https://hao-ai-lab.github.io/cse234-w25/

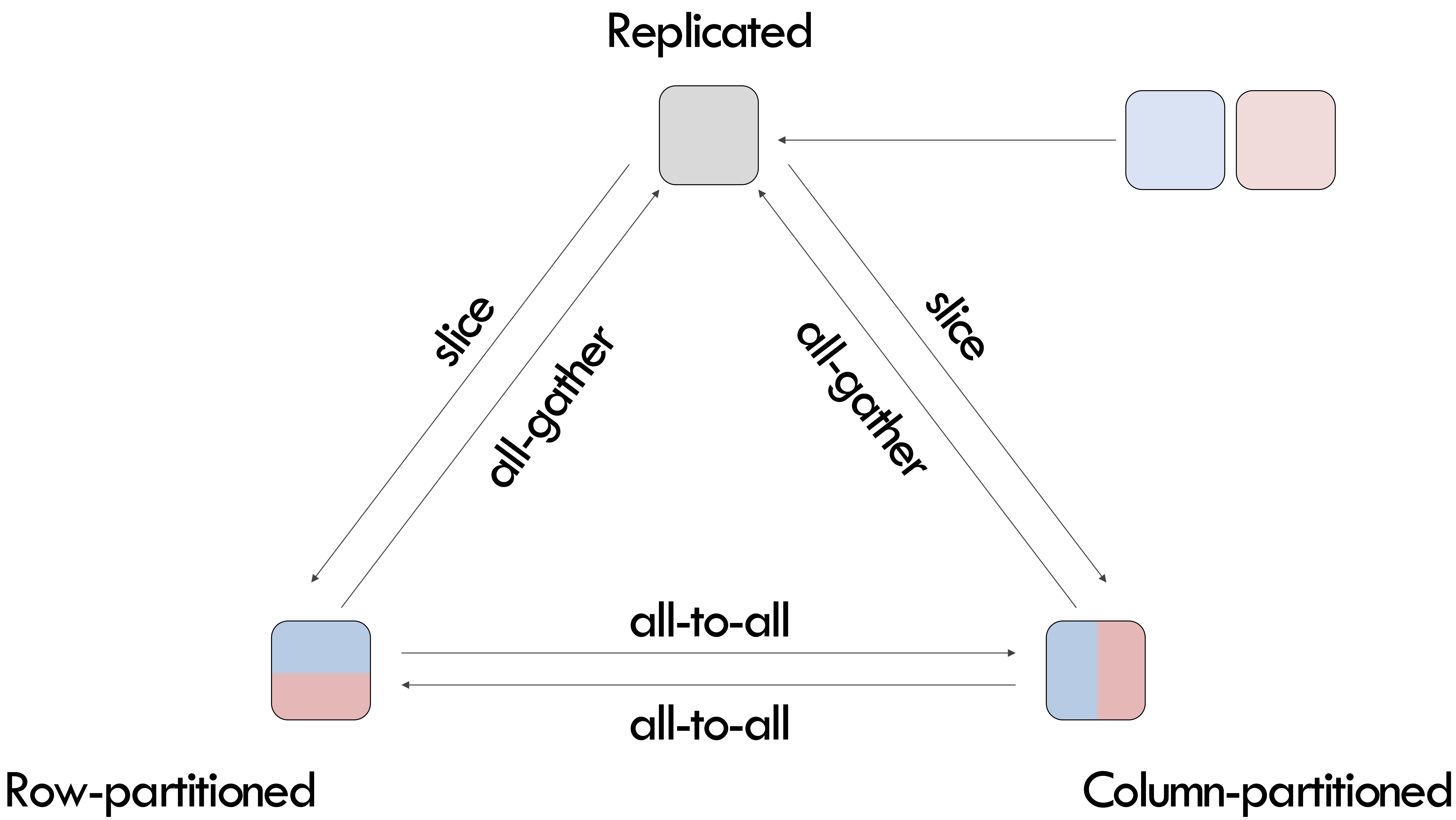
CSE 234: Data Systems for Machine Learning
Winter 2025

LLMSys

Optimizations and Parallelization

MLSys Basics

"Re-partition" in ML Parallelism yields collectives



Where We Are

- Motivation
- History
- Parallelism Overview
- **Data parallelism**
- Model parallelism
  - Inter and intra-op parallelism
- Auto-parallelization

Why Data Parallelism First

2012

2016

Focus: Data parallelism with **Parameter Server**

Asynchrony: update every N iters instead of 1

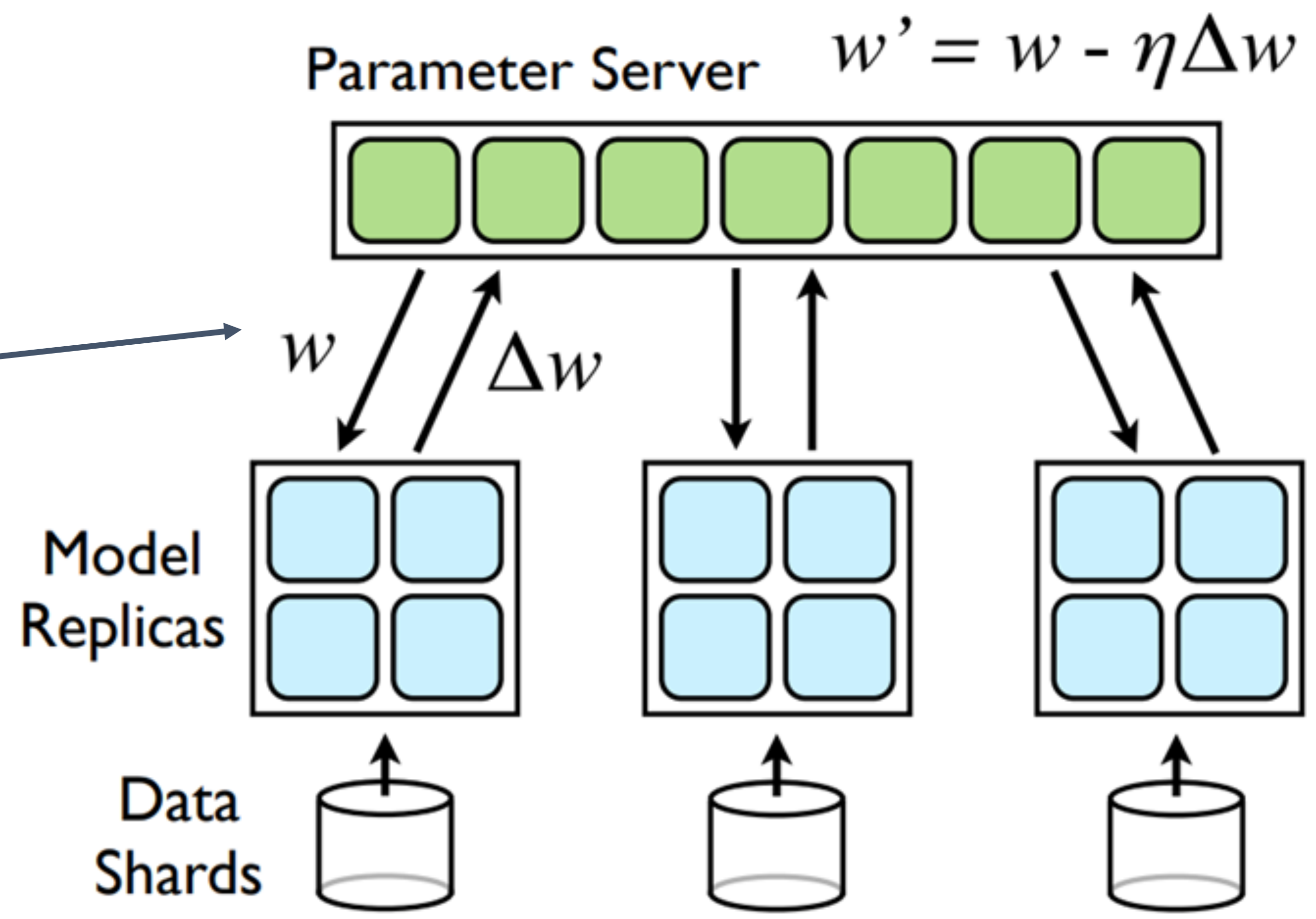


Figure from DistBelief [Dean et al., NeurIPS 2012]

Various implementations of parameter servers

- DistBelief [Dean et al., NeurIPS 2012]
- Parameter server [Li et al., NeurIPS 2012], [Li et al., OSDI 2014]
- Bosen [Wei et al., SoCC 2015]
- GeePS [Cui et al., Eurosys 2016], Poseidon [Zhang et al., ATC 2017]

Why Data Parallelism First

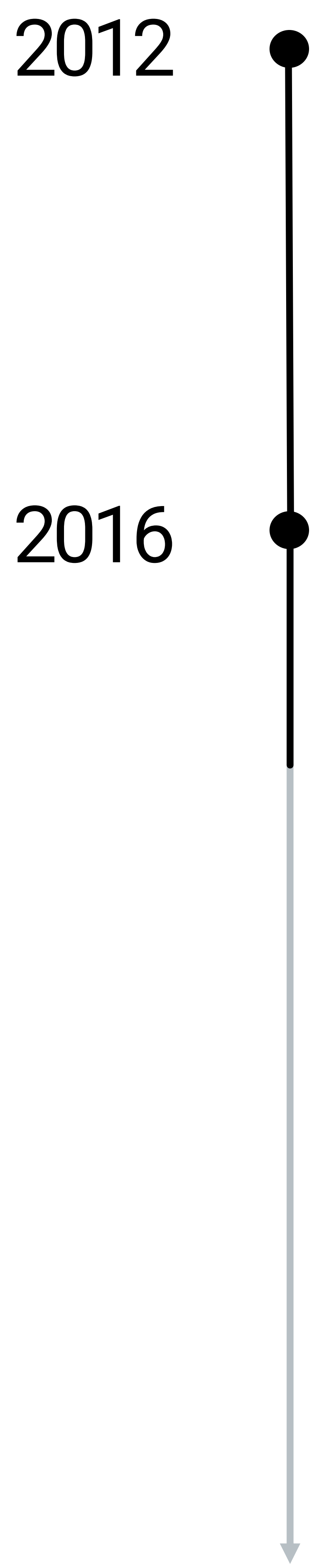
2012

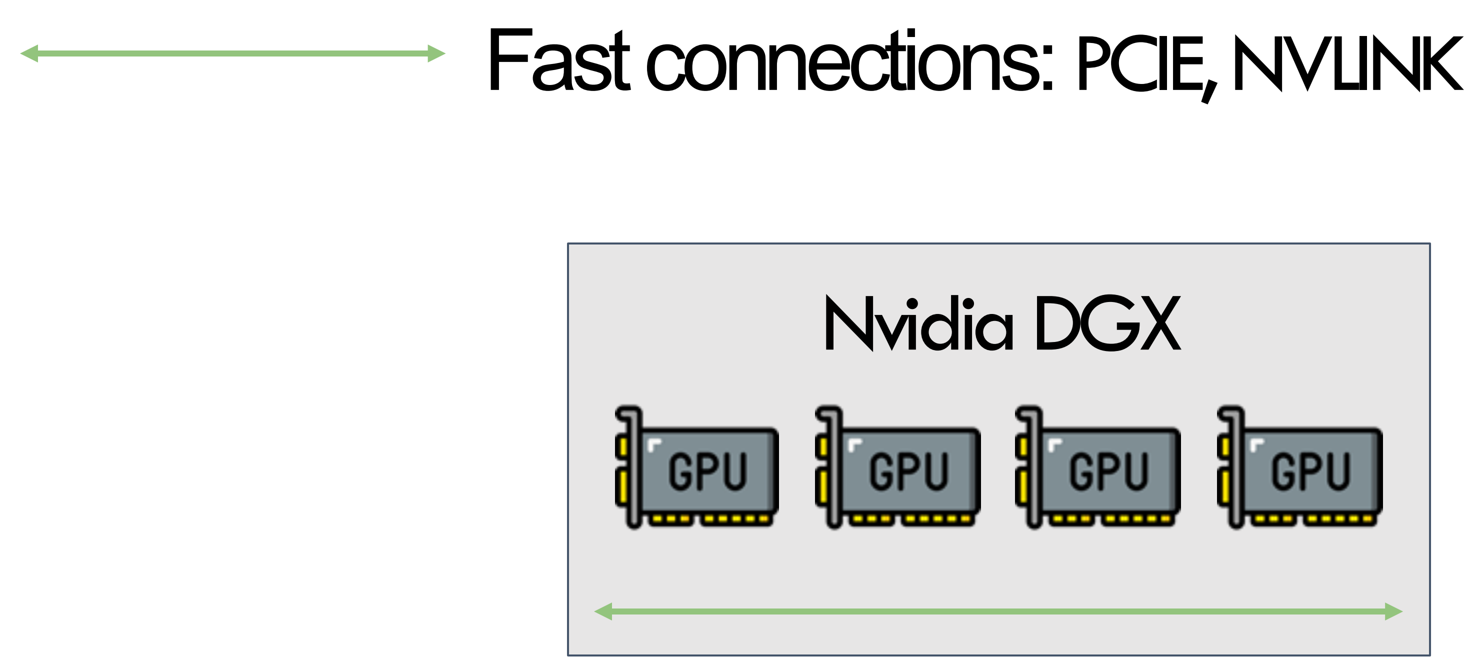
2016

```
import torch.nn.parallel as dist
from torch.nn.parallel import DistributedDataParallel as DDP

dist.init_process_group("nccl", rank=rank, world_size=world_size)
ddp_model = DDP(Model(), device_ids=[rank])

for batch in data_loader:
    loss = train_step(ddp_model, batch)
```

Sergeev et al., "Horovod: fast and easy distributed deep learning in TensorFlow". *Preprint 2018.*
Li et al., "PyTorch Distributed: Experiences on Accelerating Data Parallel Training". VLDB 2020.

Why Data Parallelism First





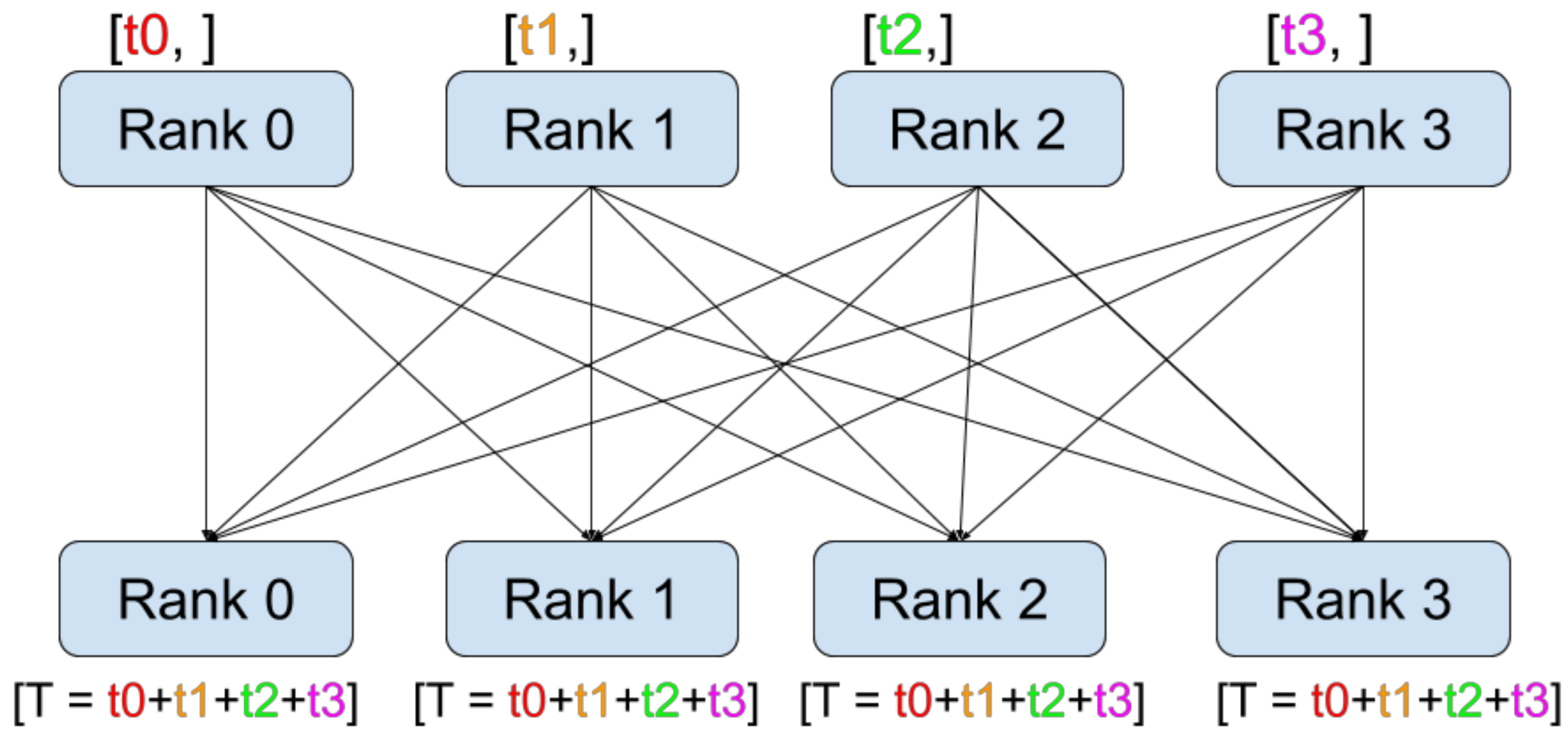
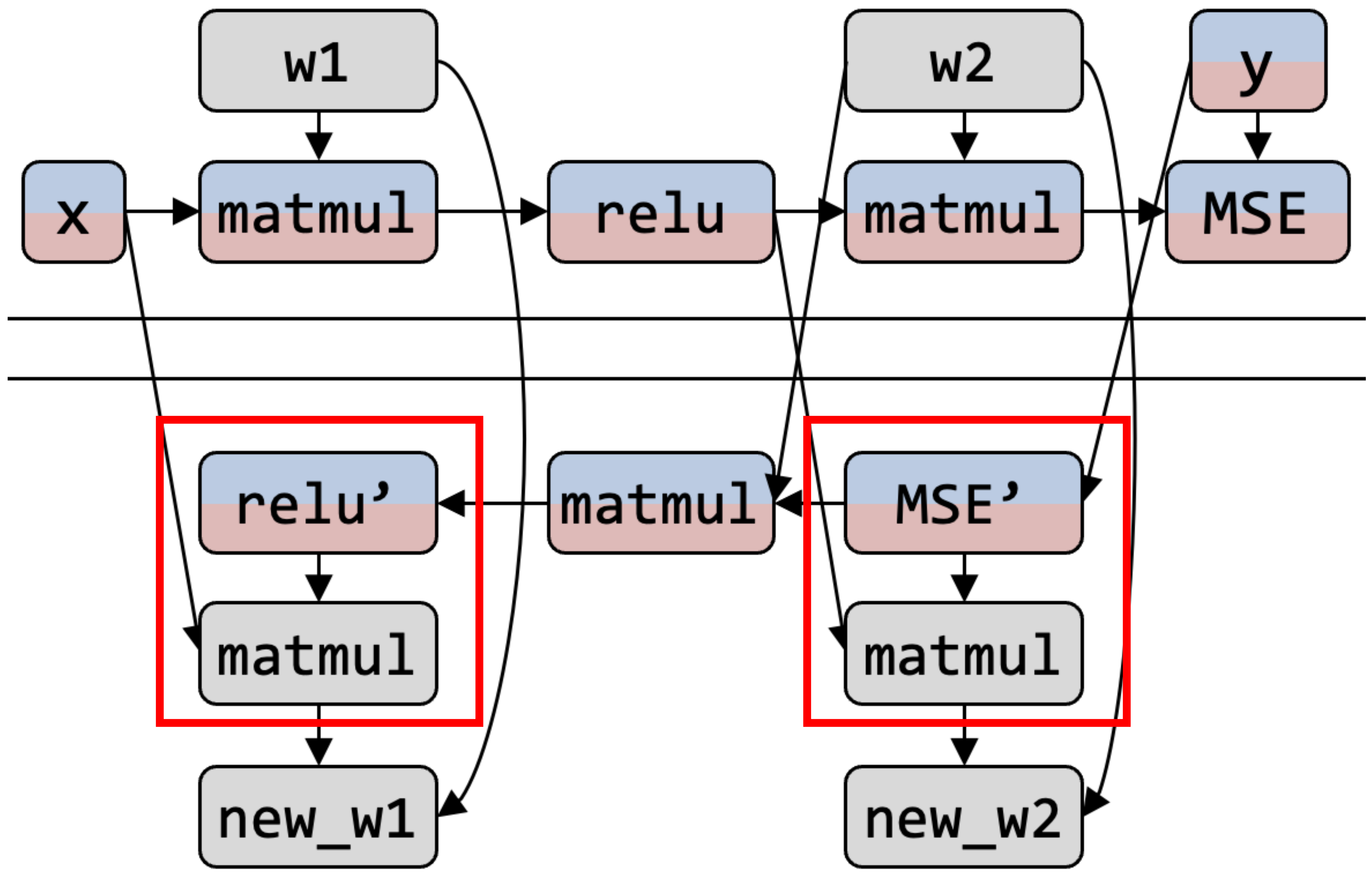


Figure from PyTorch Tutorials

Data Parallelism



How to implement this communication?

