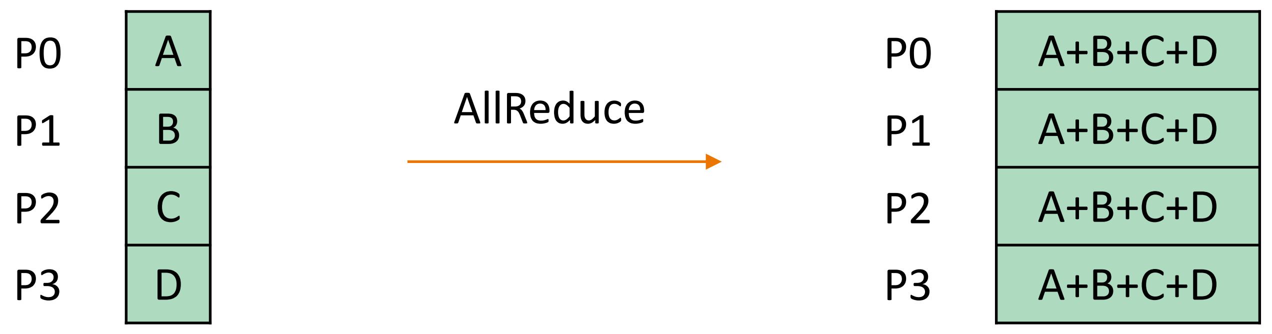
The Message Passing Interface (MPI)

But also to broadcast data to other workers…

FROM: NODE 0
TO: EVERYONE

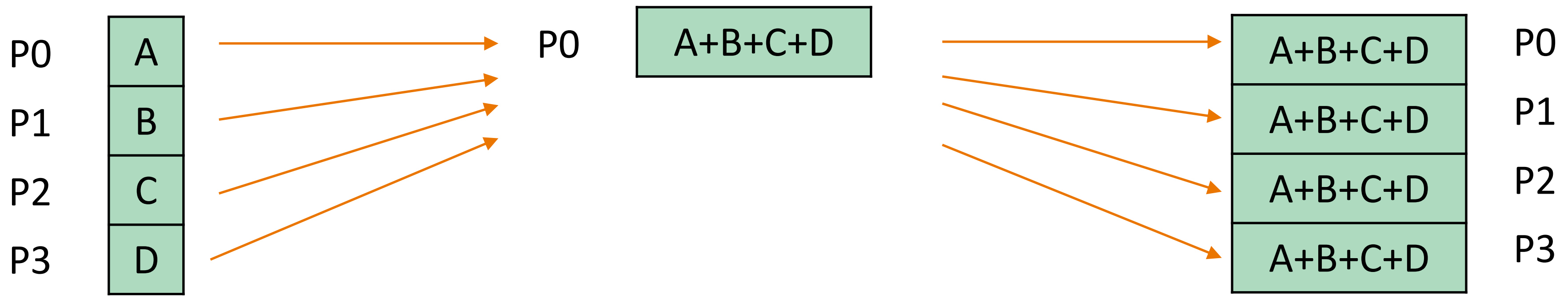
The Message Passing Interface (MPI)

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 <u>inefficient</u> allreduce operation could 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.

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 communication penalty you will pay. Switching to an architecture with more memory might be better!
- 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.

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/