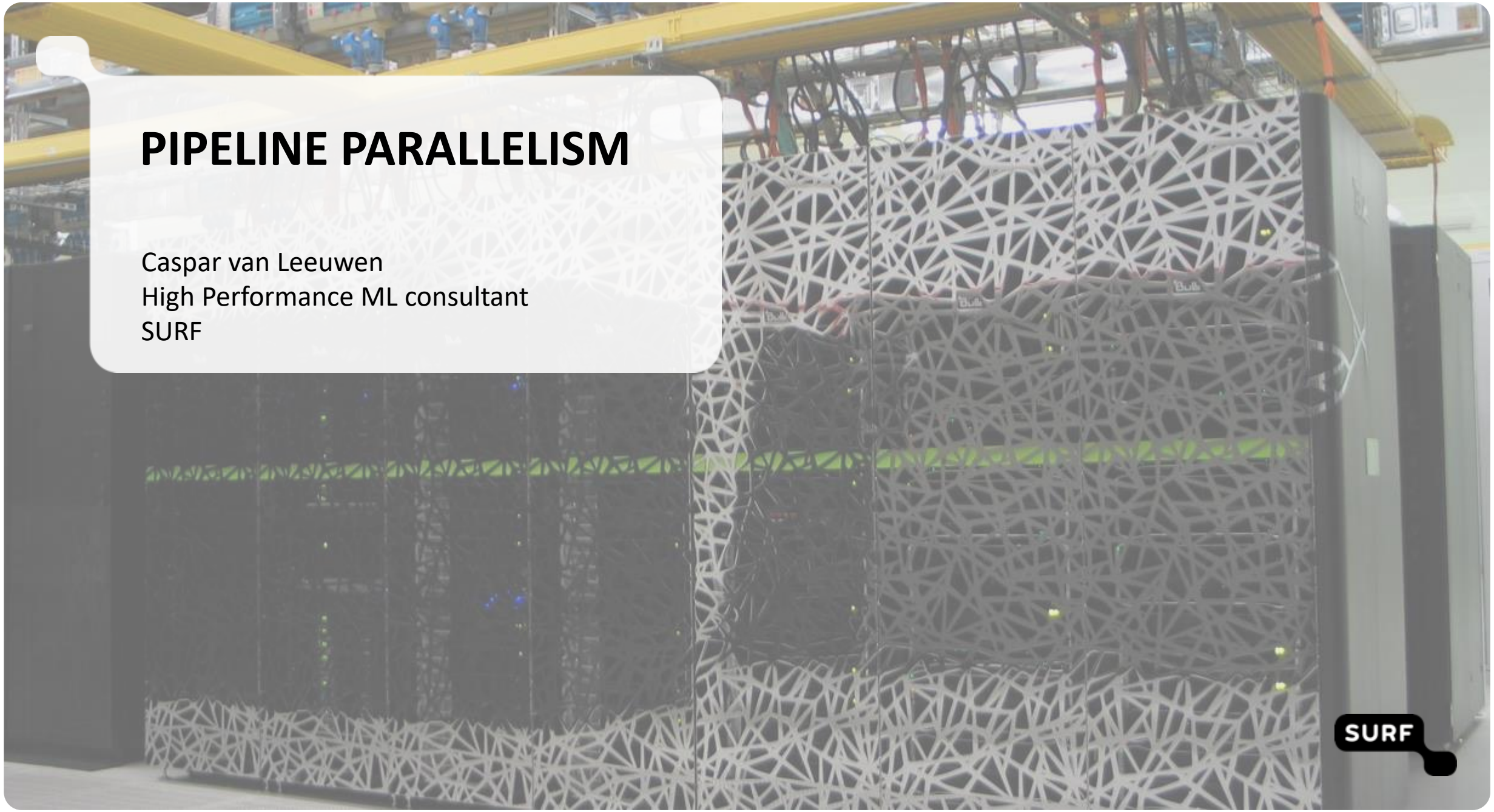


PIPELINE PARALLELISM

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

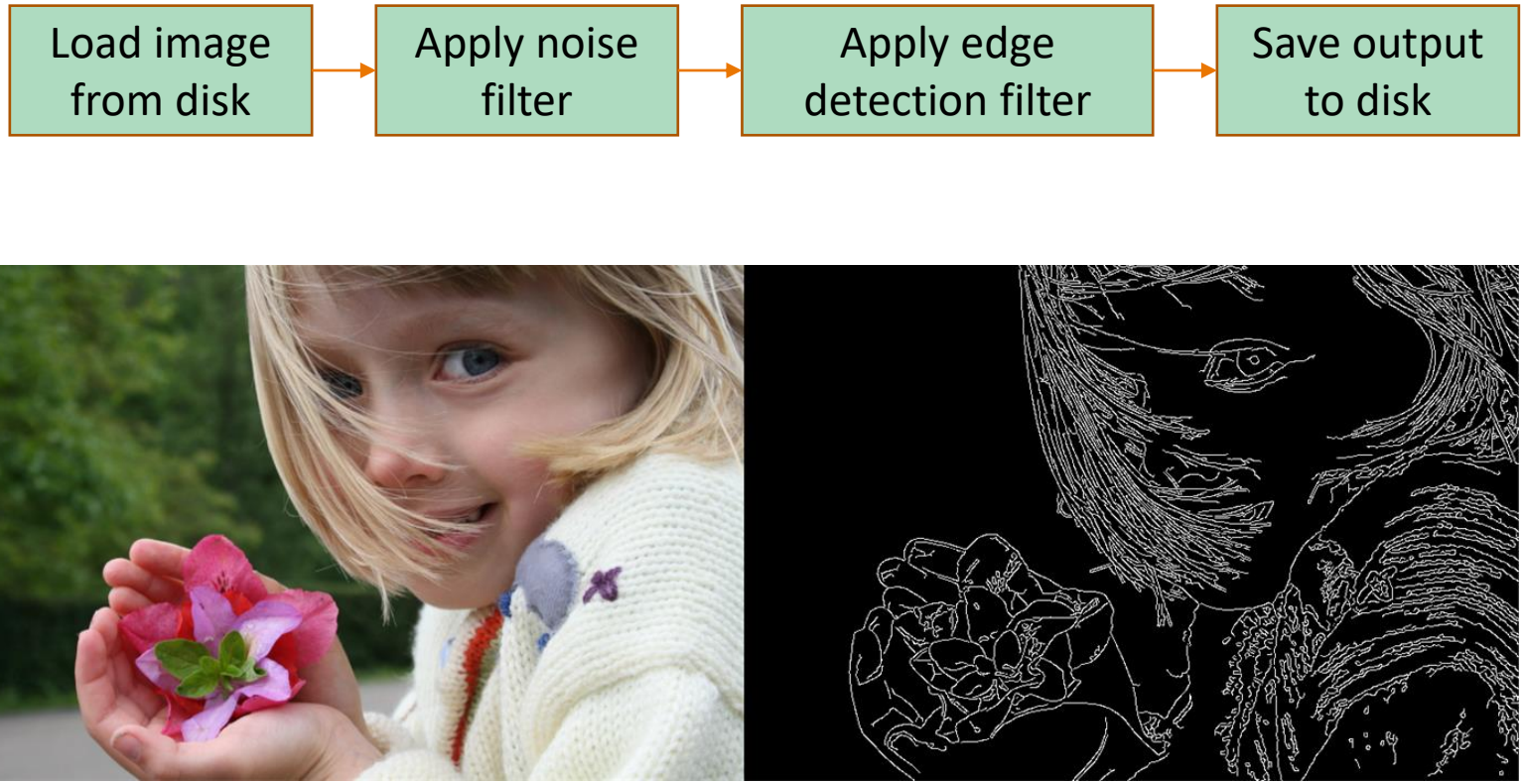
Pipeline parallelism

Goals:

- Understand pipeline parallelism
- Get first hands-on experience implementing pipeline parallelism

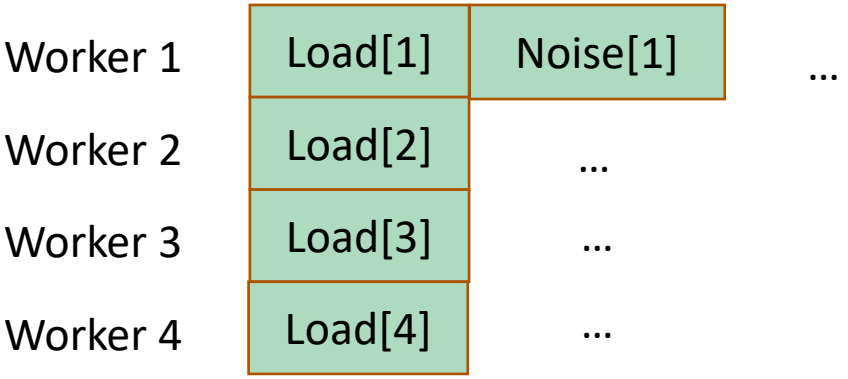
What is a (processing) pipeline?