Two Solutions

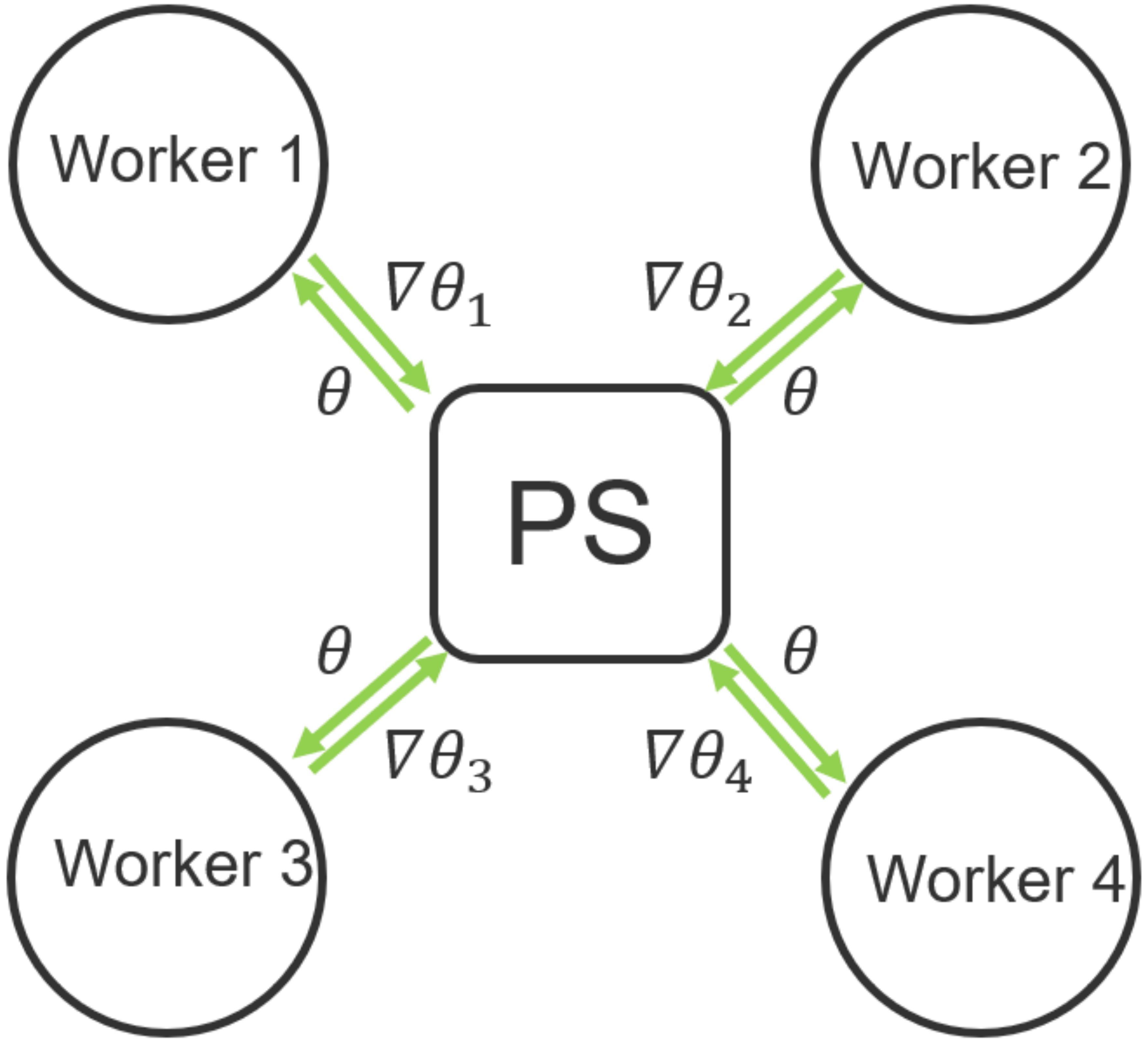
- Parameter Server
- AllReduce
- Key assumption:
  - The model can fit into an (GPU) worker memory hence we can create many replica

Parameter Server Assumption

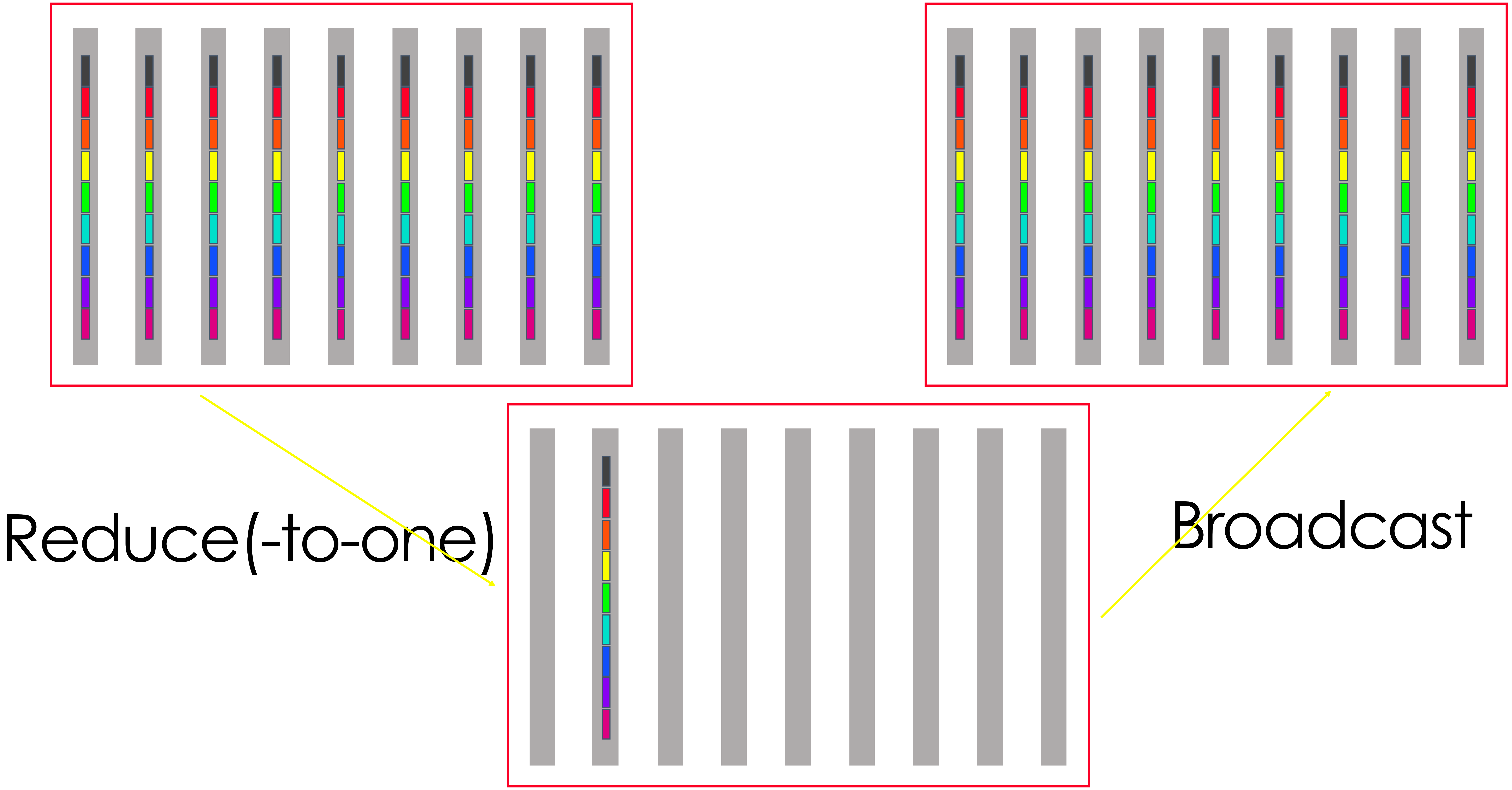
$$\theta^{(t+1)} = \theta^{(t)} + \varepsilon \sum_{p=1}^{P} \nabla_{\mathcal{L}}(\theta^{(t)}, D_p^{(t)})$$

- Very heavy communication per iteration
- Compute : communication = 1:10 in the era of 2012

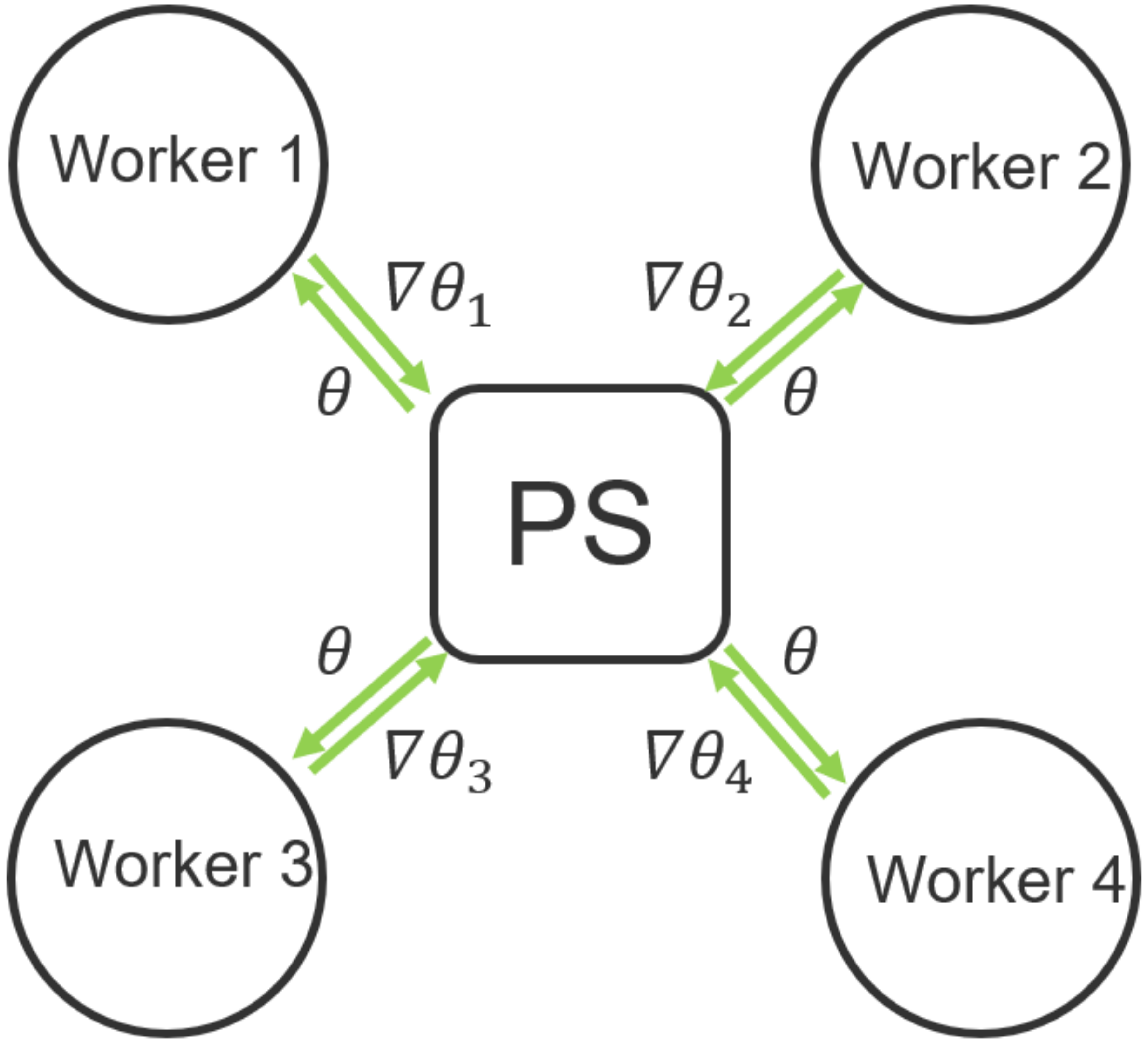
Parameter Server Naturally emerges



AllReduce = reduce + broadcast



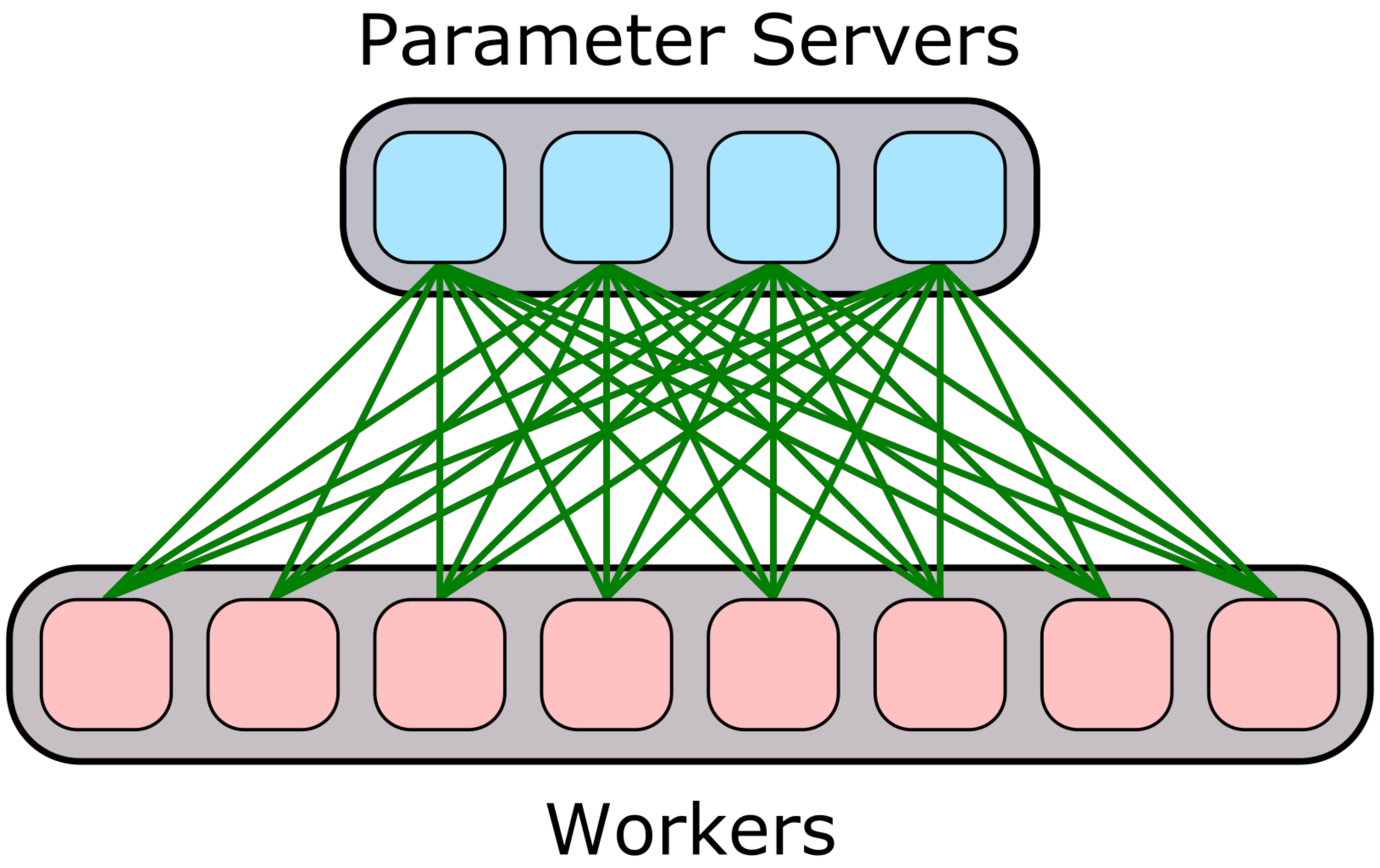
Parameter Server Naturally emerges



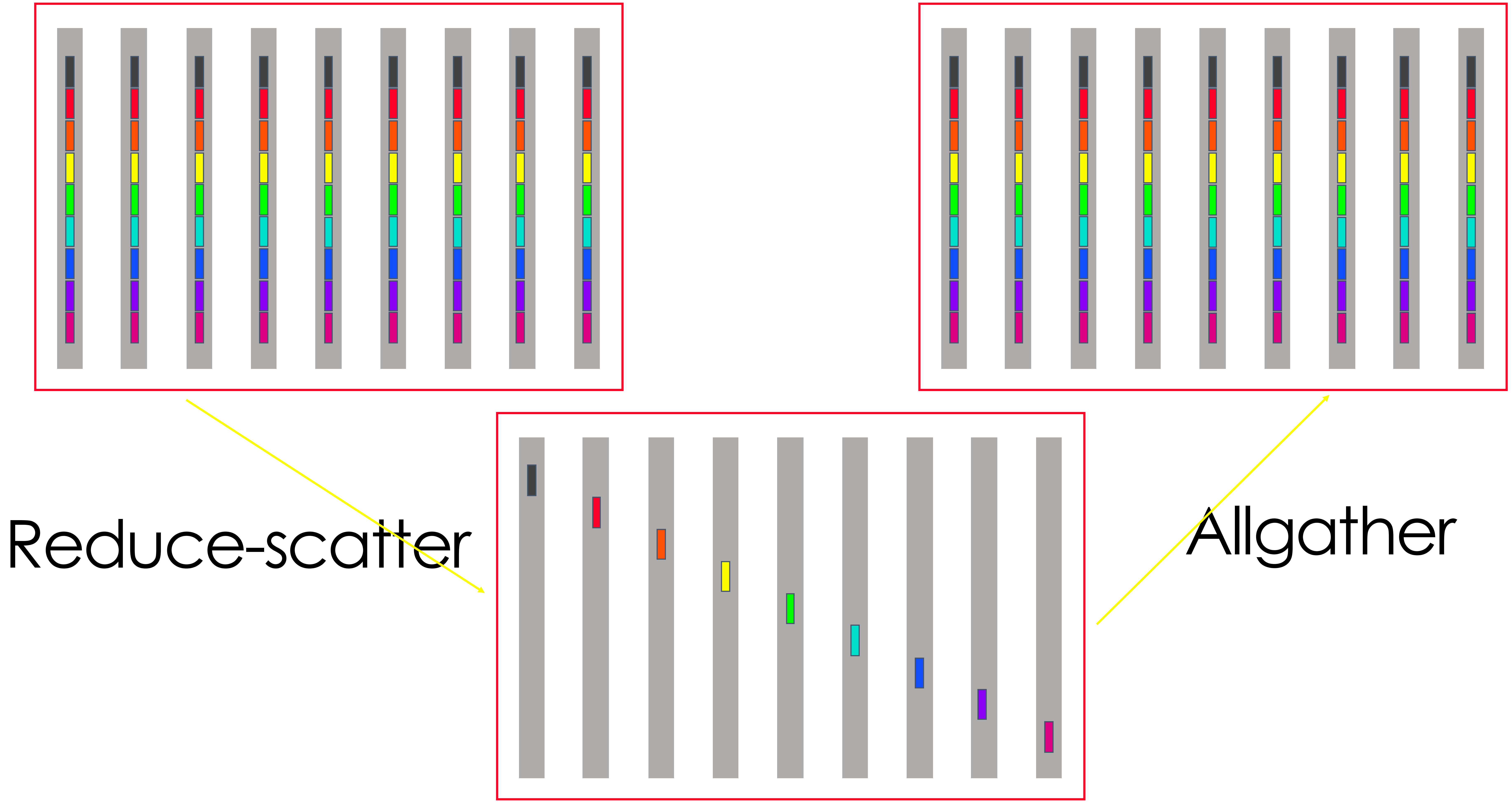
Problems:
Server bottleneck!

Parameter Server Implementation

- Sharded parameter server: sharded KV stores
  - Avoid communication bottleneck
  - Redundancy across different PS shards

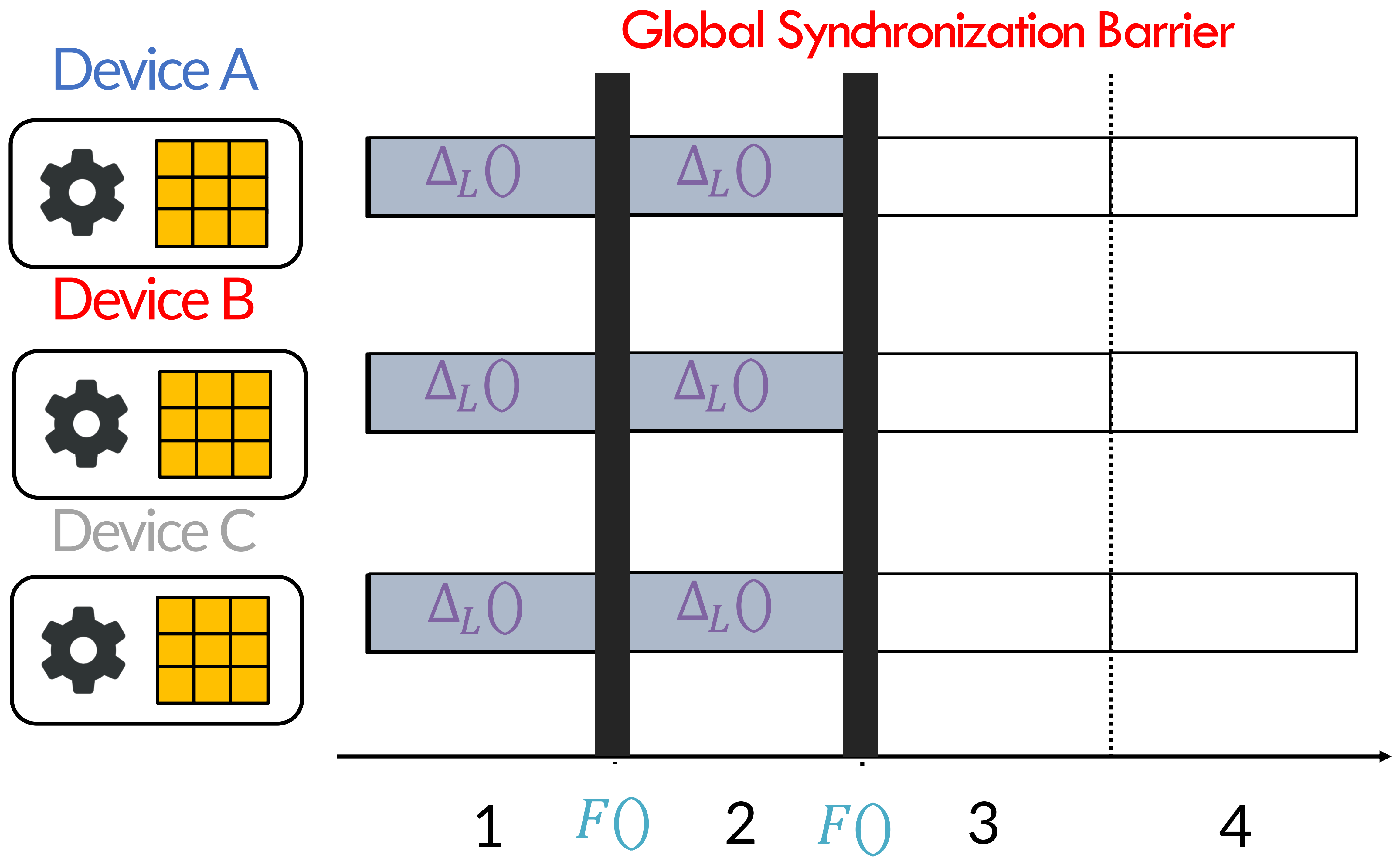


When servers nodes == worker nodes



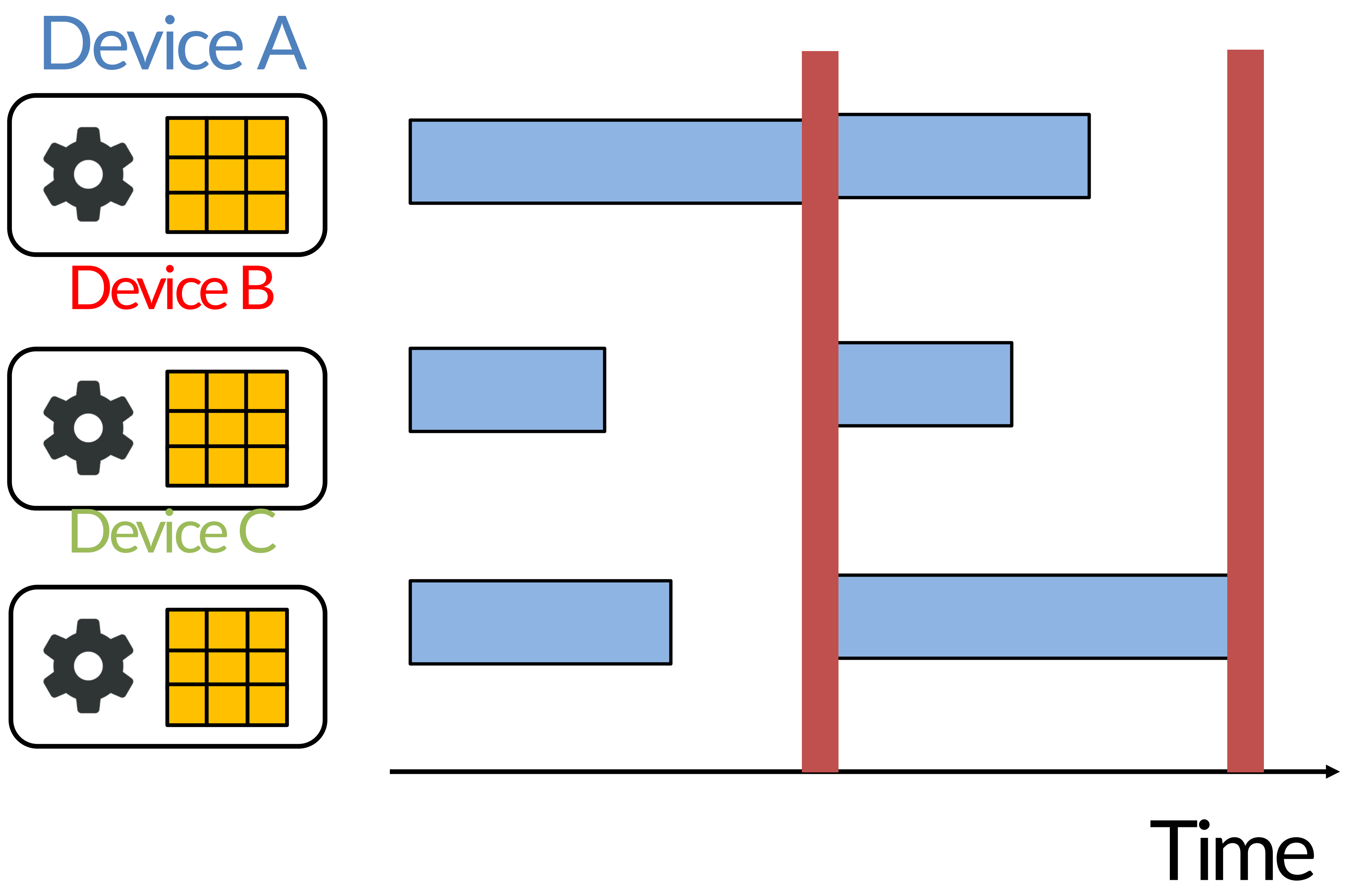
Consistency

$$\theta^{(t+1)} = \theta^{(t)} + \varepsilon \sum_{p=1}^{P} \nabla_{\mathcal{L}}(\theta^{(t)}, D_p^{(t)})$$



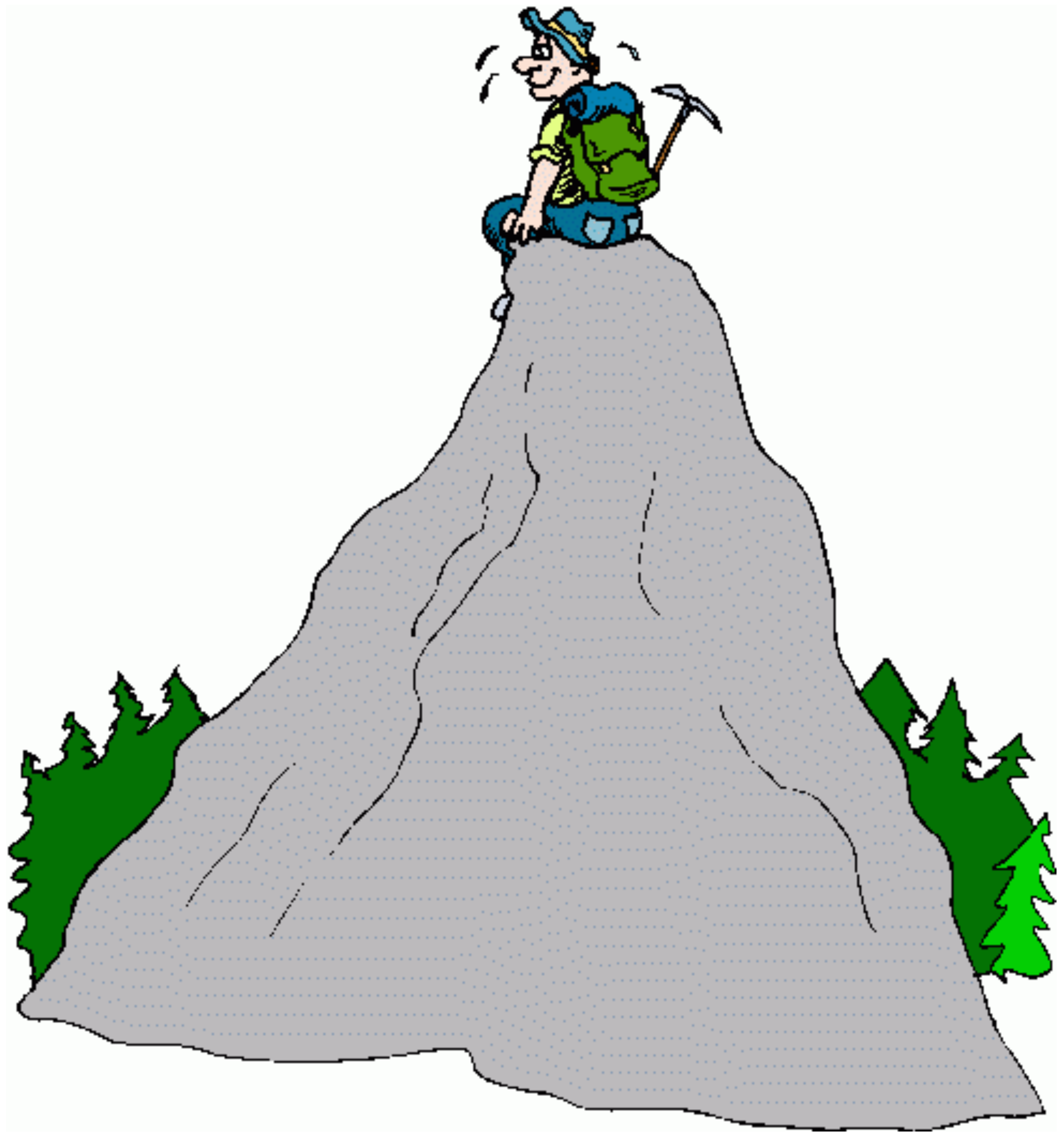
BSP's Weakness: Stragglers

- **BSP suffers from stragglers**
  - Slow devices (stragglers) force all devices to wait
  - More devices → higher chance of having a straggler

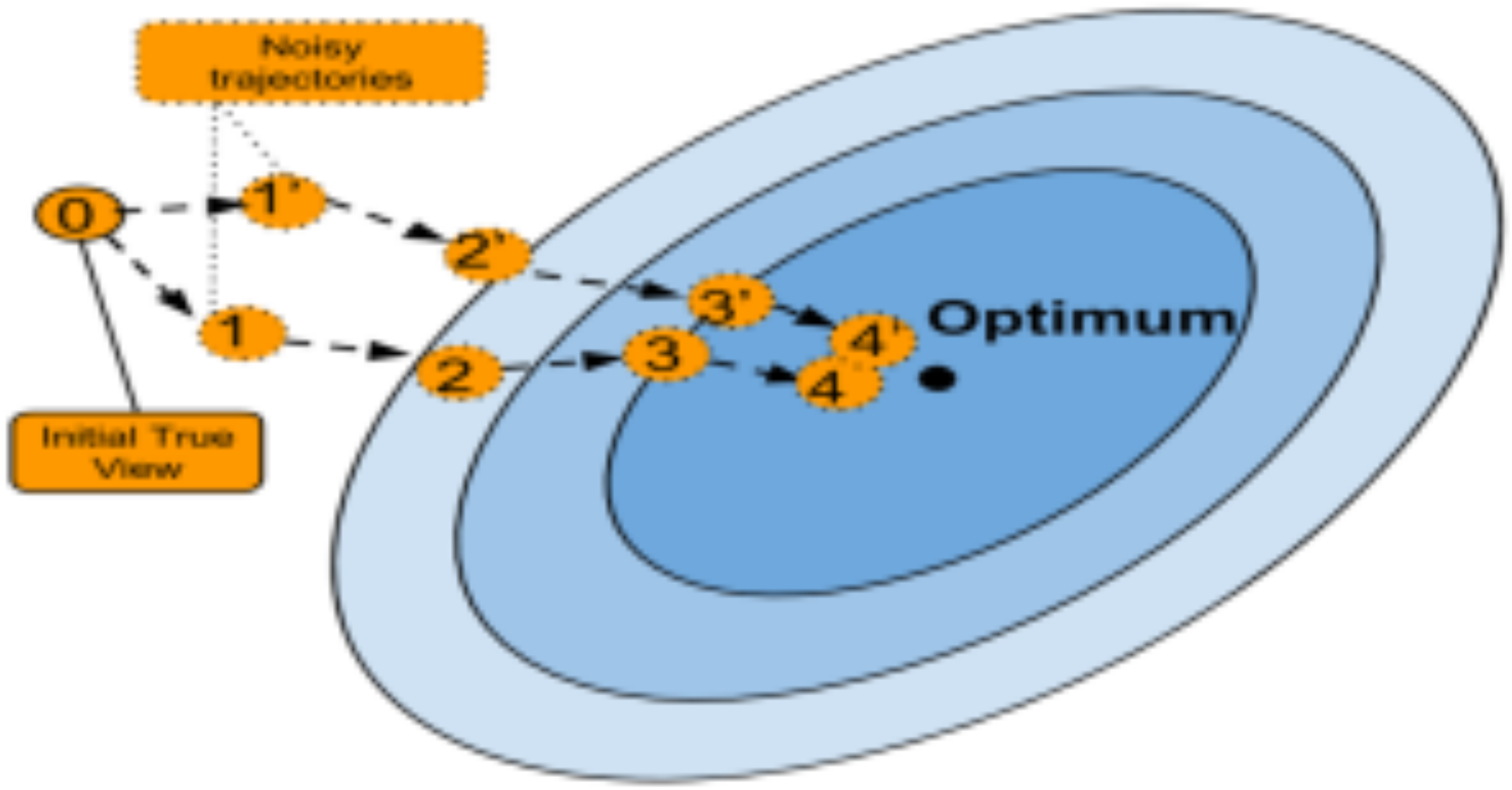


An interesting property of Gradient Descent (ascent)

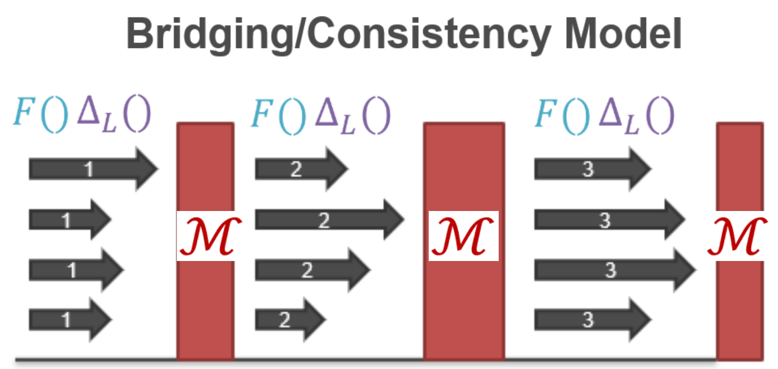
$$\theta^{(t+1)} = \theta^{(t)} + \varepsilon \sum_{p=1}^{P} \nabla_{\mathcal{L}}(\theta^{(t)}, D_p^{(t)})$$



Machine Learning is Error-tolerant (under certain conditions)

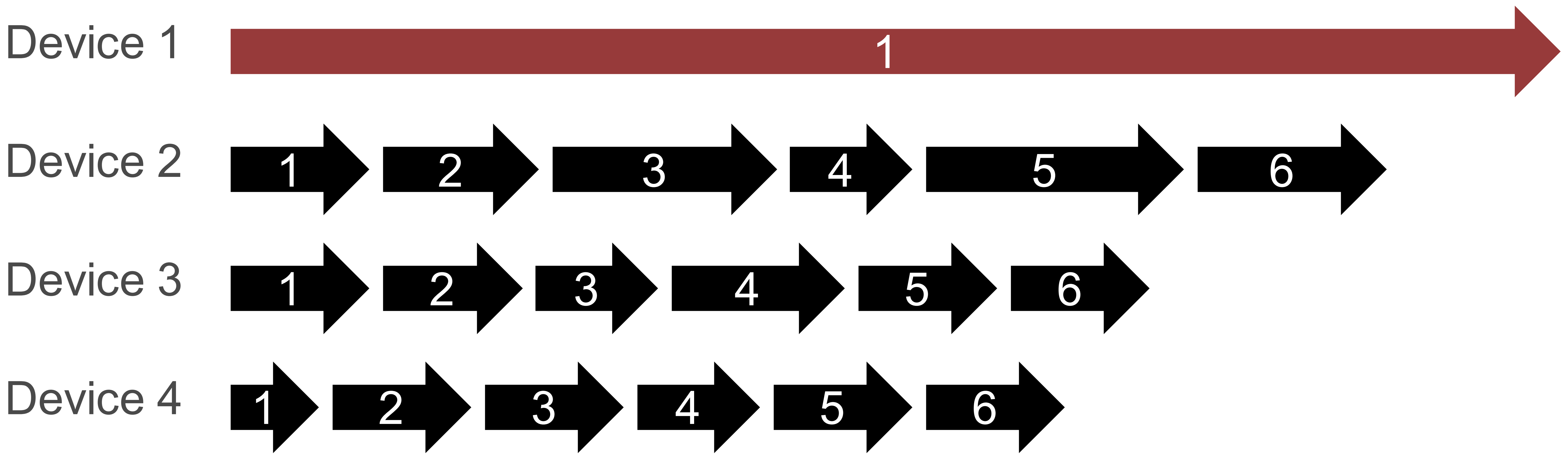


Background: Asynchronous Communication (No Consistency)

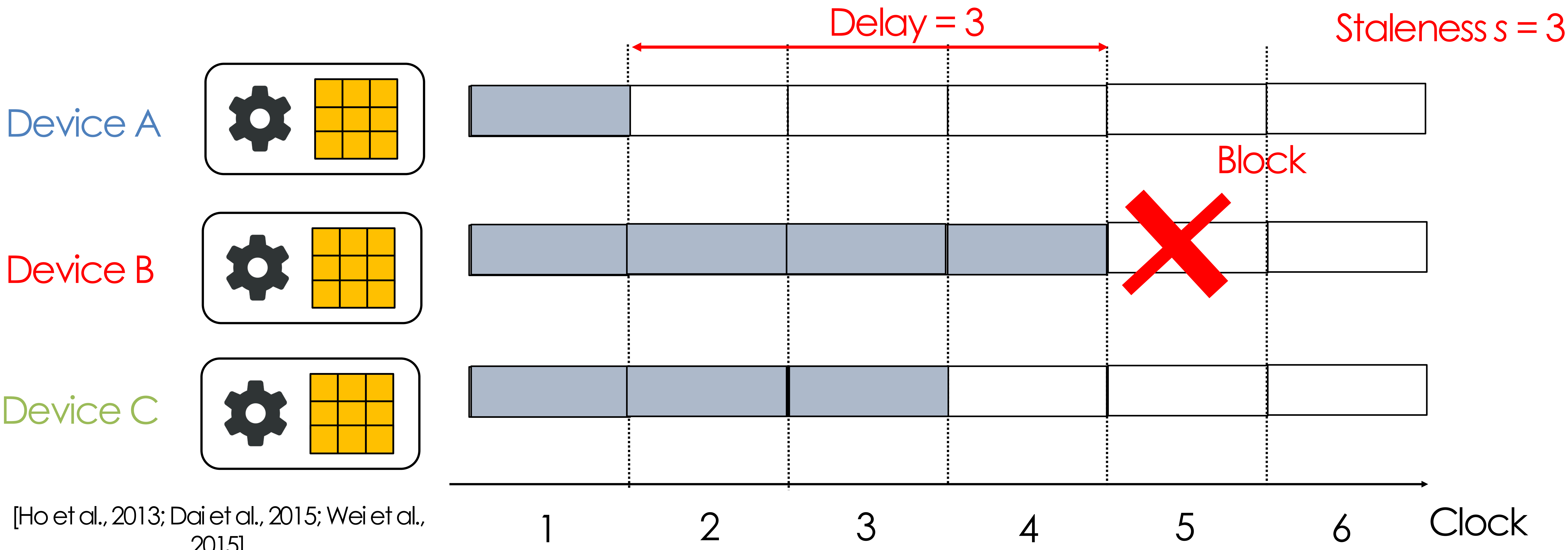


- **Asynchronous (Async):** removes all communication barriers
  - Maximizes computing time
  - Transient stragglers will cause messages to be extremely stale
    - Ex: Device 2 is at $t = 6$, but Device 1 has only sent message for $t = 1$

- **Some Async software:** <u>messages can be applied while computing</u> $F()$, $\Delta_L()$
  - Unpredictable behavior, can hurt statistical efficiency!

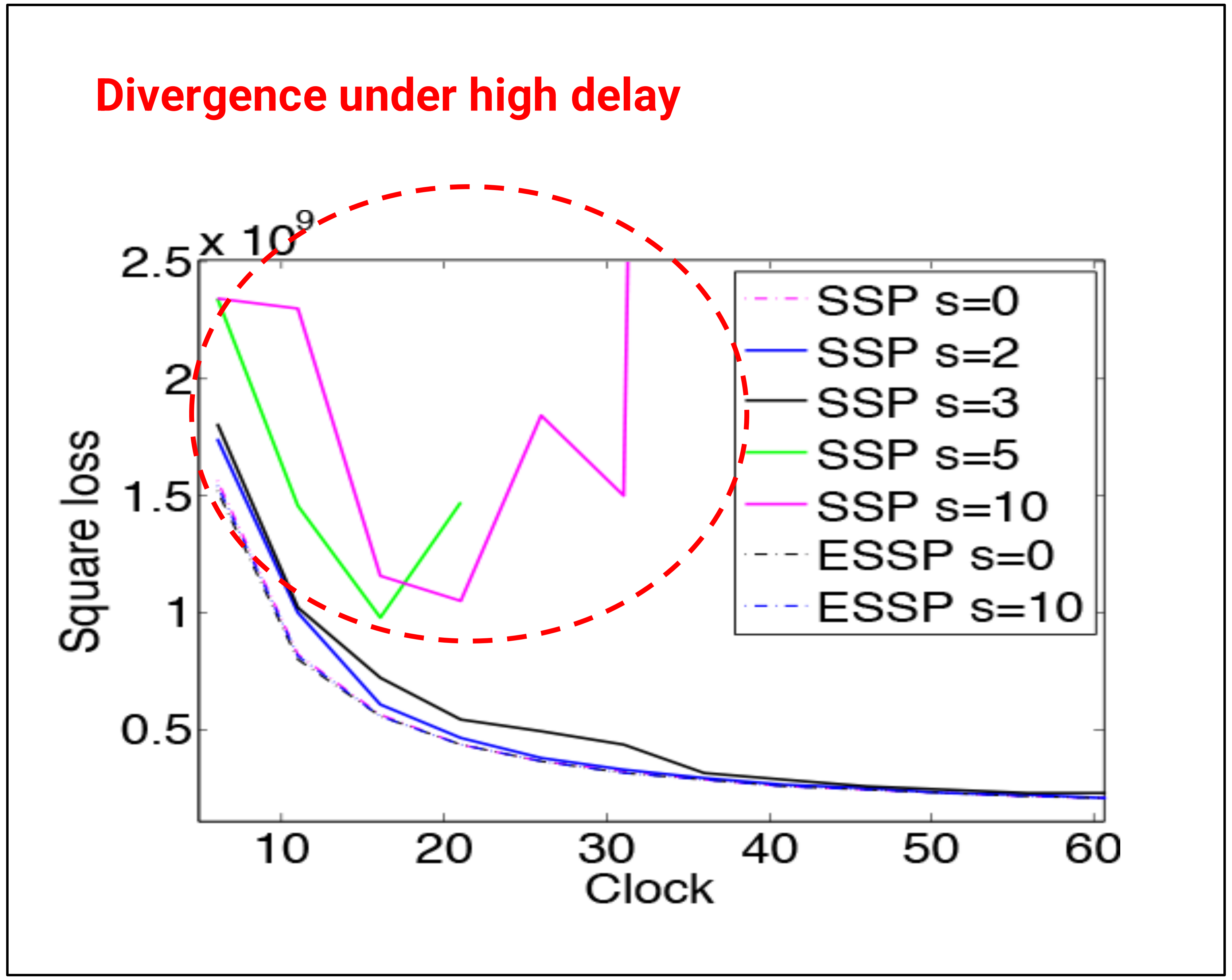


Background: Bounded Consistency

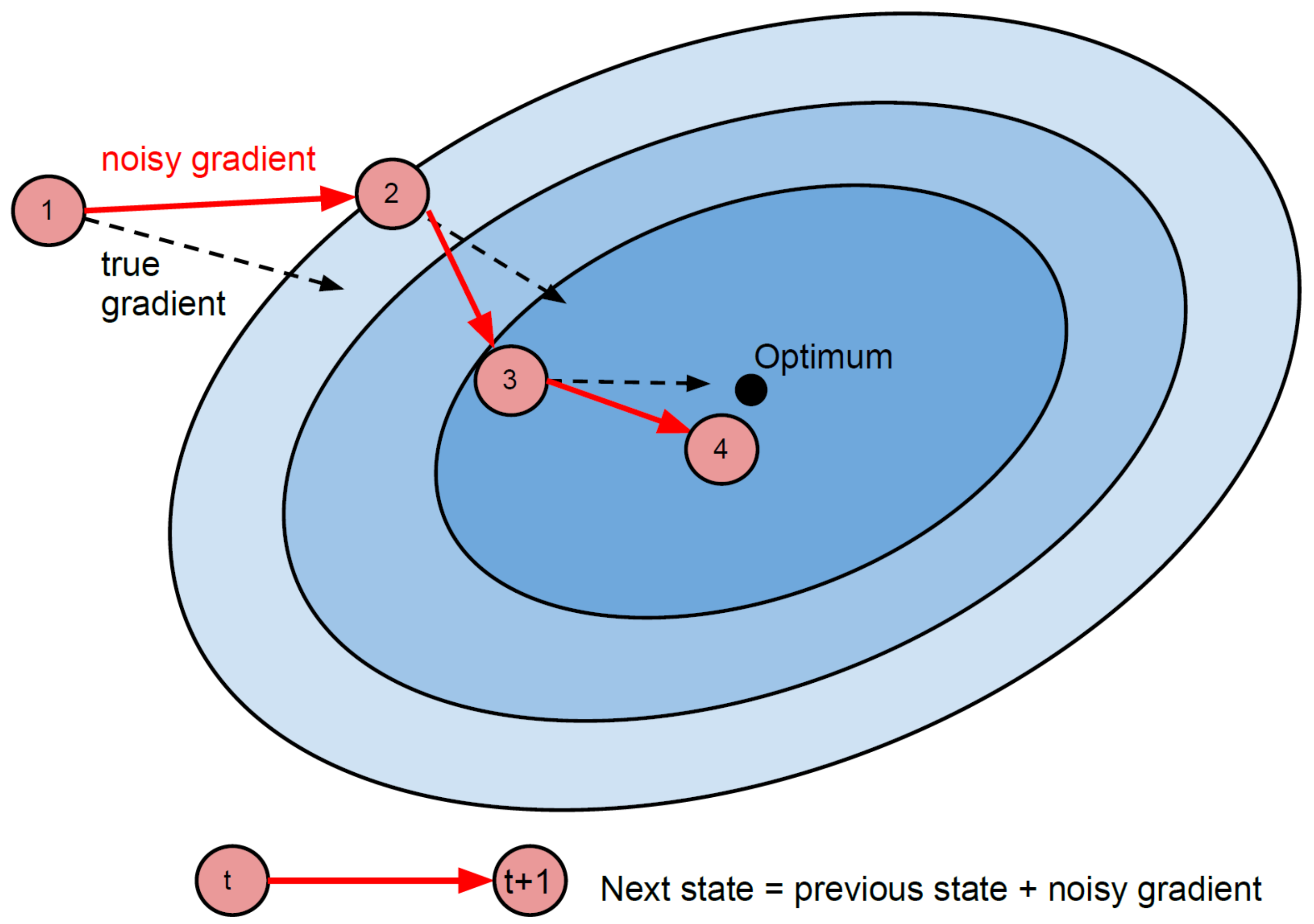


[Ho et al., 2013; Dai et al., 2015; Wei et al., 2015]