- Series of data processing steps, where the output of one step is the input of the next
- Example: edge detection on a photograph



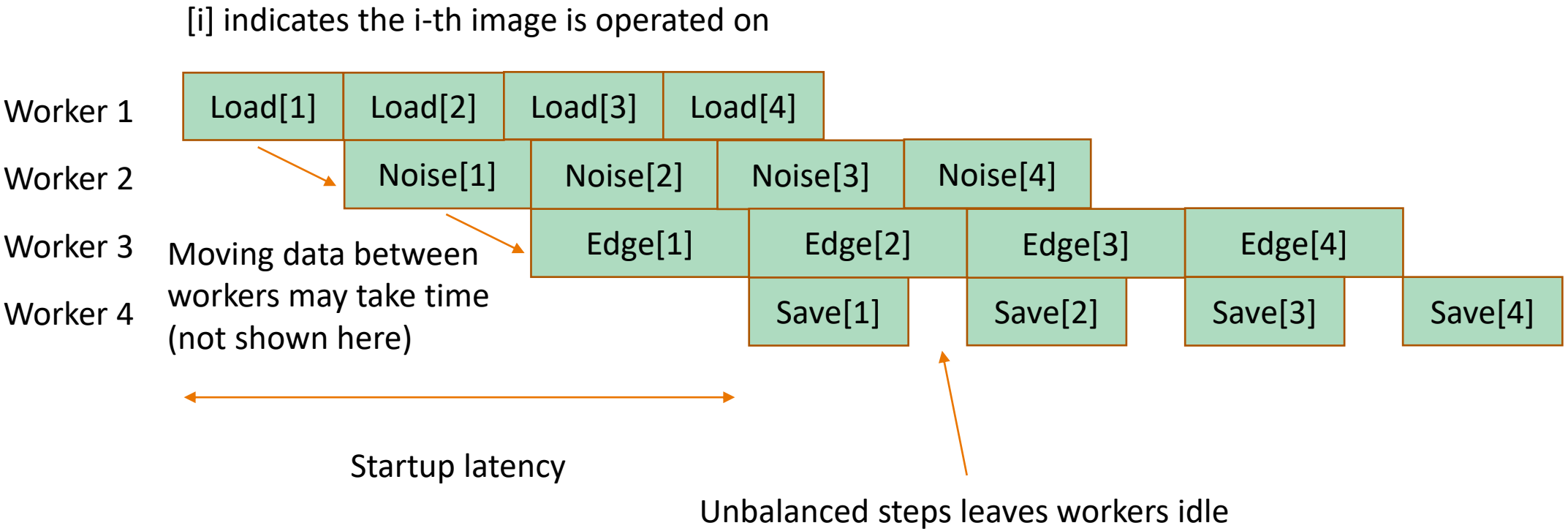
If we go data parallel…

- All workers stress e.g. I/O simultaneously
- Problem if e.g. Load[2] would depend on Load[1] (not likely for steps in this example, but may happen for other operations)



What is pipeline parallelism?

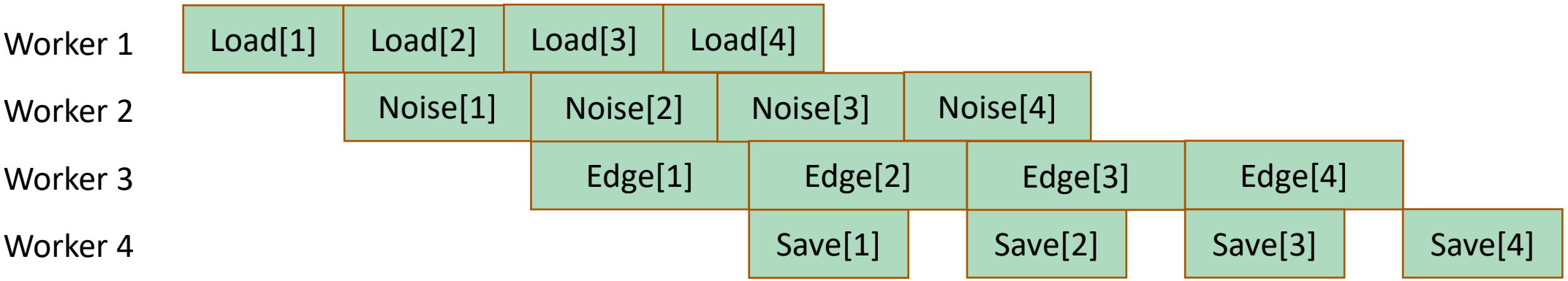
- Balanced resource usage (no simultaneous ‘Load’ operations)
- No problem if Operation[2] depends on Operation[1]