Impacts of Consistency/Staleness: Unbounded Staleness

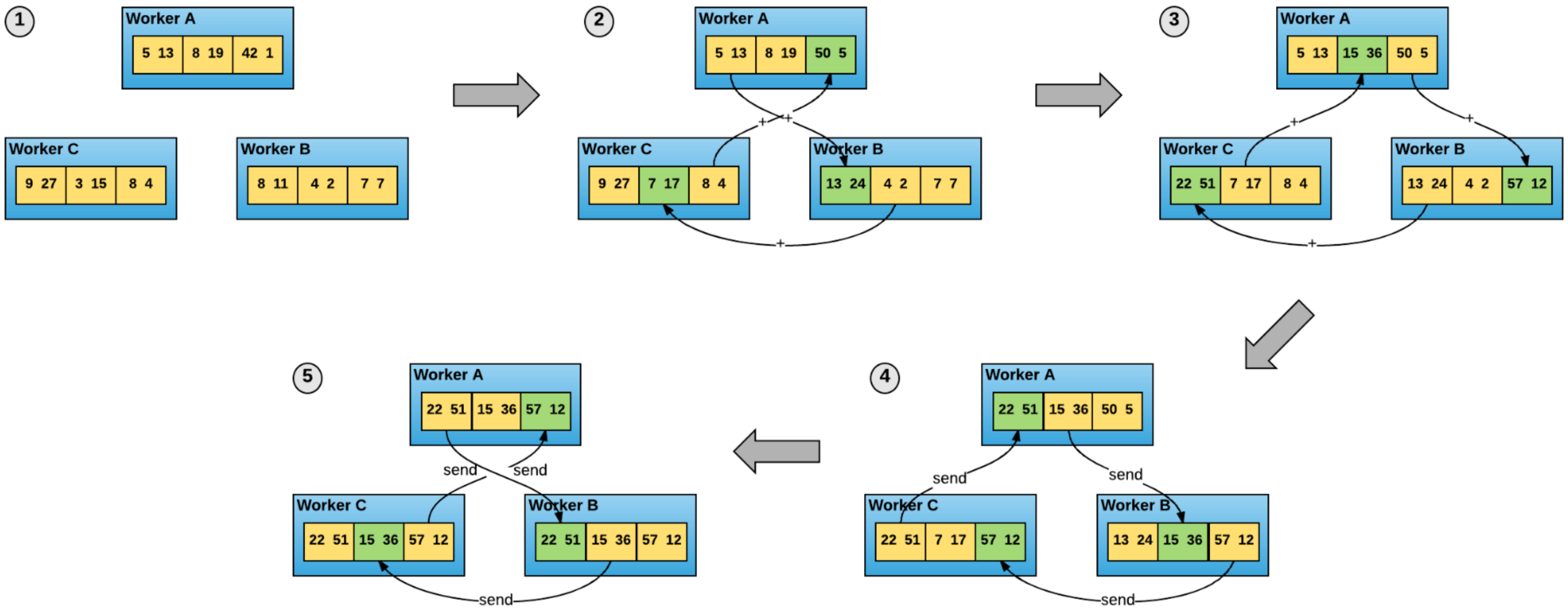


Theory: SSP Expectation Bound



$$R[\mathbf{X}] := \underbrace{\left[\frac{1}{T}\sum_{t=1}^{T} f_t(\tilde{\mathbf{x}}_t)\right] - f(\mathbf{x}^*)}_{\text{Difference between SSP estimate and true optimum}} \leq 4FL\sqrt{\frac{2(s+1)P}{T}}$$

AllReduce



1
Worker A
5 13
8 19
42 1
Worker C
9 27
3 15
8 4
Worker B
8 11
4 2
7 7
2
Worker A
5 13
8 19
50 5
Worker C
9 27
7 17
8 4
Worker B
13 24
4 2
7 7
3
Worker A
5 13
15 36
50 5
Worker C
22 51
7 17
8 4
Worker B
13 24
4 2
57 12
4
Worker A
22 51
15 36
50 5
send
send
Worker C
22 51
7 17
57 12
Worker B
13 24
15 36
57 12
send
5
Worker A
22 51
15 36
57 12
send
send
Worker C
22 51
15 36
57 12
Worker B
22 51
15 36
57 12
send

Data Parallelism with All-reduce

```
import torch.nn.parallel as dist
from torch.nn.parallel import DistributedDataParallel as DDP

dist.init_process_group("nccl", rank=rank, world_size=world_size)
ddp_model = DDP(Model(), device_ids=[rank])

for batch in data_loader:
   loss = train_step(ddp_model, batch)
```

Sergeev et al., "Horovod: fast and easy distributed deep learning in TensorFlow". *Preprint 2018.*
Li et al., "PyTorch Distributed: Experiences on Accelerating Data Parallel Training". VLDB 2020.

Allreduce

- Initially implemented in Horovod
- Being Optimized by nvidia (hw/sw co-optimizaiton) in NCCL
- Being adopted in PyTorch DDP
- Not Fault tolerant

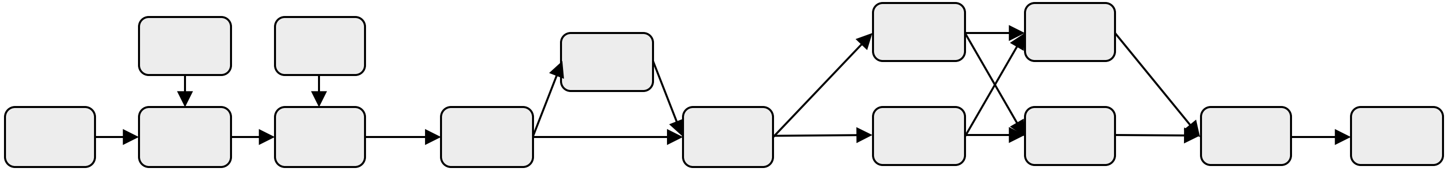
Discussion: Why Allreduce dominates parameter server today?

Where We Are

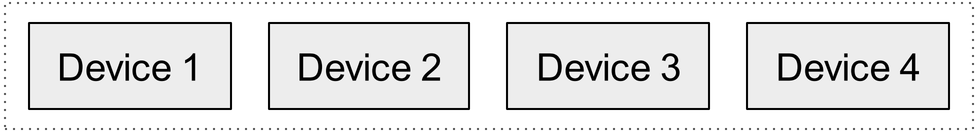
- Motivation
- History
- Parallelism Overview
- Data parallelism
- **Model parallelism**
  - Inter-op parallelism
  - Intra-op parallelism
- Auto-parallelization

Computational Graph (Neural Networks) → Stages

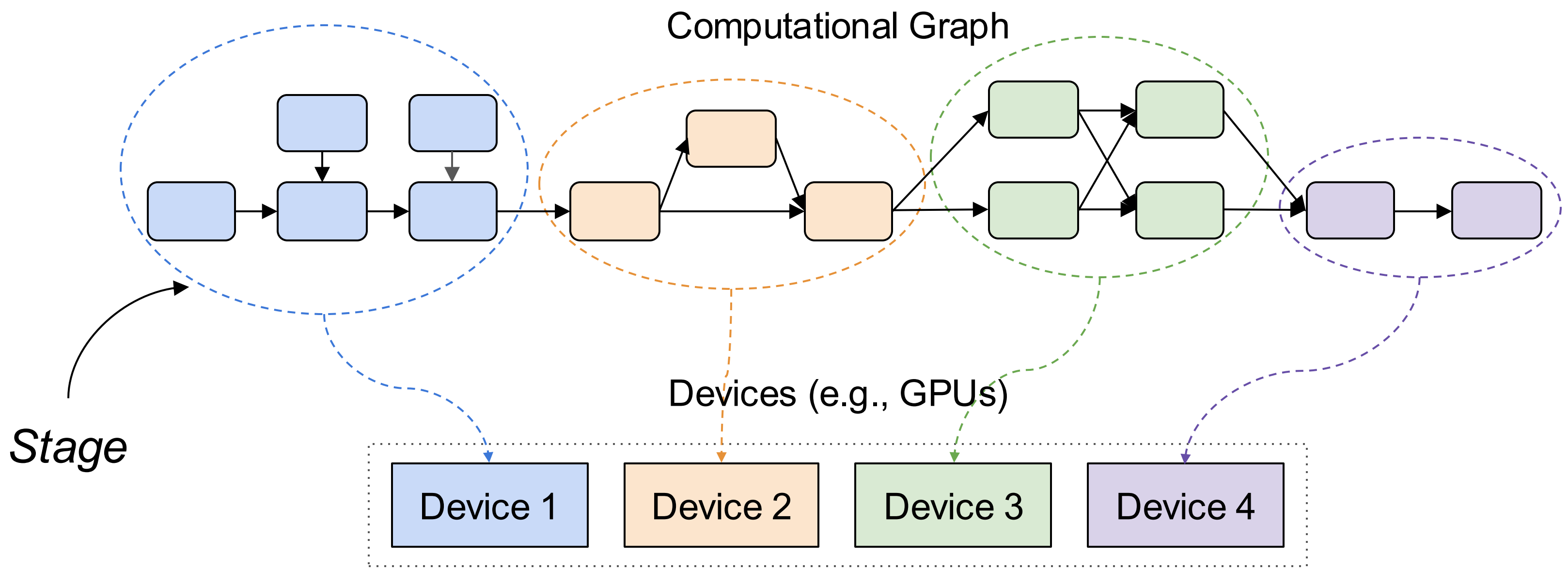
Computational Graph



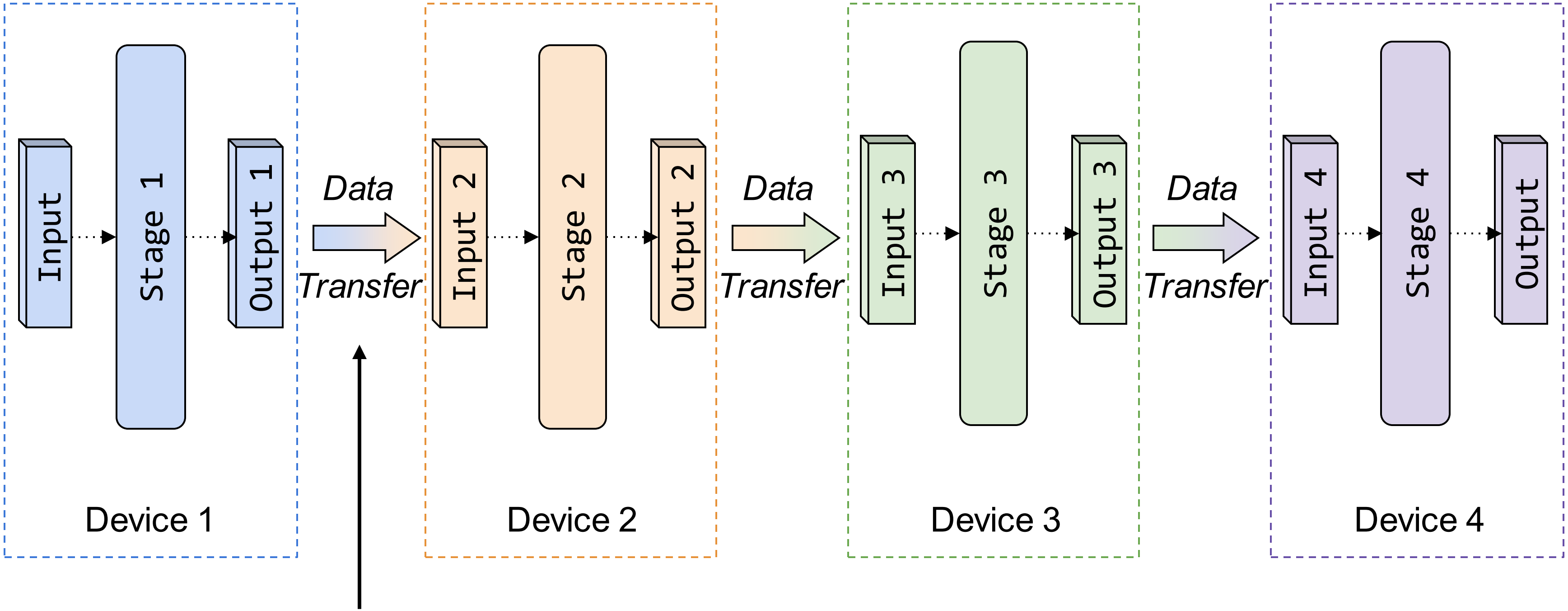
Devices (e.g., GPUs)



Computational Graph (Neural Networks) → Stages

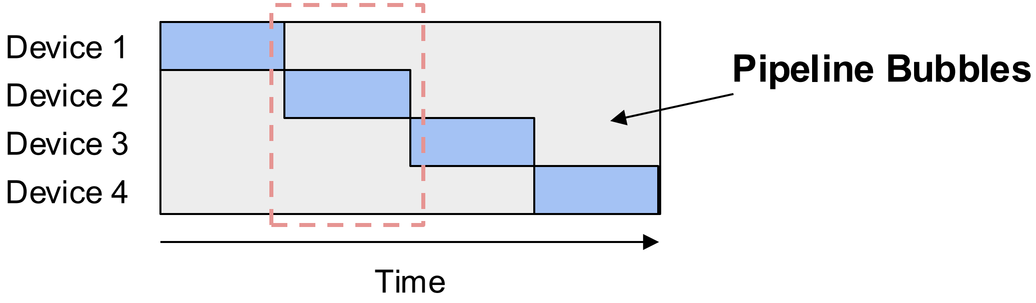


Execution & Data Movement



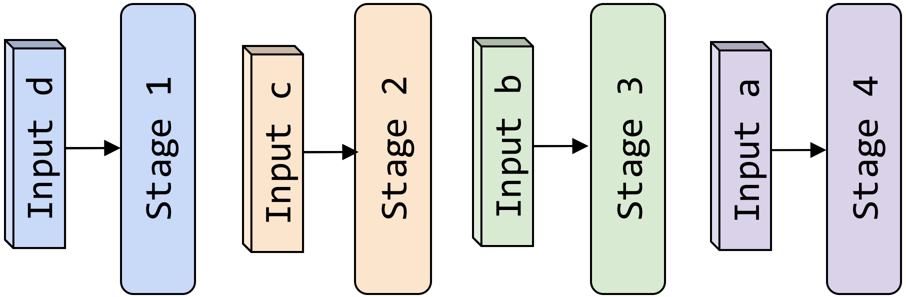
Note: The time spent on data transfer is typically **small,** since we only communicates stage outputs at stage boundaries between two stages.

Timeline: Visualization of Inter-Operator Parallelism

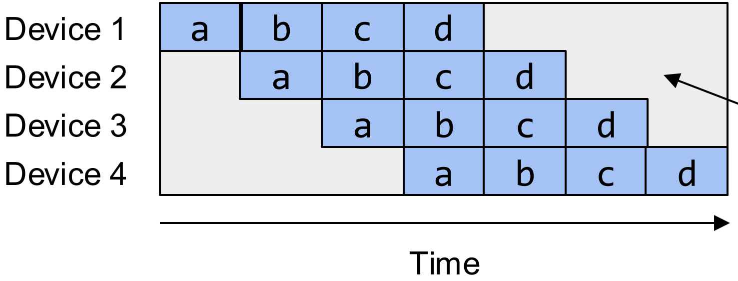


- Gray area ( indicates devices being idle (a.k.a. Pipeline bubbles).
- Only 1 device activated at a time.
- **Pipeline bubble percentage** = `bubble_area / total_area`
  = `(D - 1) / D`, assuming `D` devices.

Reduce Pipeline Bubbles via Pipelining Inputs

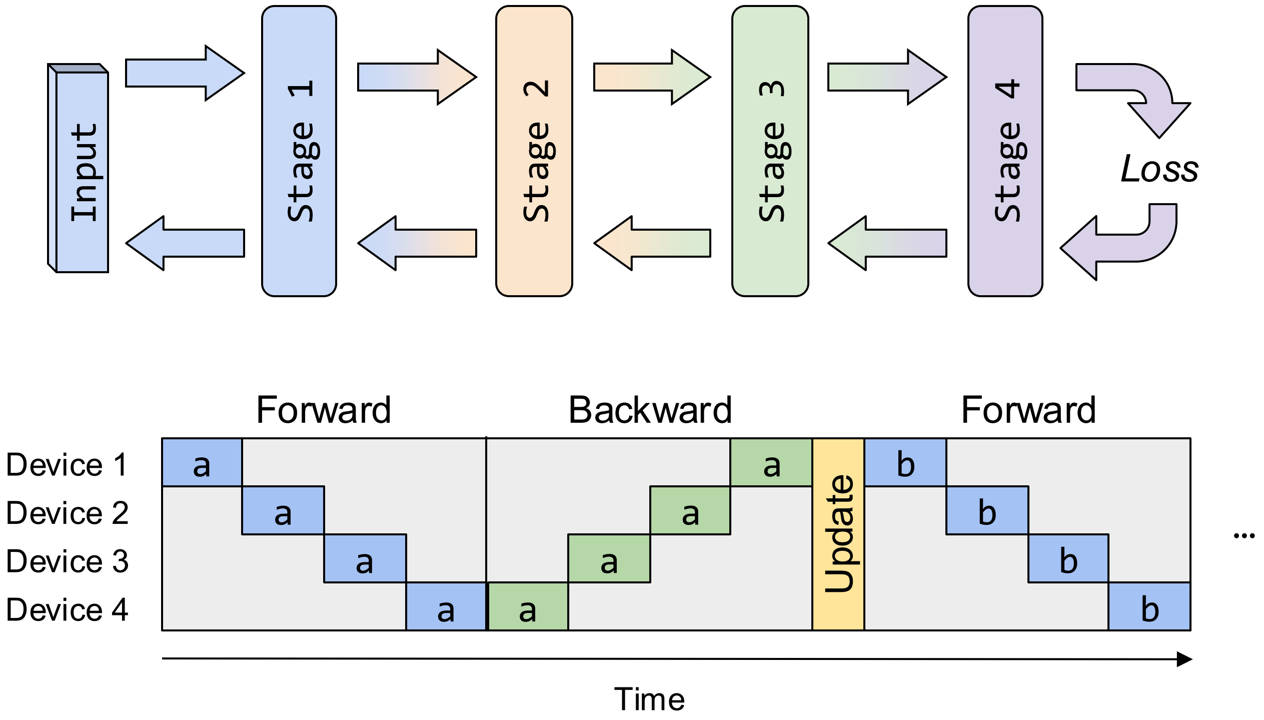


Used in inference.



Pipeline bubbles percentage
`= (D - 1) / (D - 1 + N)`
with `D` devices and `N` inputs.

Training: Forward & Backward Dependency

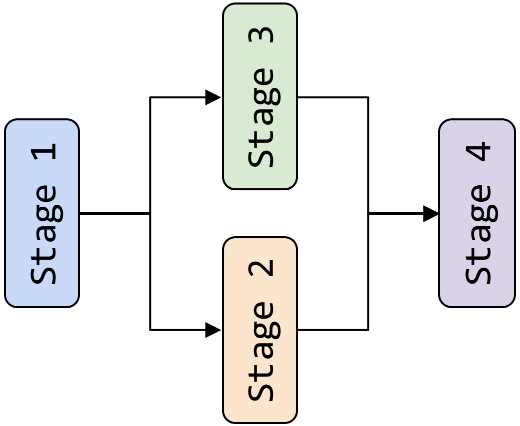


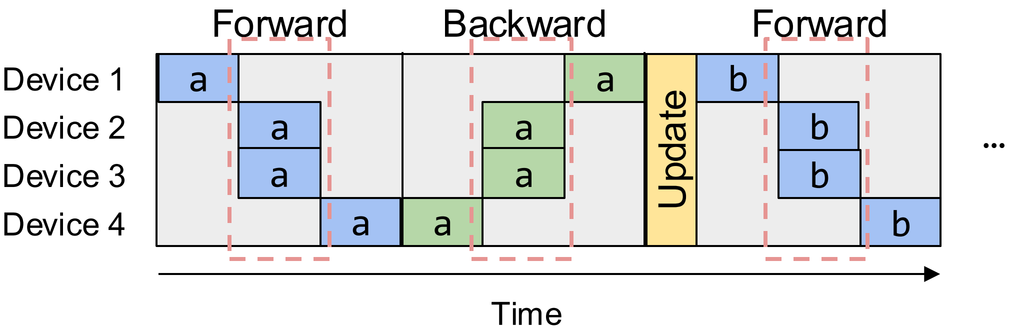
How to Reduce Pipeline Bubbles for Training?

- Device Placement
- Synchronous Pipeline Parallel Algorithms
    - GPipe
    - 1F1B
    - Interleaved 1F1B
    - TeraPipe
    - Chimera
- Asynchronous Pipeline Parallel Algorithms
    - AMPNet
    - Pipedream/Pipedream-2BW

Device Placement

Idea: Slice the branches of a neural network into multiple stages so they can be calculated concurrently.

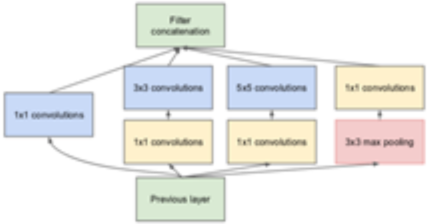




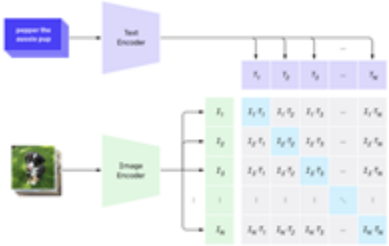
Mirhoseini, Azalia, et al. "Device placement optimization with reinforcement learning." *ICML 2017.*

Device Placement: Limitations

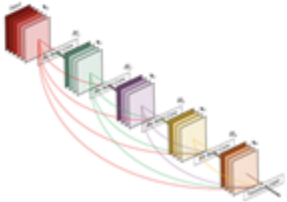
Only works for specific NNs with branches:



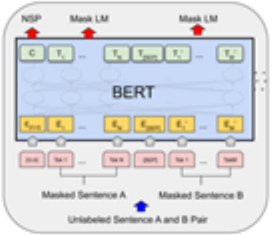
✅ Inception Module



✅ Contrastive Model

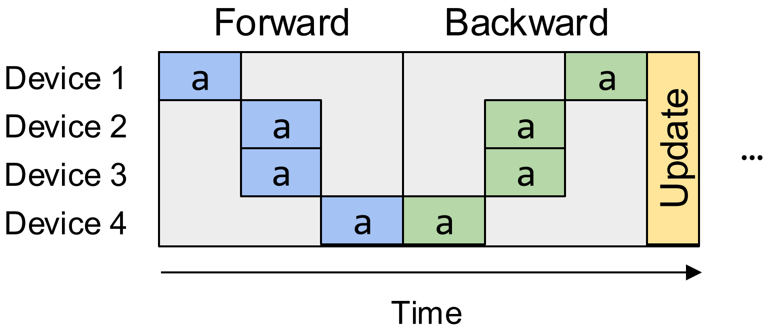


❌ Other ConvNets



❌ Transformers

Device Utilization is still low:



Note: device placement needs to be combined with the other pipeline schedules discussed later to further improve device utilization.

Synchronous Pipeline Parallel Schedule

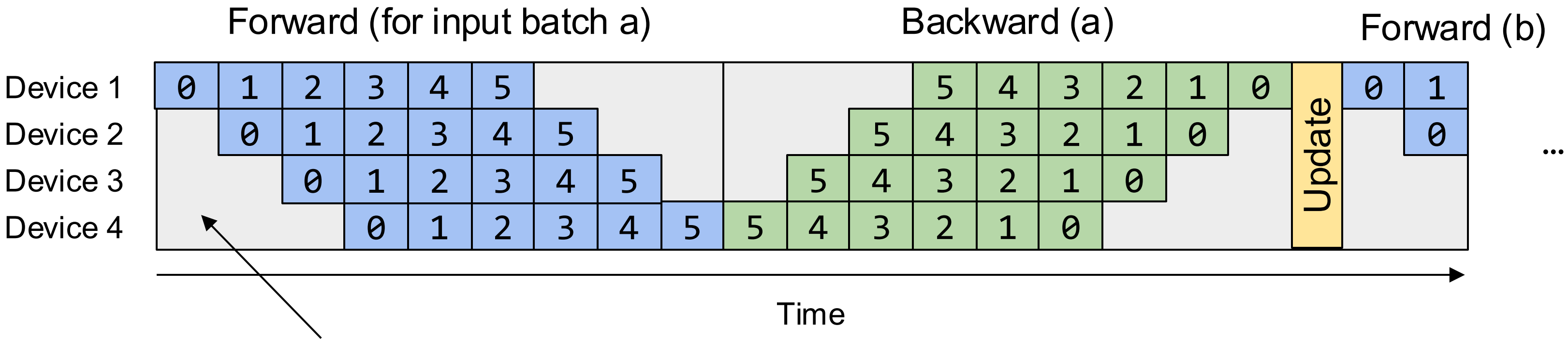
Idea: Modify pipeline schedule to improve efficiency, but keep the computation and convergence semantics exactly the same as if training with a single device.

GPipe

Idea: Partition the input batch into multiple "*micro-batches*". Pipeline the micro-batches. Accumulate the gradients of the micro-batches:

$$\nabla L_\theta(x) = \frac{1}{N} \sum_{i=1}^{N} \nabla L_\theta(x_i)$$

Example: Slice each input batch into 6 micro-batches:



Pipeline bubbles percentage `= (D - 1) / (D - 1 + N)`
with `D` devices and `N` micro-batches.

Huang, Yanping, et al. "Gpipe: Efficient training of giant neural networks using pipeline parallelism." *NeurIPS 2019.*

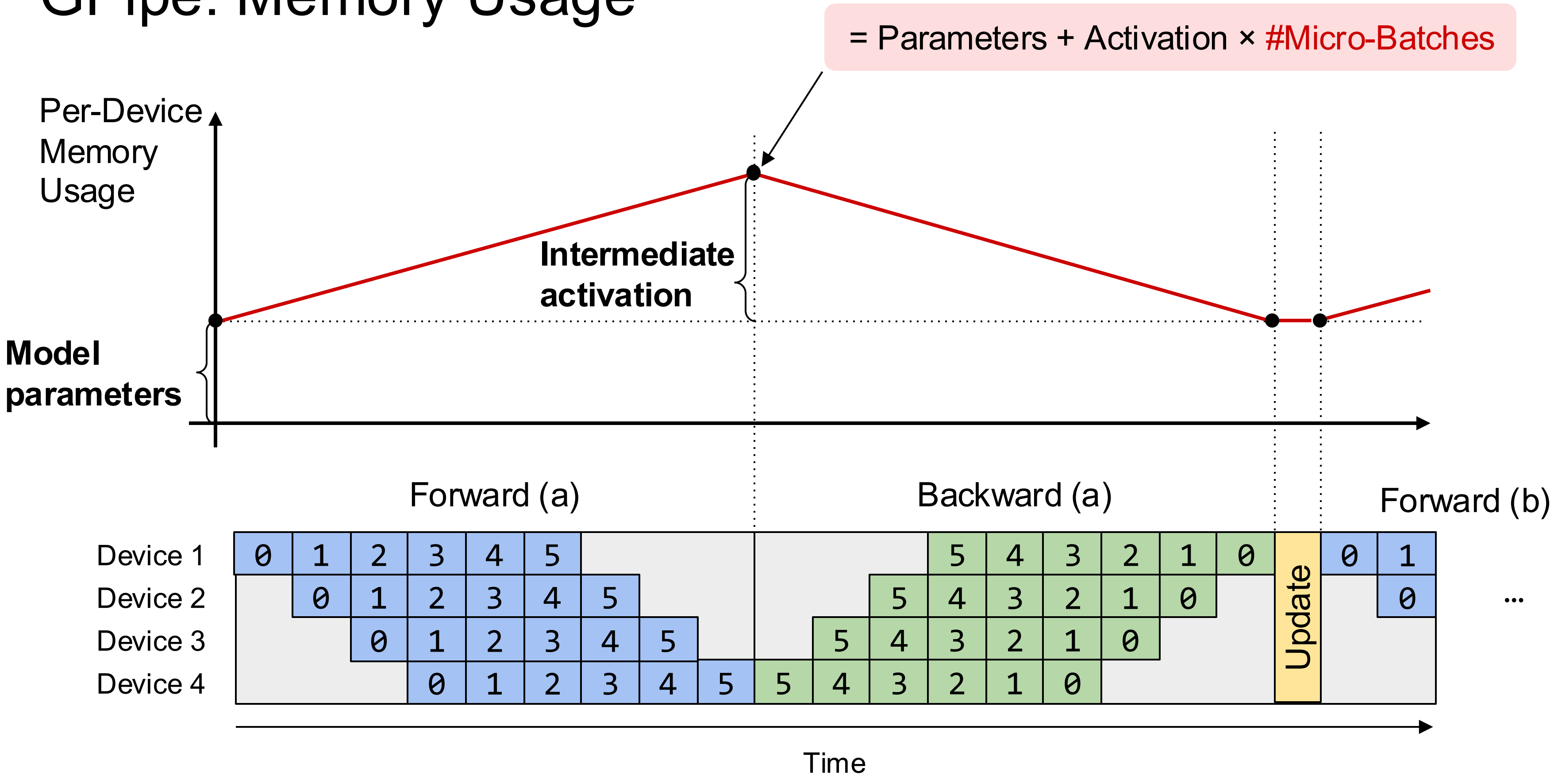
GPipe: Experimental Results

Table: Normalized training throughput using GPipe with different number of devices (stages) and different number of micro-batches M on TPUs.

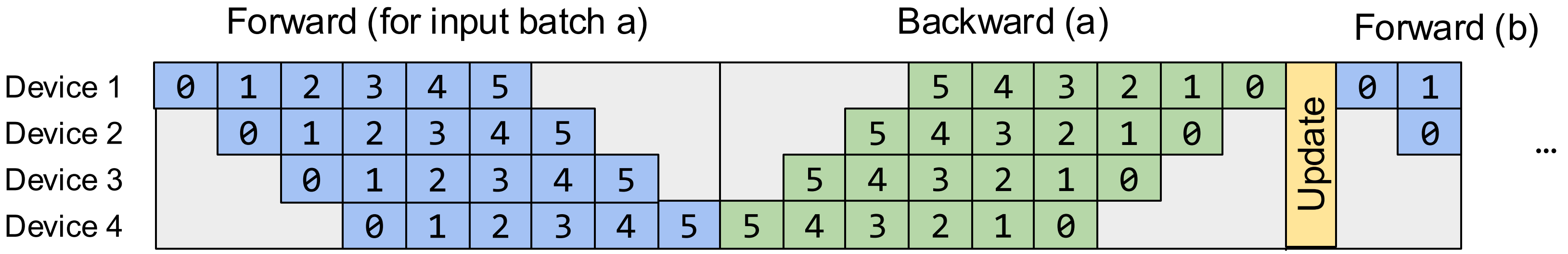
	#TPUs = 2	#TPUs = 4	#TPUs = 8
#Micro-batches = 1	1	1.07	1.3
#Micro-batches = 4	1.7	3.2	4.8
#Micro-batches = 32	1.8	3.4	6.3

Huang, Yanping, et al. "Gpipe: Efficient training of giant neural networks using pipeline parallelism." *NeurIPS 2019.*

GPipe: Memory Usage

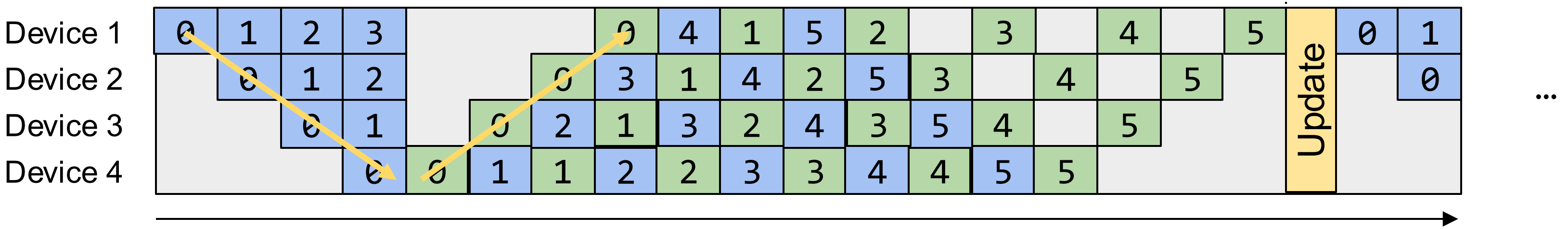


GPipe Schedule:



1F1B (1 Forward 1 Backward) Schedule:

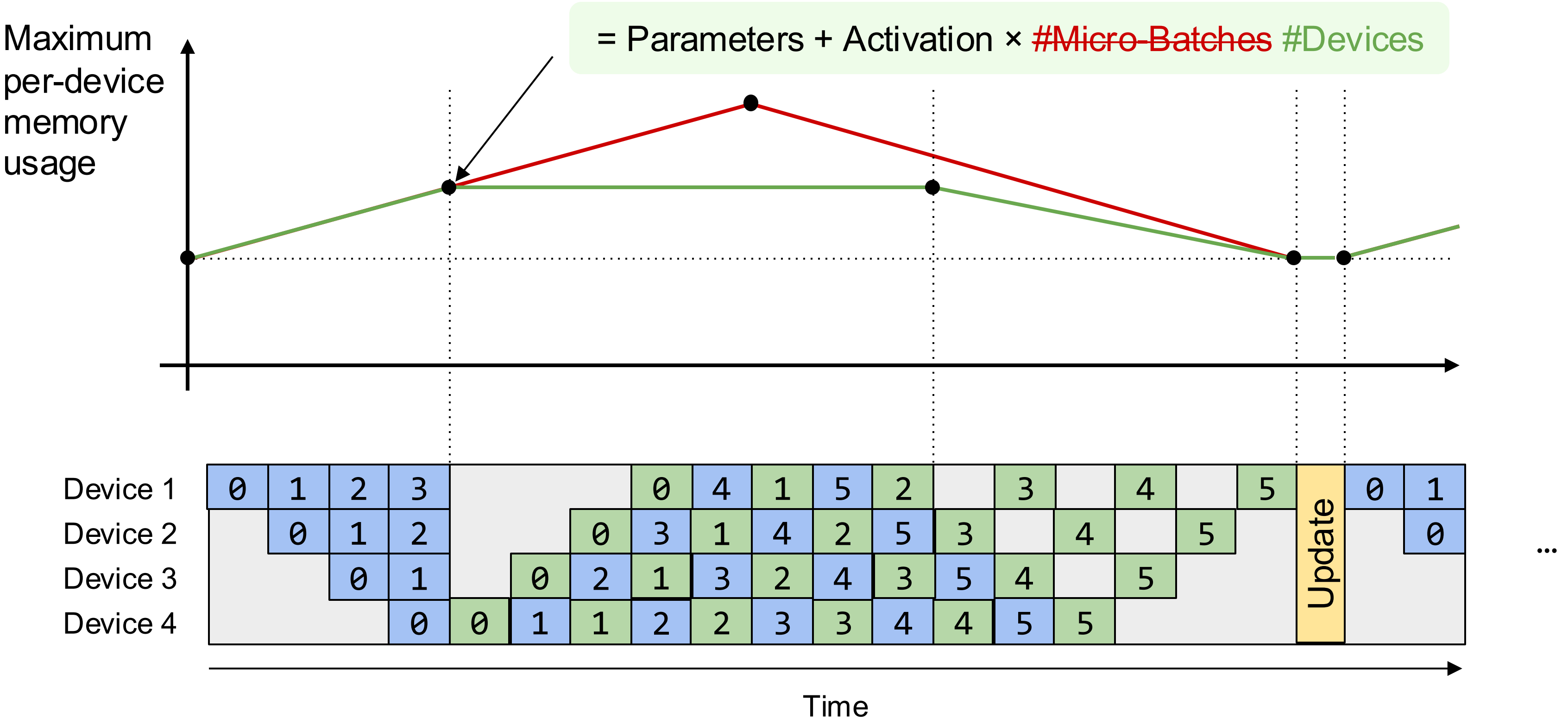
Same Latency



Perform backward as early as possible

Fan, Shiqing, et al. "DAPPLE: A pipelined data parallel approach for training large models." *PPoPP 2021.*

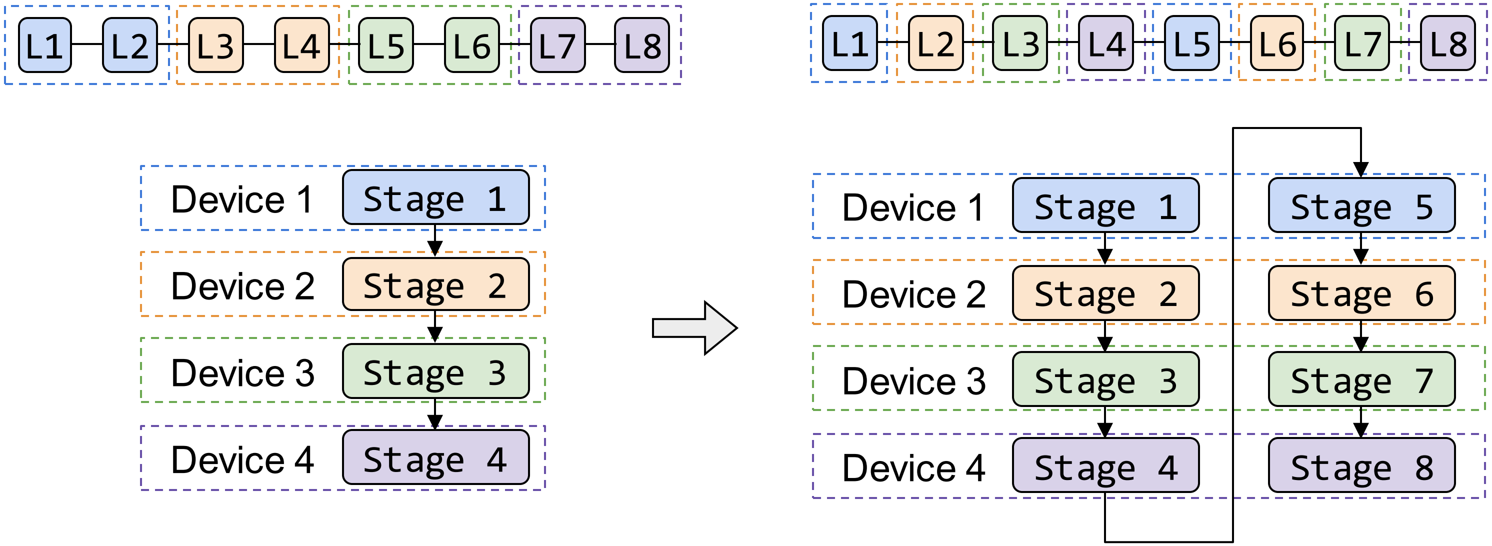
1F1B Memory Usage



Fan, Shiqing, et al. "DAPPLE: A pipelined data parallel approach for training large models." *PPoPP 2021.*

Interleaved 1F1B

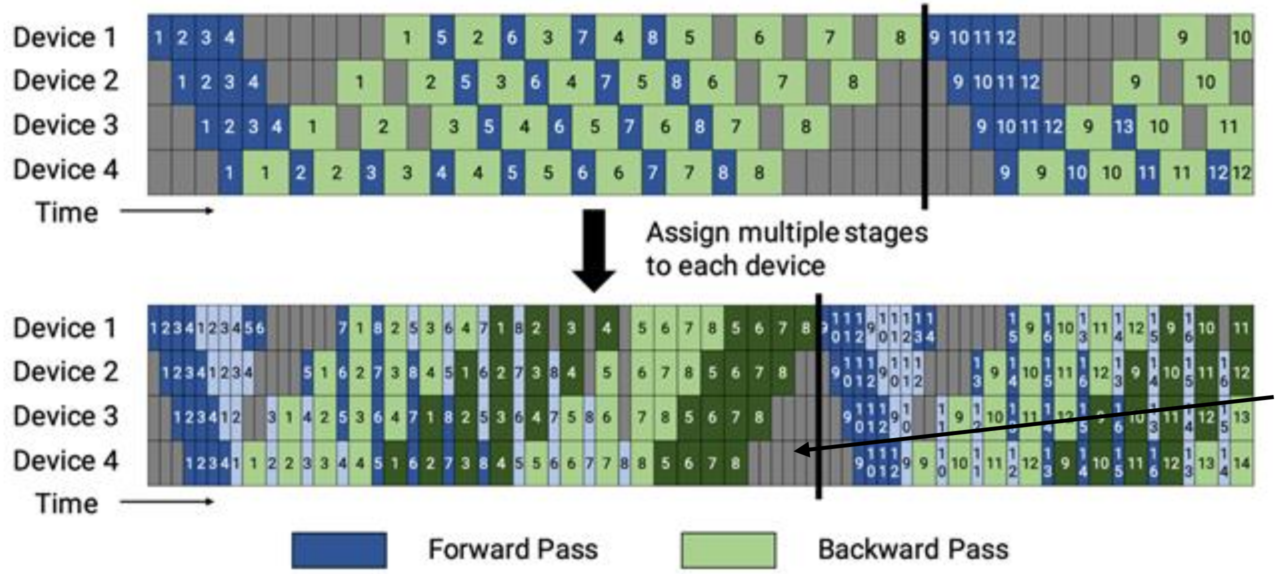
Idea: Slice the neural network into more fine-grained stages and assign multiple stages to reduce pipeline bubble.



Interleaved 1F1B

Pro:
Higher pipeline efficiency with fewer pipeline bubbles.

Con:
More communication overhead between stages.

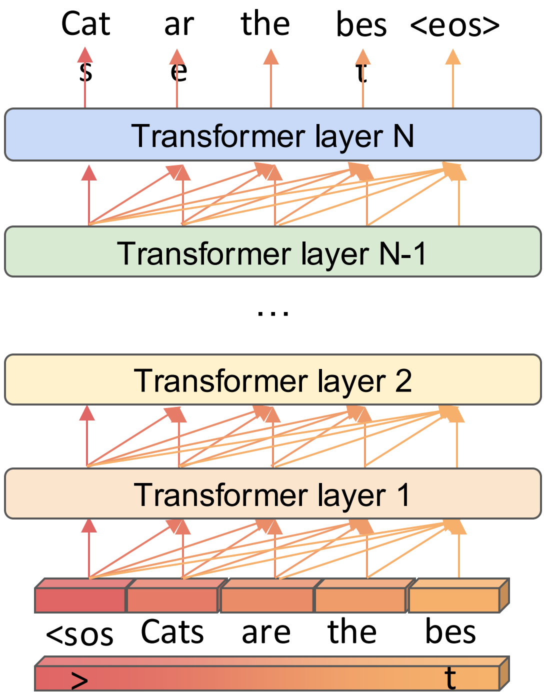


Pipeline bubbles percentage = `(D - 1) / (D - 1 + KN)` with `D` devices, `K` stages on each device, and `N` micro-batches.

TeraPipe

Idea: The computation of an input token only depends on previous tokens but not future tokens for autoregressive models.

Further reduce the bubble size by pipelining within a sequence.

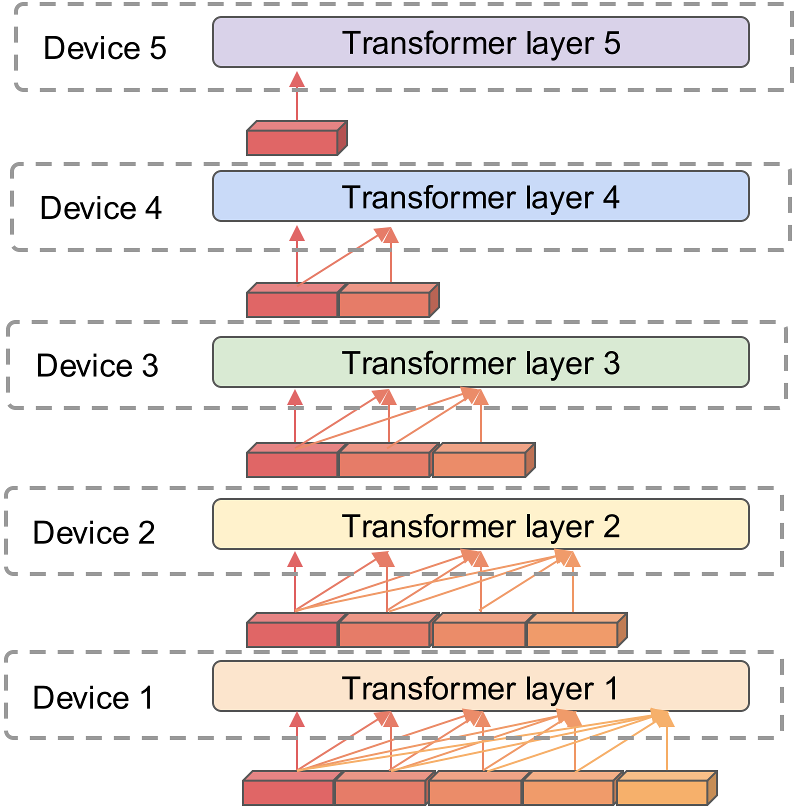


Li, Zhuohan, et al. "TeraPipe: Token-Level Pipeline Parallelism for Training Large-Scale Language Models." *ICML 2021.*

TeraPipe

Idea: The computation of an input token only depends on previous tokens but not future tokens for autoregressive models.

Further reduce the bubble size by pipelining within a sequence.

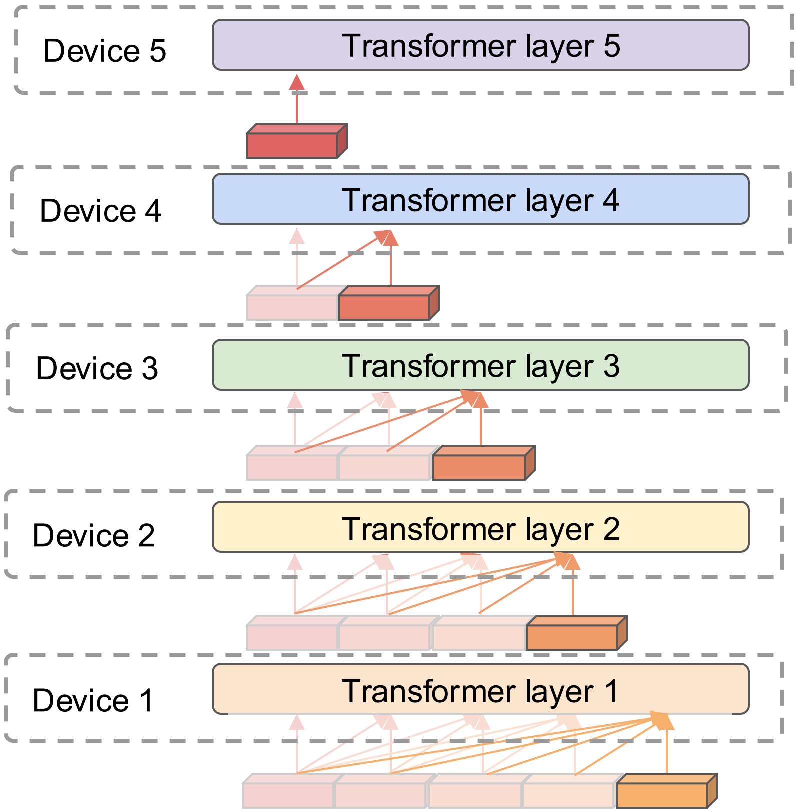


Li, Zhuohan, et al. "TeraPipe: Token-Level Pipeline Parallelism for Training Large-Scale Language Models." *ICML 2021.*

TeraPipe

Idea: The computation of an input token only depends on previous tokens but not future tokens for autoregressive models.