Pipeline parallelism works well when…

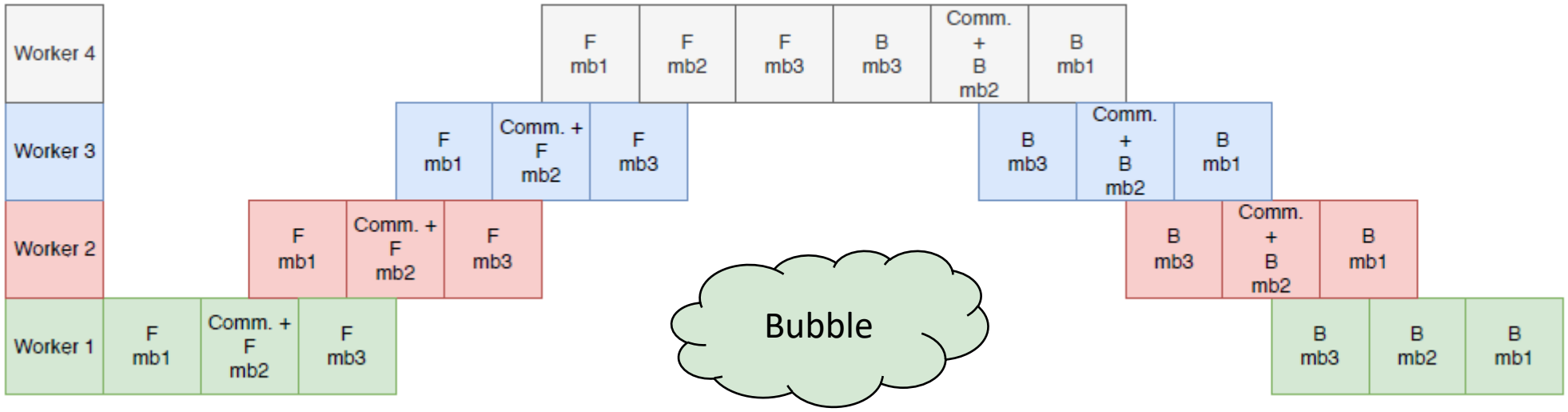
There is as little ‘white’ space as possible in the schematic below. I.e.

- # data elements >> # Workers
- workloads are balanced
- amount of data movement is limited



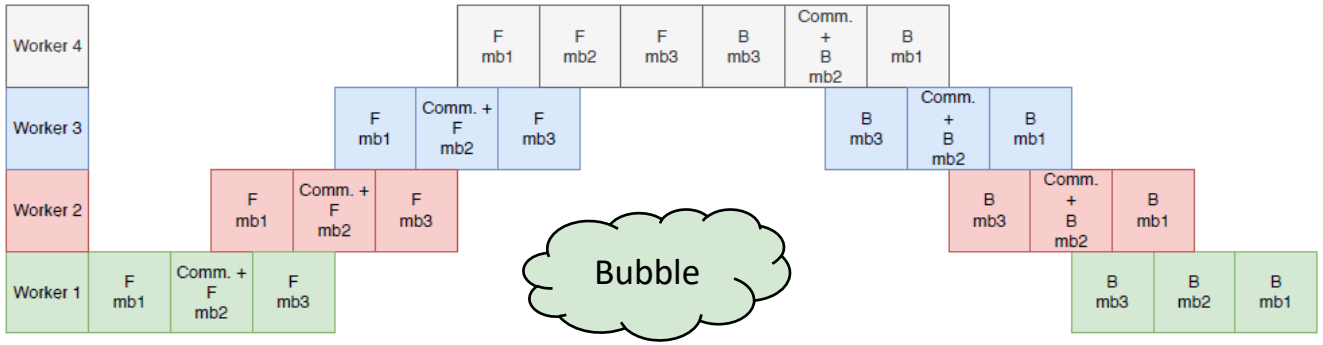
Pipeline parallelism for Deep Learning: GPipe

- Model is partitioned (each color = one model part = one worker)
- Mini-batches are split into micro-batches



Using Gpipe efficiently

- Make sure model partitions are balanced (e.g. using torchgpipe.balance)
- Pick a good number of micro-batches: more microbatches = smaller bubble, but (potentially) less efficient compute (small CUDA kernels have lots of overhead)