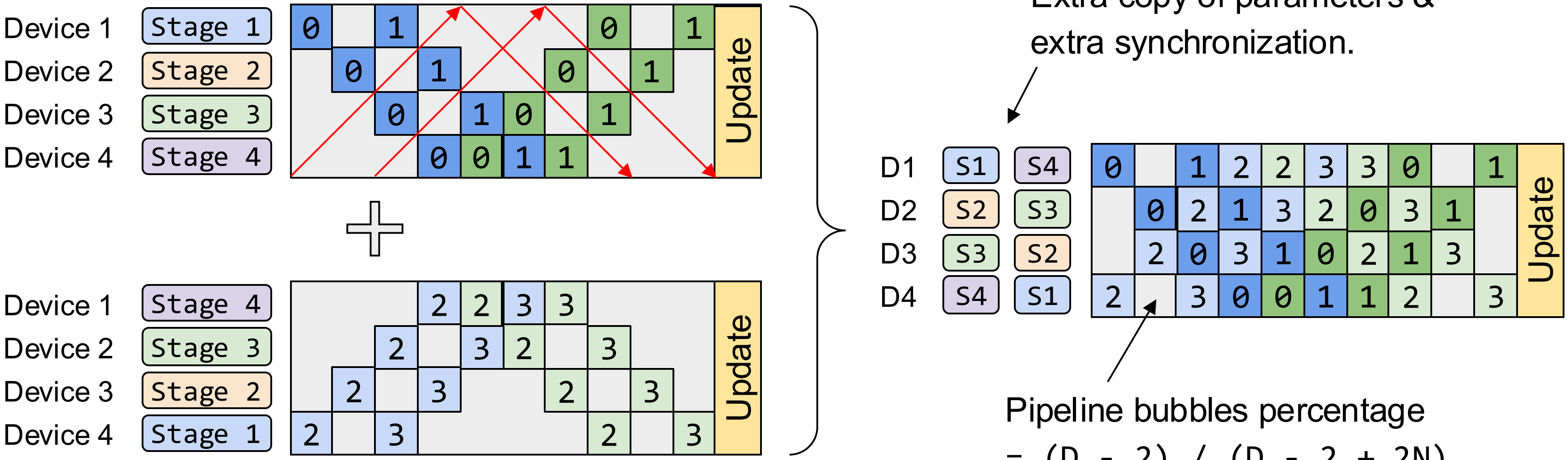
Further reduce the bubble size by pipelining within a sequence.



Li, Zhuohan, et al. "TeraPipe: Token-Level Pipeline Parallelism for Training Large-Scale Language Models." *ICML 2021.*

Chimera

Idea: Store bi-directional stages and combine bidirectional pipeline to further reduce pipeline bubbles.



Extra copy of parameters & extra synchronization.

Pipeline bubbles percentage
`= (D - 2) / (D - 2 + 2N)`
with `D` devices and `N` micro-batches.

Li, Shigang, and Torsten Hoefler. "Chimera: efficiently training large-scale neural networks with bidirectional pipelines." *SC 21.*

Synchronous Pipeline Schedule Summary

✅ **Pros:**

- Keep the convergence semantics. The training process is exactly the same as training the neural network on a single device.

❌ **Cons:**

- Pipeline bubbles.
- Reducing pipeline bubbles typically requires splitting inputs into smaller components, but too small input to the neural network will reduce the hardware efficiency.

Asynchronous Pipeline Schedules

Idea: Start next round of forward pass before backward pass finishes.

✅ **Pros:**

- No Pipeline bubbles.

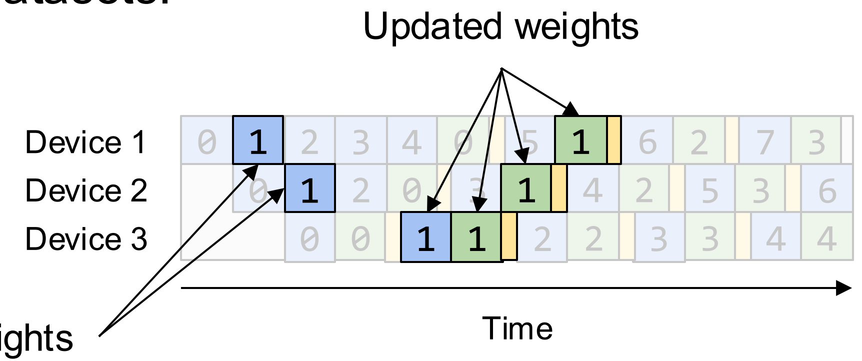
❌ **Cons:**

- Break the synchronous training semantics. Now the training will involve stalled gradient.
- Algorithms may store multiple versions of model weights for consistency.

AMPNet

Idea: Fully asynchronous. Each device performs forward pass whenever free and updates the weights after every backward pass.

Convergence: Achieve similar accuracy on small datasets (MNIST 97%), hard to generalize to larger datasets.

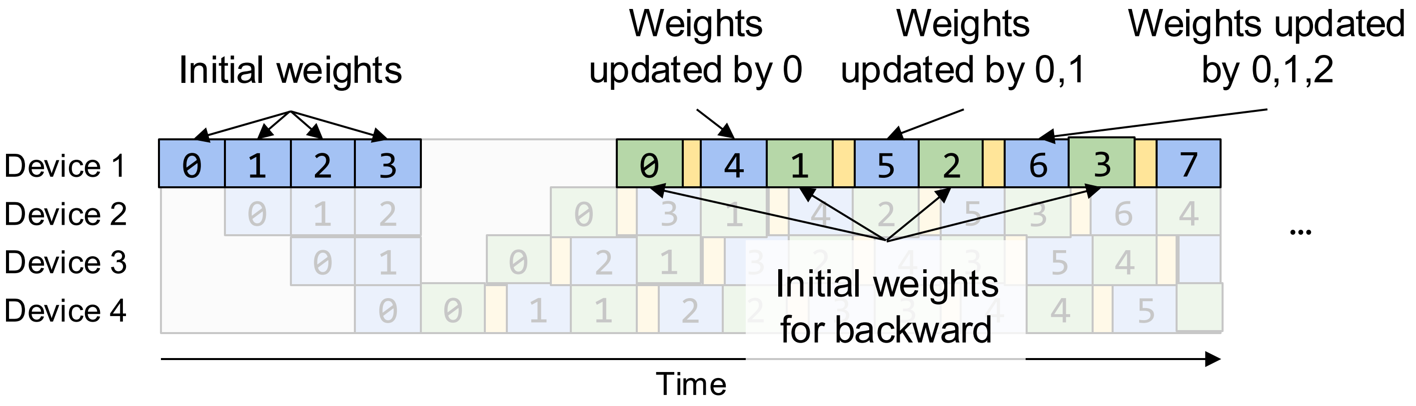


PipeMare: modify the optimizer to improve AMPNet convergence

Gaunt, Alexander L., et al. "AMPNet: Asynchronous model-parallel training for dynamic neural networks." *arXiv 2017.*
Yang, Bowen, et al. "Pipemare: Asynchronous pipeline parallel dnn training." *MLSys 2021.*

Pipedream

Idea: Enforce the same version of weight for a single input batch by storing multiple weight versions.
Convergence: Similar accuracy on ImageNet with a 5x speedup compared to data parallel.
Con: No memory saving compared to single device case.

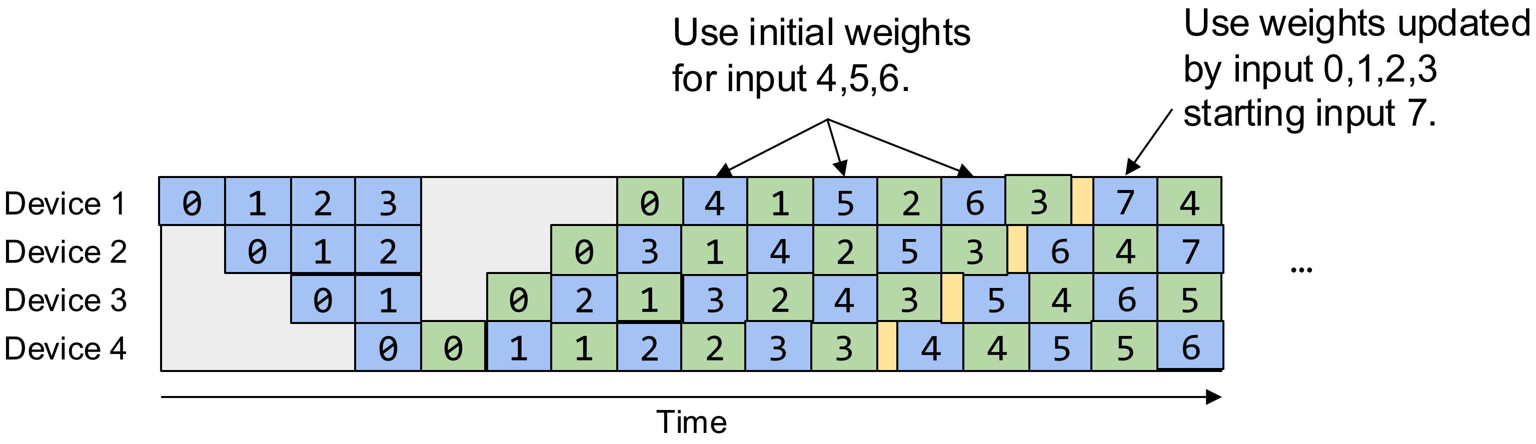


Narayanan, Deepak, et al. "PipeDream: generalized pipeline parallelism for DNN training." *SOSP 2019.*

Pipedream-2BW

Idea: Reduce Pipedream's memory usage (only store 2 copies) by updating weights less frequently. Weights always stalled by 1 update.

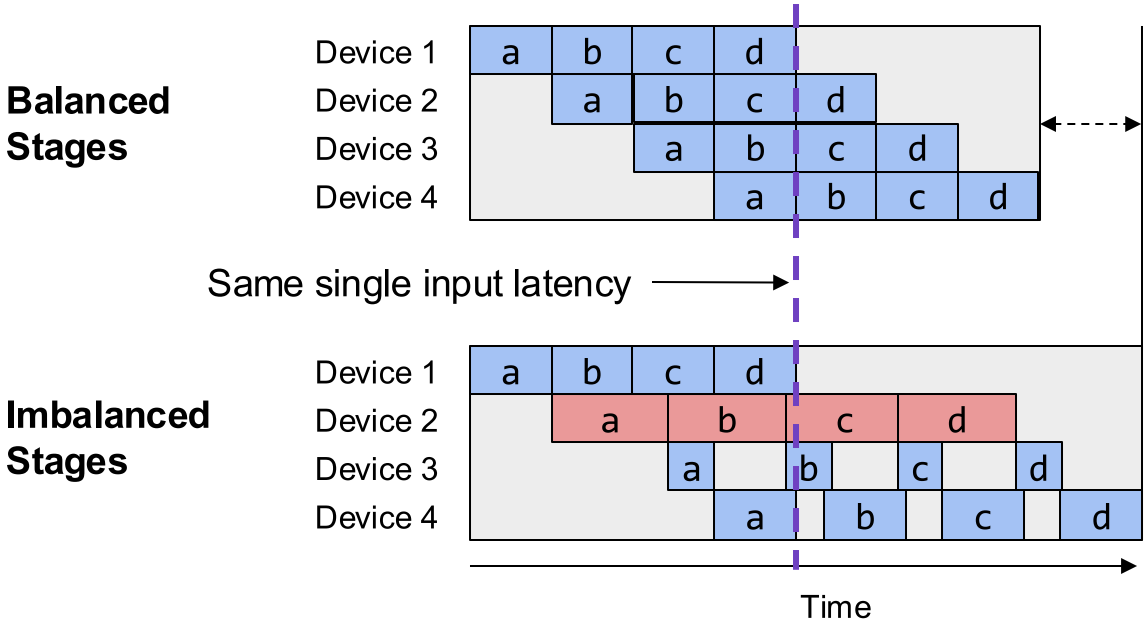
Convergence: Similar training accuracy on language models (BERT/GPT)



Narayanan, Deepak, et al. "Memory-efficient pipeline-parallel dnn training." *ICML 2021.*

Imbalanced Pipeline Stages

Pipeline schedules works best with balanced stages:



Frontier: Automatic Stage Partitioning

Goal: Minimize maximum stage latency & maximize parallelization

Reinforcement Learning Based (mainly for device placement):

1. Mirhoseini, Azalia, et al. "Device placement optimization with reinforcement learning." *ICML 2017.*
2. Gao, Yuanxiang, et al. "Spotlight: Optimizing device placement for training deep neural networks." *ICML 2018.*
3. Mirhoseini, Azalia, et al. "A hierarchical model for device placement." *ICLR 2018.*
4. Addanki, Ravichandra, et al. "Placeto: Learning generalizable device placement algorithms for distributed machine learning." *NeurIPS 2019.*
5. Zhou, Yanqi, et al. "Gdp: Generalized device placement for dataflow graphs." *Arxiv 2019.*
6. Paliwal, Aditya, et al. "Reinforced genetic algorithm learning for optimizing computation graphs." *ICLR 2020.*
7. *…*

Optimization (Dynamic Programming/Linear Programming) Based:

1. Narayanan, Deepak, et al. "PipeDream: generalized pipeline parallelism for DNN training." *SOSP 2019.*
2. Tarnawski, Jakub M., et al. "Efficient algorithms for device placement of dnn graph operators." *NeurIPS 2020.*
3. Fan, Shiqing, et al. "DAPPLE: A pipelined data parallel approach for training large models." *PPoPP 2021.*
4. Tarnawski, Jakub M., Deepak Narayanan, and Amar Phanishayee. "Piper: Multidimensional planner for dnn parallelization." *NeurIPS 2021.*
5. Zheng, Lianmin, et al. "Alpa: Automating Inter-and Intra-Operator Parallelism for Distributed Deep Learning." *OSDI 2022.*
6. *…*

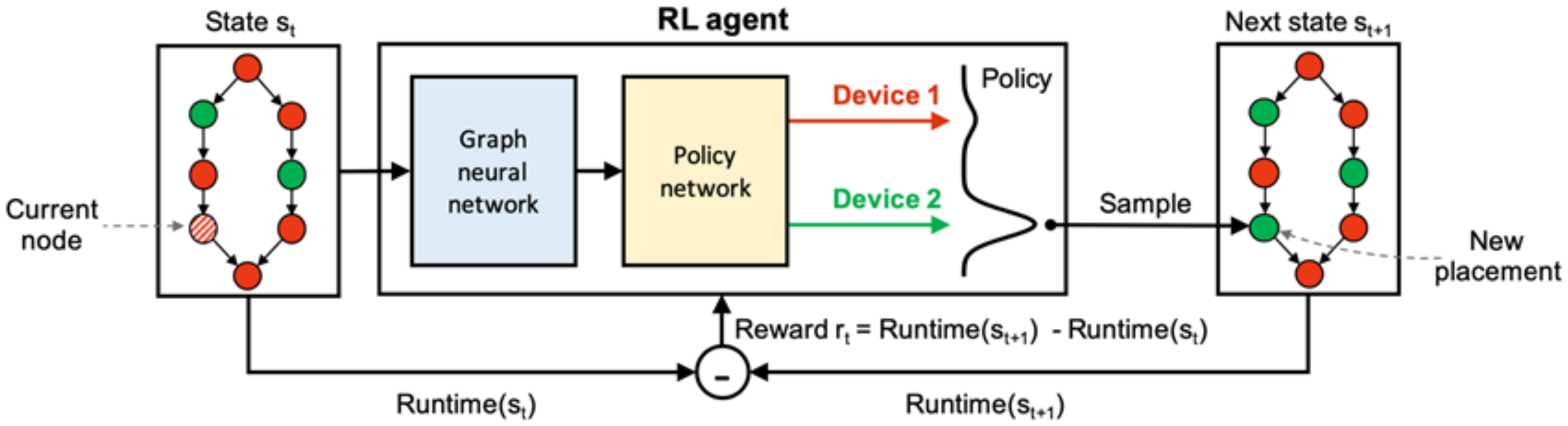
RL-Based Partitioning Algorithm

State: Device assignment plan for a computational graph.

Action: Modify the device assignment of a node.

Reward: Latency difference between the new and old placements.

Trained with **policy gradient** algorithm.



Addanki, Ravichandra, et al. "Placeto: Learning generalizable device placement algorithms for distributed machine learning." NeurIPS 2019.

Optimization-Based Partitioning Algorithm

Integer Linear Programming:

Variable: Decision variable vector for each operator, representing device assignment.

Minimize: Maximum finishing time of all operators.

Constraint: Execution dependency & memory capacity of each device.

$$
\begin{aligned}
\min \quad & \text{TotalLatency} \\
\text{s.t.} \quad & \textstyle\sum_{i=0}^{k} x_{vi} = 1 \\
& \text{subgraph } \{v \in V : x_{vi} = 1\} \text{ is contiguous} \\
& M \geq \textstyle\sum_{v} m_v \cdot x_{vi} \\
& \text{CommIn}_{ui} \geq x_{vi} - x_{ui} \\
& \text{CommOut}_{ui} \geq x_{ui} - x_{vi} \\
& \text{TotalLatency} \geq \text{Latency}_v \\
& \text{SubgraphStart}_i \geq \text{Latency}_v \cdot \text{CommIn}_{vi} \\
& \text{SubgraphFinish}_i = \text{SubgraphStart}_i + \textstyle\sum_{v} \text{CommIn}_{vi} \cdot c_v \\
& \qquad + \textstyle\sum_{v} x_{vi} \cdot p_v^{\text{acc}} + \textstyle\sum_{v} \text{CommOut}_{vi} \cdot c_v \\
& \text{Latency}_v \geq x_{v0} \cdot p_v^{\text{cpu}} \\
& \text{Latency}_v \geq x_{v0} \cdot p_v^{\text{cpu}} + \text{Latency}_u \\
& \text{Latency}_v \geq x_{vi} \cdot \text{SubgraphFinish}_i \\
& x_{vi} \in \{0, 1\}
\end{aligned}
$$

Tarnawski, Jakub M., et al. "Efficient algorithms for device placement of dnn graph operators." NeurIPS 2020.

Inter-operator Parallelism Summary

Idea: Assign different operators of the computational graph to different devices and executed in a pipelined fashion.

Method	General computational graph	No pipeline bubbles	Same convergence as single device
Device Placement	❌	❌	✅
Synchronous Schedule	✅	❌	✅
Asynchronous Schedule	✅	✅	❌

Stage Partitioning: Imbalance stage → More pipeline bubble

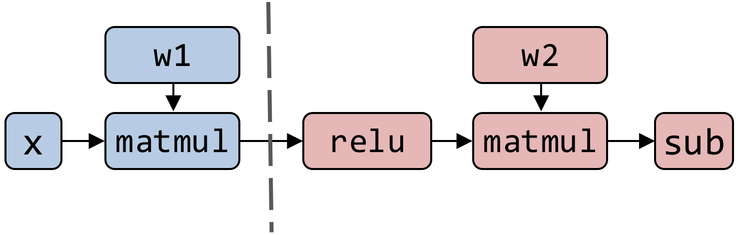
RL-Based / **Optimization-Based** Automatic Stage Partitioning

Where We Are

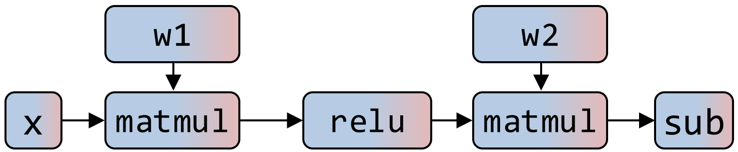
- Motivation
- History
- Parallelism Overview
- Data parallelism
- **Model parallelism**
    - Inter-op parallelism
    - Intra-op parallelism
- Auto-parallelization

Recap: Intra-op and Inter-op

Strategy 1: Inter-operator Parallelism



Strategy 2: Intra-operator Parallelism



This section:

1. How to parallelize an **operator** ?
2. How to parallelize a **graph** ?

Parallelize One Operator

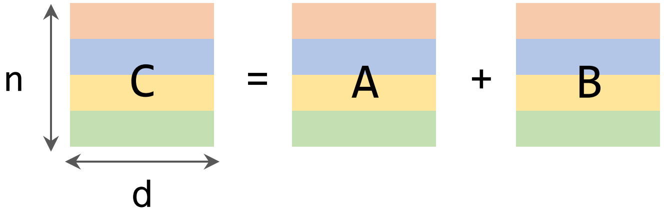
Element-wise operators

```
for n in range(0, N):
  for d in range(0, D):
    C[n,d] = A[n,d] + B[n,d]
```

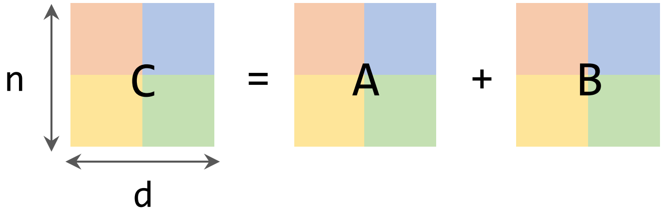
No dependency on the two for-loops.
Can arbitrarily split the for-loops on different devices.

device 1 device 2 device 3 device 4

Parallelize loop n



Parallelize both loop n and loop d



a lot of other variants …

Parallelize One Operator

Matrix multiplication

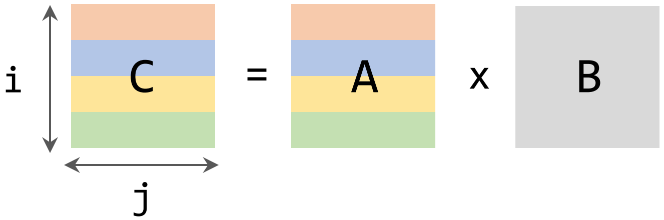
No dependency on the two spatial for-loops.
Can arbitrarily split the for-loops on different devices.

```
for i in range(0, N):
  for j in range(0, M):
    for k in range(0, K):
      C[i,j] = C[i,j] + A[i,k] x B[k,j]
```

Accumulation on this reduction loop.
Have to accumulate partial results if we split this for-loop

device 1 device 2 device 3 device 4 replicated

Parallelize loop i



$$\begin{bmatrix} C_1 \\ C_2 \\ C_3 \\ C_4 \end{bmatrix} = \begin{bmatrix} A_1 \\ A_2 \\ A_3 \\ A_4 \end{bmatrix} \times B$$

Parallelize One Operator

Matrix multiplication

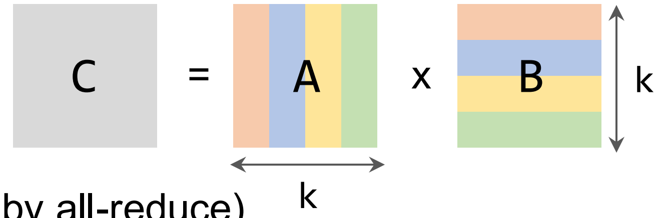
No dependency on the two spatial for-loops.
Can arbitrarily split the for-loops on different devices.

```
for i in range(0, N):
  for j in range(0, M):
    for k in range(0, K):
      C[i,j] = C[i,j] + A[i,k] x B[k,j]
```

Accumulation on this reduction loop.
Have to accumulate partial results if we split this for-loop

device 1 device 2 device 3 device 4 replicated

Parallelize loop k



(got by all-reduce)

$$C = [A_1 \ A_2 \ A_3 \ A_4] \begin{bmatrix} B_1 \\ B_2 \\ B_3 \\ B_4 \end{bmatrix} = A_1 B_1 + A_2 B_2 + A_3 B_3 + A_4 B_4$$

Parallelize One Operator

Matrix multiplication

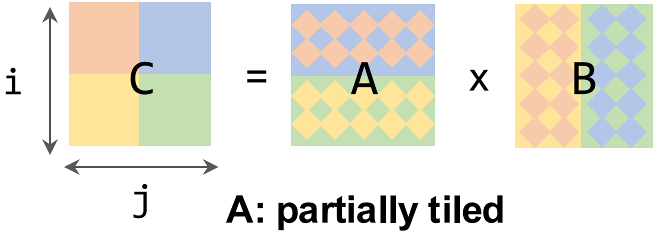
No dependency on the two spatial for-loops.
Can arbitrarily split the for-loops on different devices.

```
for i in range(0, N):
  for j in range(0, M):
    for k in range(0, K):
      C[i,j] = C[i,j] + A[i,k] x B[k,j]
```

Accumulation on this reduction loop.
Have to accumulate partial results if we split this for-loop

device 1 device 2 device 3 device 4

Parallelize loop i and j

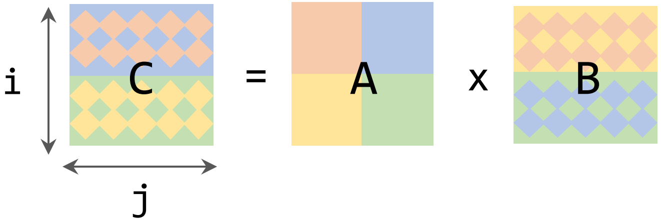


A: partially tiled

Device 1 and 2 hold a replicated tile

Device 3 and 4 hold a replicated tile

Parallelize loop i and k



C: got by all-reduce

a lot of
other variants
…

Parallelize One Operator

2D Convolution

```
for n in range(0, N):
  for co in range(0, CO):
    for h in range(0, H):
      for w in range(0, W):
        for ci in range(0, CI):
          for kh in range(0, KH):
            for kw in range(0, KW):
              C[n,co,h,w] += A[n,co,h+kh,w+kw] x B[kh,kw,co,ci]
```

Simple spatial loops. Can be arbitrarily split.