Frameworks for pipeline parallelism

- Gpipe (PyTorch, TensorFlow) https://arxiv.org/pdf/1811.06965.pdf
- PipeDream (PyTorch) https://arxiv.org/abs/1806.03377
- PipeMare (PyTorch) https://arxiv.org/abs/1910.05124
- …

- Differences in how the pipeline parallelism is implemented
  - Gpipe recomputes forward activations on the backward pass, minimizing memory footprint
  - PipeDream adds parallelism between minibatches, filling the ‘bubble’ better (at the cost of asynchronous weight updates)
- All are relatively ‘immature’: practical performance not always as good as theoretical promise

Pipeline parallelism for Deep Learning

Pro's and con's:

- Model is partitioned => less memory usage per worker (just like model parallel)
- Pipeline parallelism gives some speedup (but not as much as data parallel)

Summary:

- Only go pipeline parallel for large models that don't fit memory (otherwise: data parallel)

Hands-on: Gpipe Mnist tutorial

In this exercise, you will change a regular PyTorch code that trains ResNet-50 on MNIST to use GPipe.

Typically, the following steps are needed to make existing PyTorch code use Gpipe:

- Change model to nn.Sequential (if not already in that form, not needed in this exercise)
- Model no longer needs to be moved to GPU (i.e. model.cuda() or model.to(device) can be removed) since Gpipe does it for you
- Determine how to split the model (manually, or by automatic balancing https://torchgpipe.readthedocs.io/en/stable/guide.html#automatic-balancing)
- Wrap sequential model as a torchgpipe.Gpipe model (see https://torchgpipe.readthedocs.io/en/stable/guide.html#applying-gpipe)
- Input to first device, target to last device (see https://torchgpipe.readthedocs.io/en/stable/guide.html#applying-gpipe)

Hands-on: Gpipe Mnist tutorial (15-20 min)

- Open a Terminal (New -> Terminal)
- Go to the folder **JHL_notebooks/Gpipe**
- Submit the job **job_pytorch.sh** (sbatch job_pytorch.sh). This is will run **mnist_pytorch.py** and serve as a reference
- Start from **mnist_gpipe_exercise.py**. Try to change it so it will use GPipe. Don't peak in **mnist_gpipe_answer.py** unless you're really stuck or done!
- Submit the job using **job_gpipe_exercise.sh**
- N.B. note that **mnist_pytorch.py** and **mnist_gpipe_exercise.py** use a different default batch size

The output will look something like this:

train | 1/1 epoch (4%) | 692.340 samples/sec (estimated)

GPU 0: allocated 0.271 GB, reserved 8.072 GB

This shows how fast the training is(692 img/sec), how much memory is used by the model (0.27 GB) and how much by the activations (8.1 GB).

What do you notice when comparing the speed & memory usage for GPipe and PyTorch?

Hands-on: Gpipe Mnist tutorial (15-20 min)

What do you notice when comparing the speed & memory usage for GPipe and pure PyTorch?

PyTorch
train | 1/1 epoch (99%) | 718.463 samples/sec
GPU 0: allocated 0.270 GB, reserved 8.074 GB

GPipe
train | 1/1 epoch (99%) | 961.705 samples/sec
GPU 0: allocated 0.015 GB, reserved 3.221 GB
GPU 1: allocated 0.256 GB, reserved 1.596 GB

- Speed is 33% higher for GPipe (but at twice the resources, and only if we use a batch size of 512 or higher, compared to 256 for pure PyTorch)
- Memory consumption for the model is split in GPipe
- Most of the memory consumption of *this* model is in activations, not in model parameters
- GPipe manages to have similar total memory consumption at 4 (!) times the batch size. It is very memory efficient, partly because it recomputes the activations during the backwards pass.
- Maximum memory consumption on a individual GPUs is lower => you can run larger models
- GPU utilization quite low => practical performance not as good as theoretical promise
- Speed may strongly depend on your model! Models with many weights but few activations *probably* do better.
- **GPipe is very memory efficient; scaling performance is quite poor (but better than model parallelism!).**

Hands-on: Gpipe Mnist tutorial part 2 (5-10 min)

Using **mnist_gpipe_answer.py**, experiment with

- Automatic balancing by time v.s. by memory (set --balancy_by=time or memory)
- Number of microbatches (--num_microbatches)
- Batch size (--batchsize, how high can you go without Gpipe, and with?)

You can change these arguments in **job_gpipe_answer.sh**

What do you see if you chance automatic balancing, is it reflected in speed & memory consumption?