Stencil computation loops. Splitting these requires careful boundary handling.

Reduction loop. Need to accumulate partial results.

Reduction loops. But usually too small (<= 5) for parallelization.

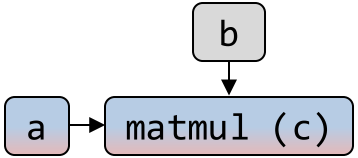
Simple case: Parallelize loop n, co, ci, then the parallelization strategies are almost the same as matmul's.

Complicated case: Parallelize loop h and w

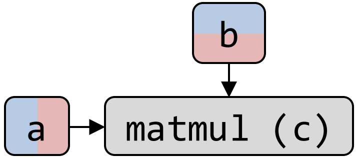
Data Parallelism as A Case of Intra-op Parallelism

Replicated Row-partitioned Column-partitioned

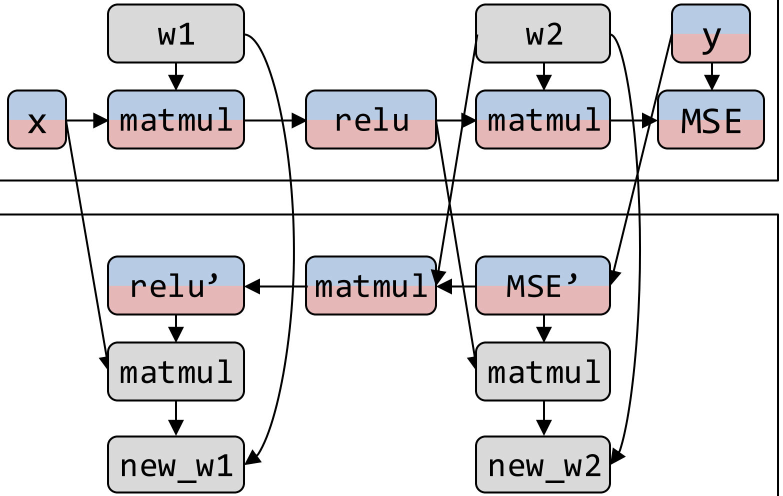
Matmul Parallelization Type 1
communication cost = 0



Matmul Parallelization Type 2
communication cost = all-reduce(c)



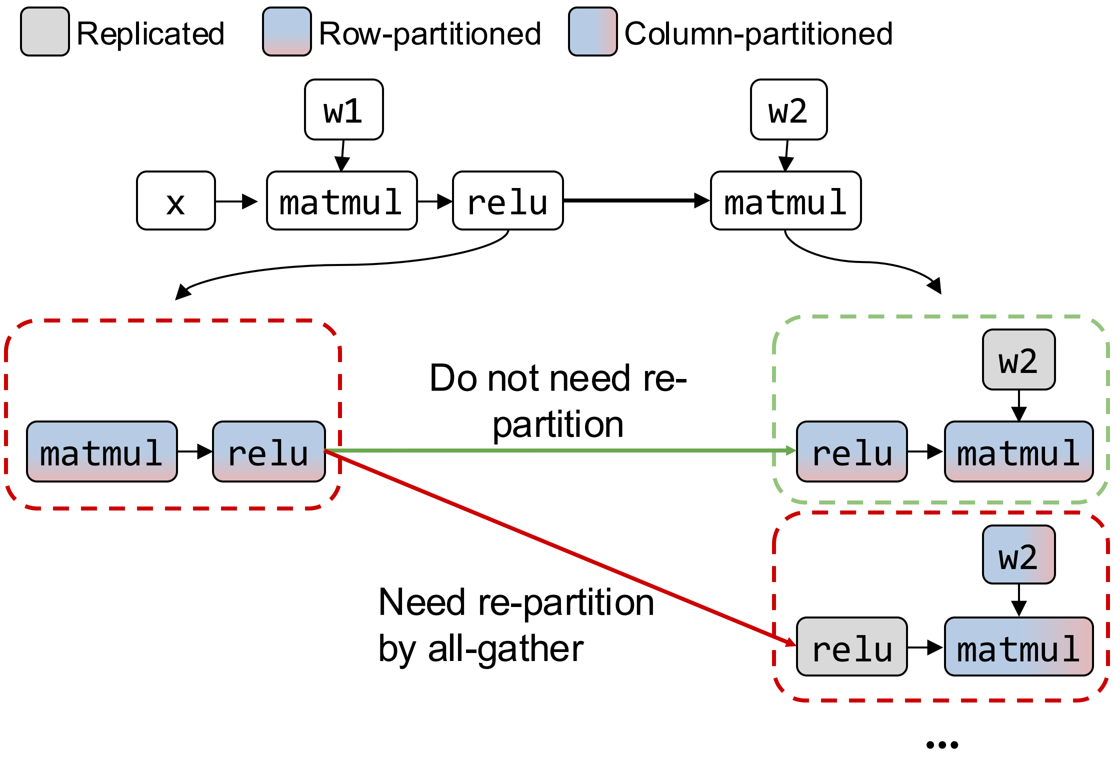
Forward Pass
Two “Type 1” matmuls: no communication



Backward Pass
One “Type 1” matmul: no communication
Two “Type 2” matmuls: require all-reduce

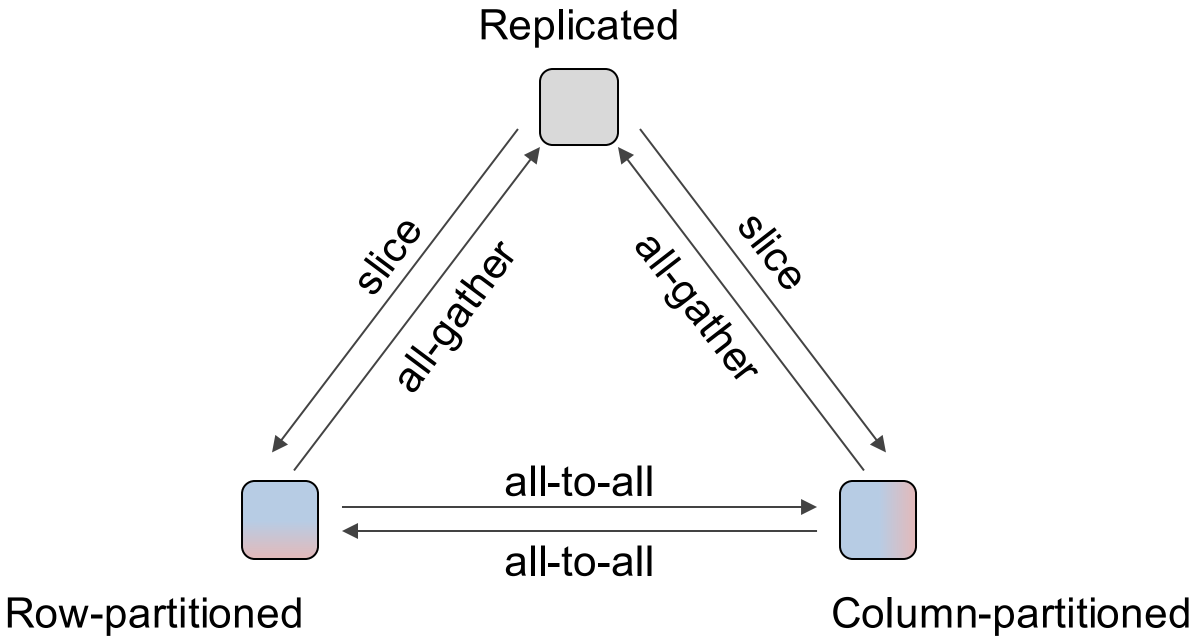
Re-partition Communication Cost

Different operators' parallelization strategies require different partition format of the same tensor



Re-partition Communication Cost

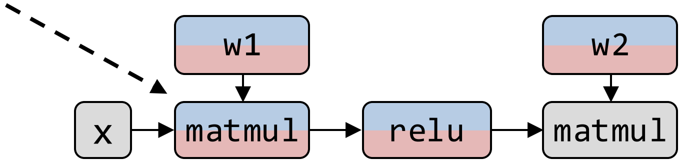
Different operators' parallelization strategies require different partition format of the same tensor



Parallelize All Operators in a Graph

Problem

Pick a parallel strategy
of each operator



Minimize Node costs (computation + communication) + Edge costs (re-partition communication)

Solution

Manual design
Randomized search
Dynamic programming
Integer linear programming

Important Projects

Model-specific Intra-op Parallel Strategies

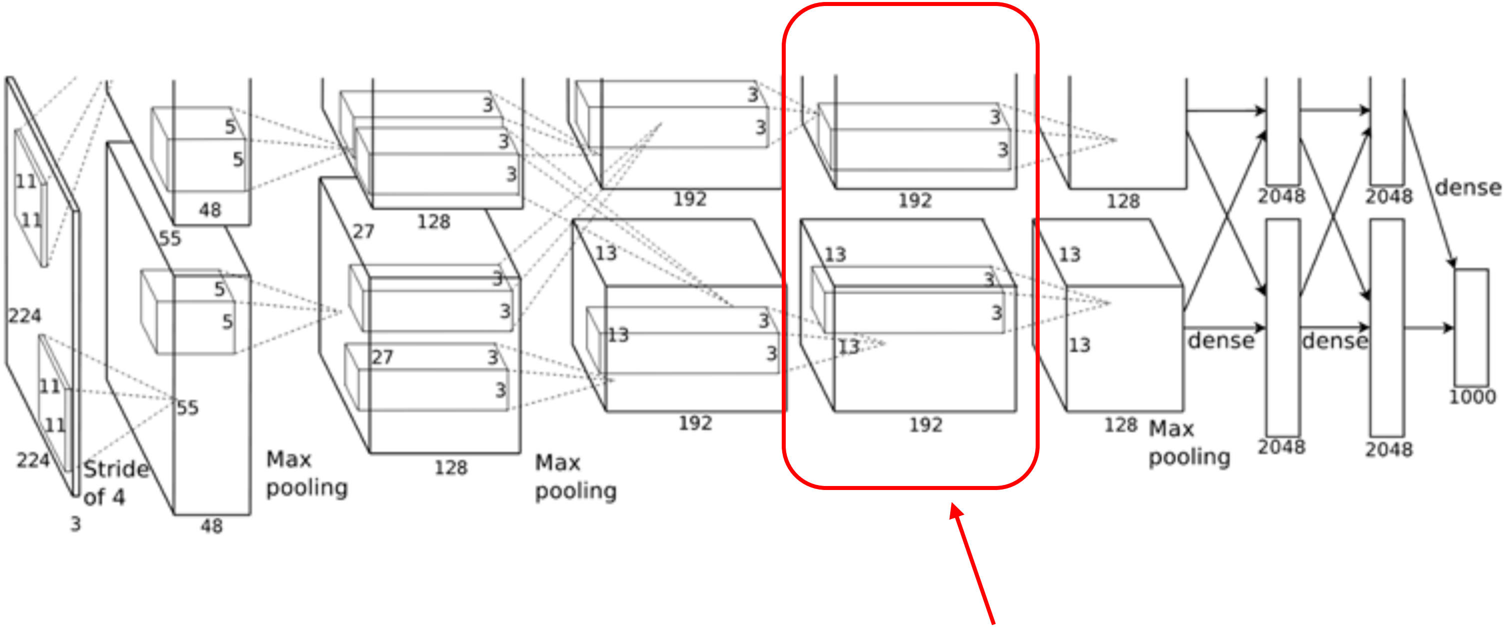
- AlexNet
- Megatron-LM
- GShard MoE

Systems for Intra-op Parallelism

- ZeRO
- Mesh-Tensorflow
- GSPMD
- Tofu
- FlexFlow

AlexNet

Result: increase top-1 accuracy by 1.7%



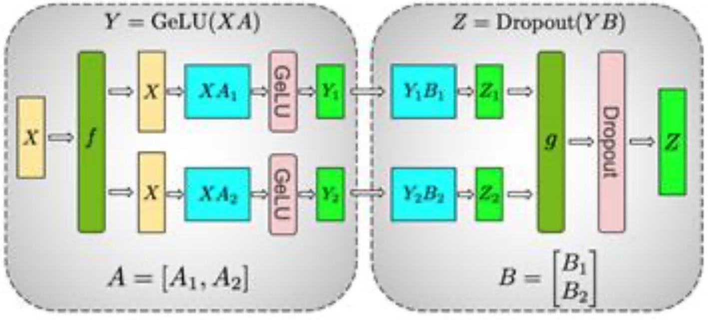
Assign a group convolution layer to 2 GPUs

Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E. Hinton. "Imagenet classification with deep convolutional neural networks." *NeurIPS 2012*

Megaton-LM

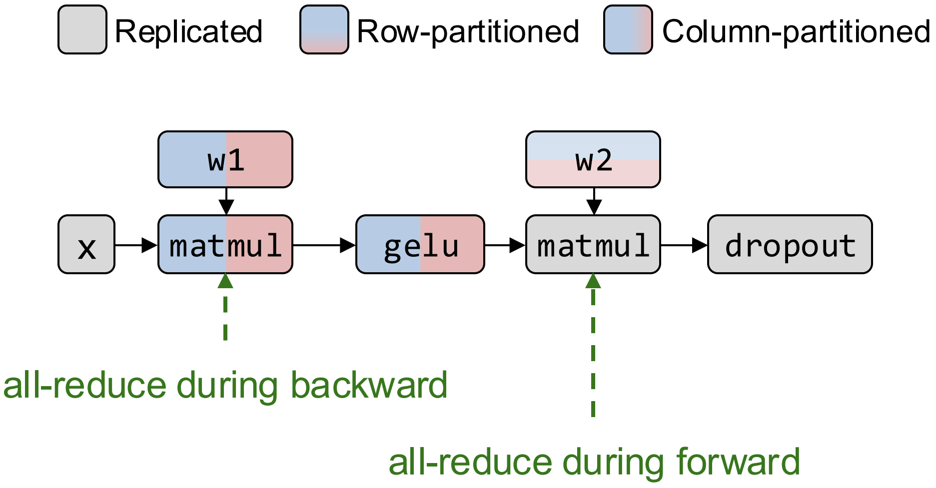
Result: a large language model with 8.3B parameters that outperforms SOTA results

Figure 3 from the paper :
How to partition the MLP in the transformer.



(a) MLP

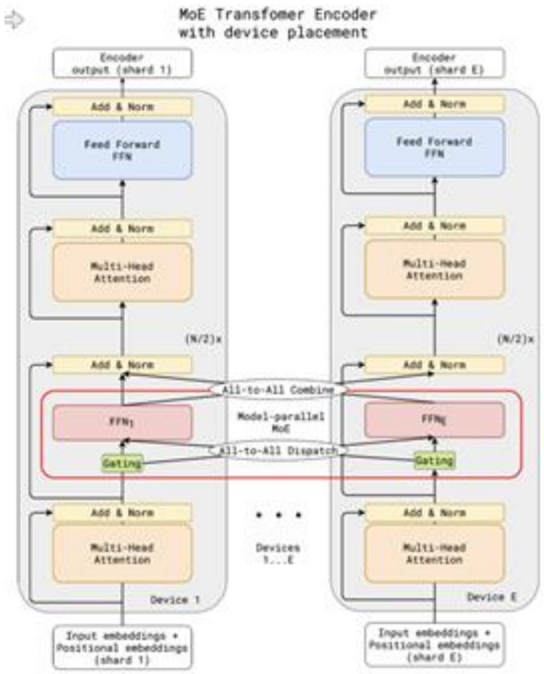
Illustrated with the notations in this tutorial



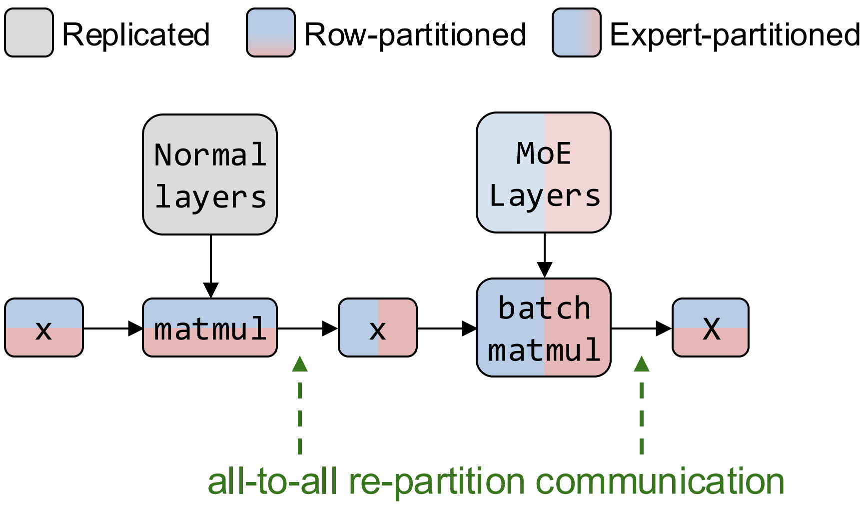
Shoeybi, Mohammad, et al. "Megatron-LM: Training multi-billion parameter language models using model parallelism."

GShard MoE

Result: a multi-language translation model with 600B parameters that outperforms SOTA



Illustrated with the notations in this tutorial



Lepikhin, Dmitry, et al. "GShard: Scaling giant models with conditional computation and automatic sharding." *ICLR 2021*

ZeRO Optimizer

Problem
Data parallelism replicates gradients, optimizer states and model weights on all devices.

Idea
Partition gradients, optimizer states and model weights.

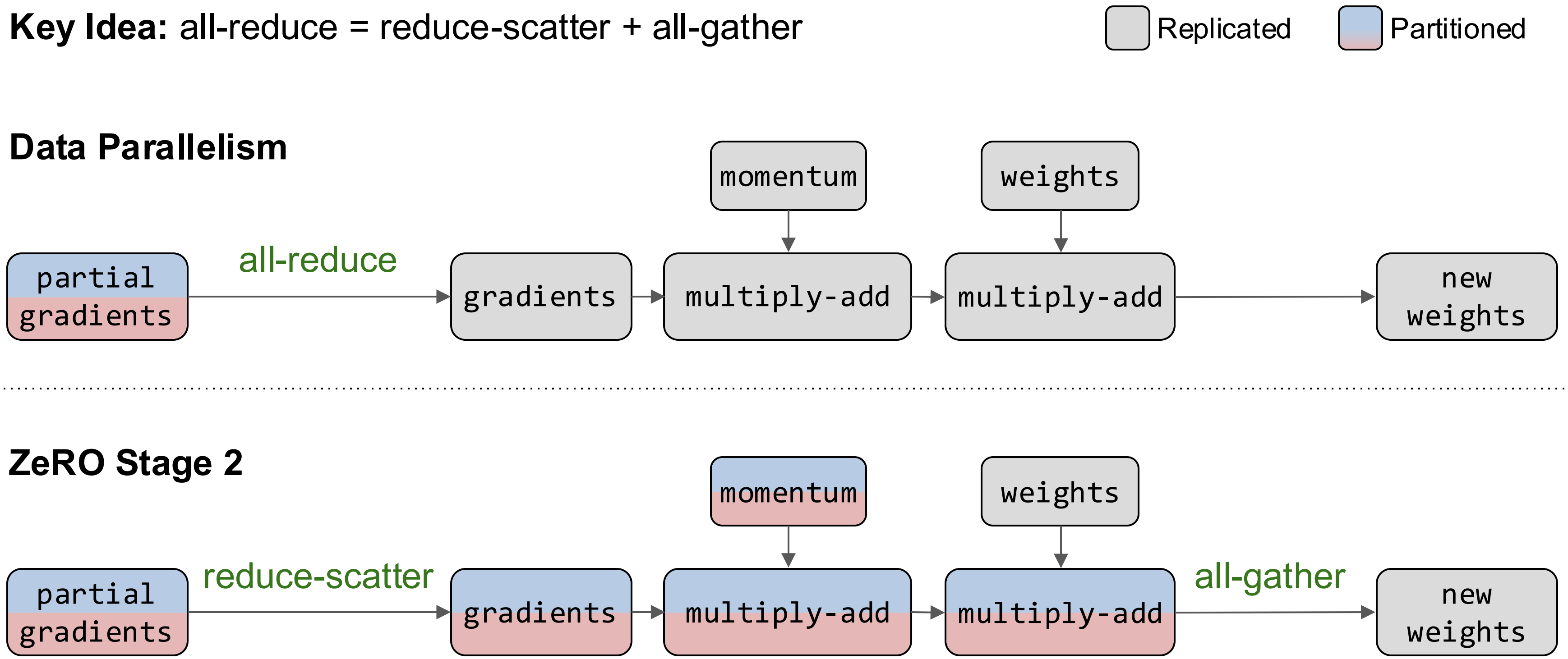
M is the number of parameters, N is the number of devices.

	Optimizer States (12M)	Gradients (2M)	Model Weights (2M)	Memory Cost	Communication Cost
Data Parallelism	Replicated	Replicated	Replicated	$16M$	all-reduce(2M)
ZeRO Stage 1	Partitioned	Replicated	Replicated	$4M + \frac{12M}{N}$	all-reduce(2M)
ZeRO Stage 2	Partitioned	Partitioned	Replicated	$2M + \frac{14M}{N}$	all-reduce(2M)
ZeRO Stage 3	Partitioned	Partitioned	Partitioned	$\frac{16M}{N}$	1.5 all-reduce(2M)

Rajbhandari, Samyam, et al. "Zero: Memory optimizations toward training trillion parameter models." *SC 2020*

ZeRO Stage 2

Key Idea: all-reduce = reduce-scatter + all-gather



Same communication cost but save memory by partitioning more tensors

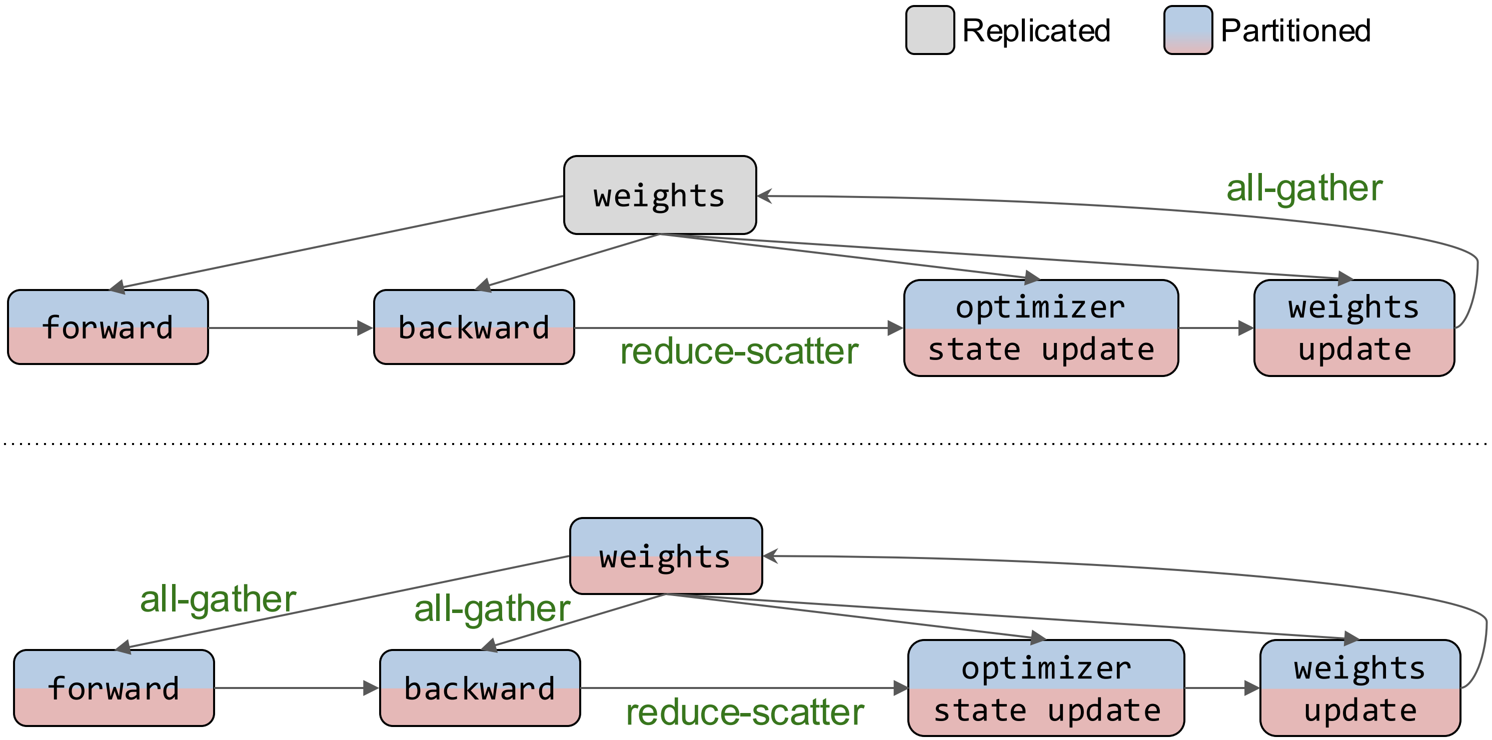
ZeRO Stage 3

ZeRO Stage 2

communication cost
= all-reduce

ZeRO Stage 3

communication cost
= 1.5 all-reduce



Mesh-Tensorflow

Map tensor dimension to mesh dimension for parallelism

```
...
batch = mtf.Dimension("batch", b)
io = mtf.Dimension("io", d_io)
hidden = mtf.Dimension("hidden", d_h)
# x.shape == [batch, io]
w = mtf.get_variable("w", shape=[io, hidden])
bias = mtf.get_variable("bias", shape=[hidden])
v = mtf.get_variable("v", shape=[hidden, io])
h = mtf.relu(mtf.einsum(x, w, output_shape=[batch, hidden]) + bias)
y = mtf.einsum(h, v, output_shape=[batch, io])
...
```

Tensor dimension

```
mesh_shape = [("rows", r), ("cols", c)]
computation_layout = [("batch", "rows"), ("hidden", "cols")]
```

Mesh dimension

Mapping

Shazeer, Noam, et al. "Mesh-tensorflow: Deep learning for supercomputers." *NeurIPS* 2018.

GSPMD

- Use annotations to specify partition strategy
- Propagate the annotations to whole graph
- Use compiler to generate SPMD (Single Program Multiple Data) parallel executables

```
  # Partition inputs along group (G) dim.
+ inputs = split(inputs, 0, D)
  # Replicate the gating weights
+ wg = replicate(wg)
  gates = softmax(einsum("GSM,ME->GSE", inputs, wg))
  combine_weights, dispatch_mask = Top2Gating(gating_logits)
  dispatched_expert_inputs = einsum(
    "GSEC,GSM->EGCM", dispatch_mask, reshaped_inputs)
  # Partition dispatched inputs along expert (E) dim.
+ dispatched_expert_inputs = split(dispatched_expert_inputs, 0, D)
  h = einsum("EGCM,EMH->EGCH", dispatched_expert_inputs, wi)
  ...
```

Xu, Yuanzhong, et al. "GSPMD: general and scalable parallelization for ML computation graphs." *arXiv 2021*

Tofu

Tensor description language for automatic parallelization analysis

```
@tofu.op
def conv1d(data, filters):
  return lambda b, co, x:
    Sum(lambda ci, dx: data[b, ci, x+dx]*filters[ci, co, dx]
```

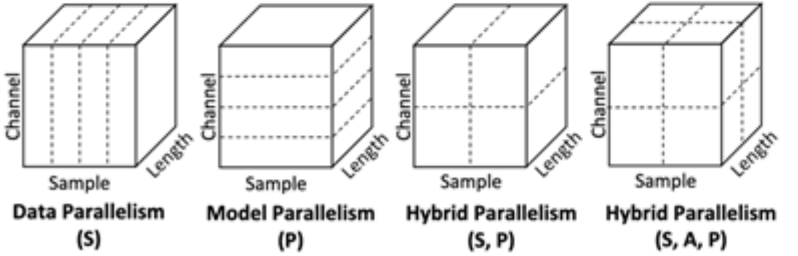
Dynamic programming for graph-level optimization

- Use graph coarsening to merge operators (e.g., elementwise-ops)
- Use dynamic programming with recursive partitioning

Wang, Minjie, Chien-chin Huang, and Jinyang Li. "Supporting very large models using automatic dataflow graph partitioning." *EuroSys 2019*

FlexFlow

Operator	Parallelizable Dimensions		
	(S)ample	(A)ttribute	(P)arameter
1D pooling	sample	length, channel	
1D convolution	sample	length	channel
2D convolution	sample	height, width	channel
Matrix multiplication	sample		channel



SOAP parallelism space
- Sample, Operator, Attribute, Parameter

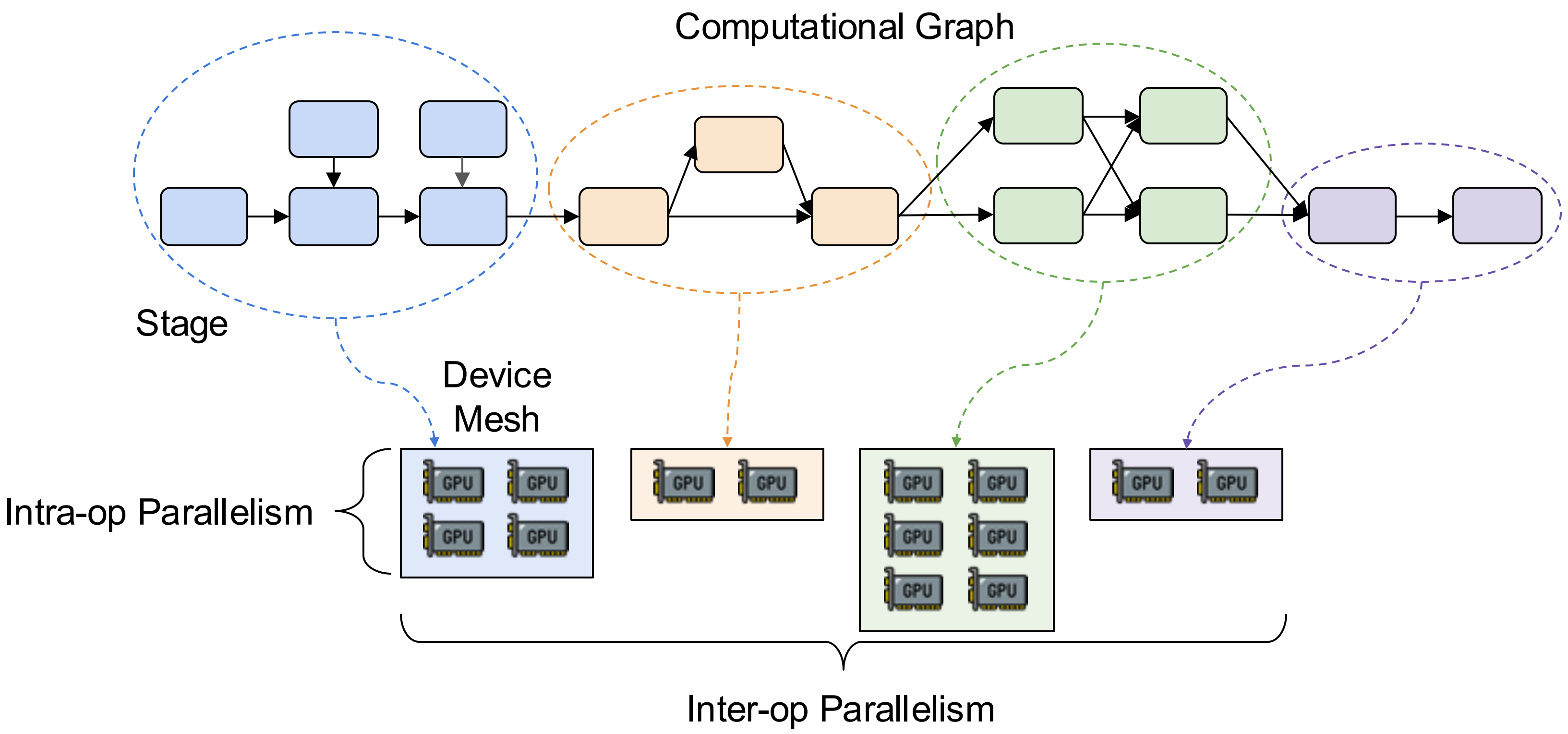
Intra-op Parallelism

Inter-op Parallelism
(w/o pipeline)

Simulator + MCMC for finding parallel strategies
- Details will be discussed later

Jia, Zhihao, Matei Zaharia, and Alex Aiken. "Beyond Data and Model Parallelism for Deep Neural Networks." *MLSys 2019*

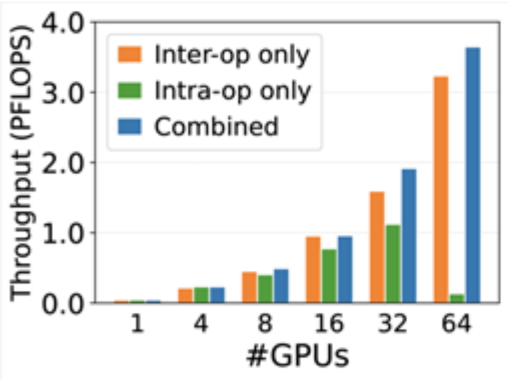
Combine Intra-op Parallelism and Inter-op Parallelism



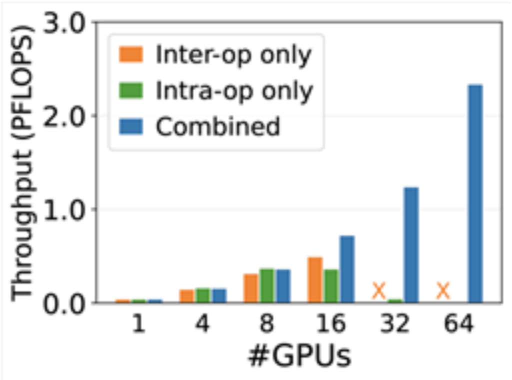
Narayanan, Deepak, et al. "Efficient large-scale language model training on gpu clusters using megatron-lm." *SC 2021*
Zheng, Lianmin, et al. "Alpa: Automating Inter-and Intra-Operator Parallelism for Distributed Deep Learning." *OSDI 2022*

Combine Intra-op Parallelism and Inter-op Parallelism

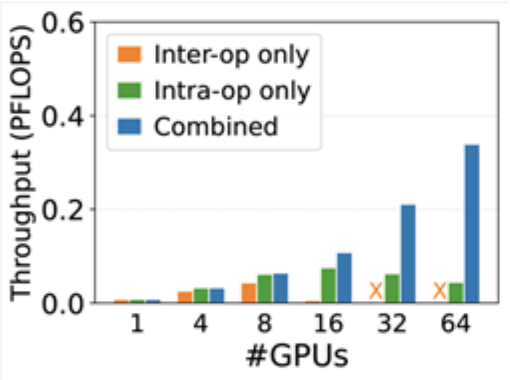
GPT



GShard MoE



Wide-ResNet



Combining inter- and intra-operator parallelism scales to more devices.

Zheng, Lianmin, et al. "Alpa: Automating Inter-and Intra-Operator Parallelism for Distributed Deep Learning." *OSDI 2022*

Intra-operator Parallelism Summary

- We can parallelize a single operator by exploiting its internal parallelism
- To do this for a whole computational graph, we need to choose strategies for all nodes in the graph to minimize the communication cost
- Intra-op and inter-op can be combined

Other Techniques for Training Large Models

System-level Memory Optimizations

- Rematerialization/Gradient Checkpointing
- Swapping

ML-level Optimizations

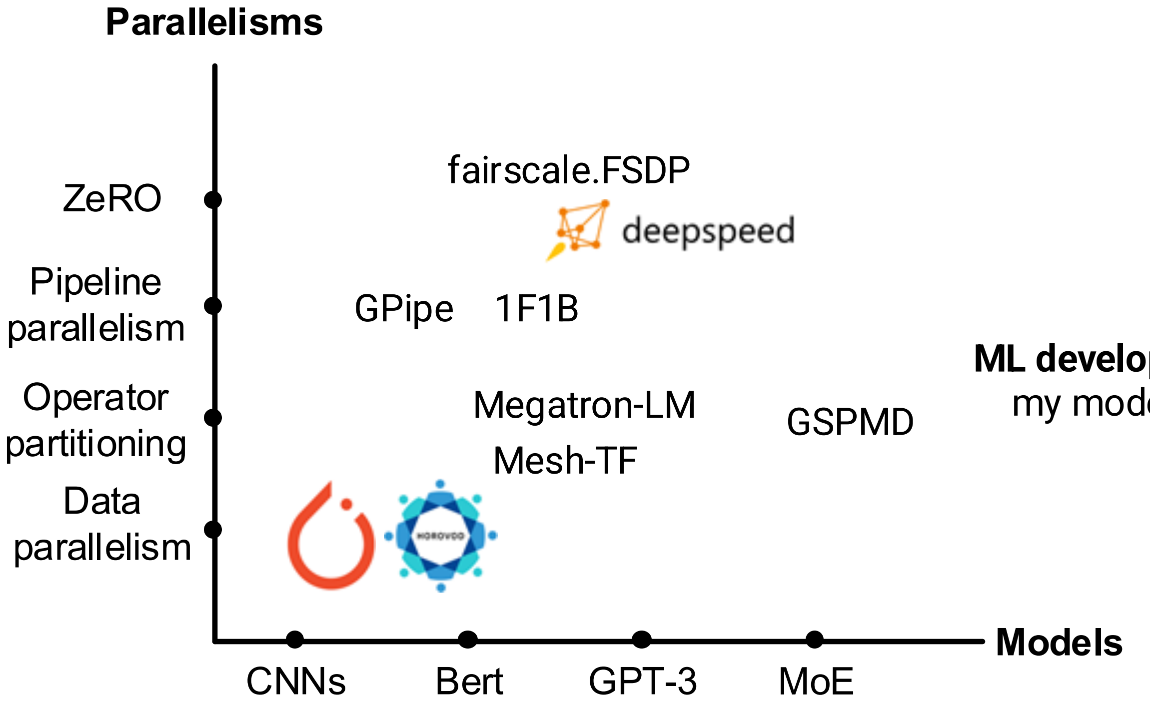
- Quantization
- Sparsification
- Low-rank approximation

Chen, Tianqi, et al. "Training deep nets with sublinear memory cost." *arXiv 2016*
Rajbhandari, Samyam, et al. "Zero-infinity: Breaking the gpu memory wall for extreme scale deep learning." *SC* 2021.
Tang, Hanlin, et al. "1-bit adam: Communication efficient large-scale training with adam's convergence speed." *ICML* 2021.
Shazeer, Noam, and Mitchell Stern. "Adafactor: Adaptive learning rates with sublinear memory cost." *ICML* 2018.

Where We Are

- Motivation
- History
- Parallelism Overview
- Data parallelism
- Model parallelism
    - Inter-op parallelism
    - Intra-op parallelism
- **Auto-parallelization**

Auto-parallelization: Motivation

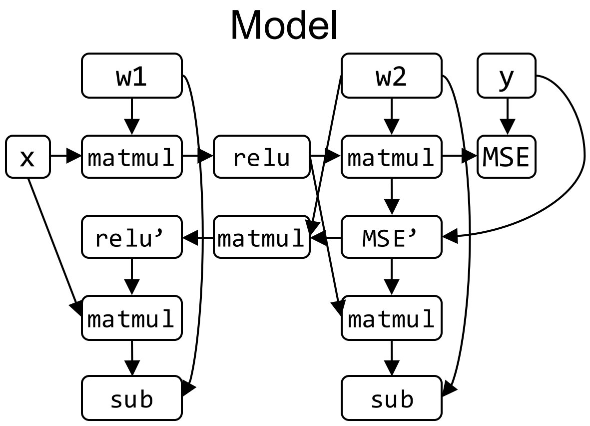


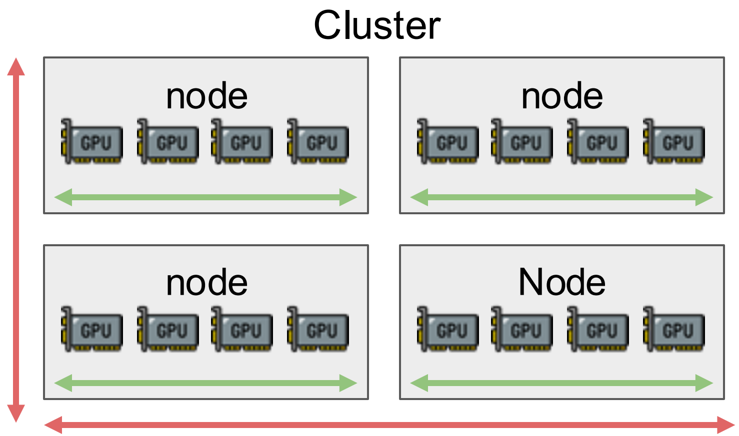
ML developer: which one is for my model and my cluster?

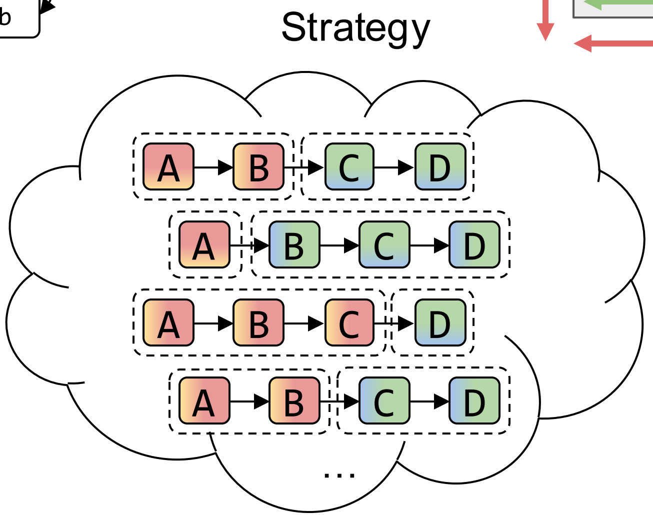
Auto-parallelization: Problem

$$\max_{\text{strategy}} \ \text{Performance}(\text{Model, Cluster})$$
$$s.t. \ \text{strategy} \in \text{Inter-op} \cup \text{Intra-op}$$

Auto-parallelization: Problem







The Search Space is Huge

#ops in a real model (nodes to color)	#op types (type of nodes)	#devices on a cluster (available colors)
100 - 10K	**80 - 200+**	**10s - 1000s**

Automatic Parallelization Methods

Search-based methods

- MCMC:
  - ➔ [Jia et al., 2018]
  - ➔ [Jia et al., 2019]
- Heuristics
  - ➔ [Fan et al., 2021]

The complete list of references is available on the tutorial website

Learning-based methods

- Reinforcement Learning:
  - ➔ [Mirhoseini et al., 2017]
  - ➔ [Mirhoseini et al., 2018]
  - ➔ [Addanki, et al., 2019]
- ML-based cost model:
  - ➔ [Chen et al., 2018],
  - ➔ [Zhou et al., 2020],
  - ➔ [Zhang, 2020]
- Bayesian optimization:
  - ➔ [Sergeev et al., 2018],
  - ➔ [Peng et al., 2019]

Optimization-based methods

- Dynamic programming
  - ➔ [Wang, et al., 2018]
  - ➔ [Narayanan, et al., 2019]
  - ➔ [Li, et al., 2021]
  - ➔ [Narayanan, et al., 2012]
  - ➔ [Tarnawski, et al., 2020]
  - ➔ [Tarnawski, et al., 2021]
- Integer linear programming
  - ➔ [Tarnawski, et al., 2020]
- Hierarchical Optimization
  - ➔ [Zheng, et al., 2022]

Automatic Parallelization Methods

Search-based methods

- MCMC:
  - ➔ [Jia et al., 2018]
  - ➔ [Jia et al., 2019]
- Heuristics
  - ➔ [Fan et al., 2021]

The complete list of references is available on the tutorial website

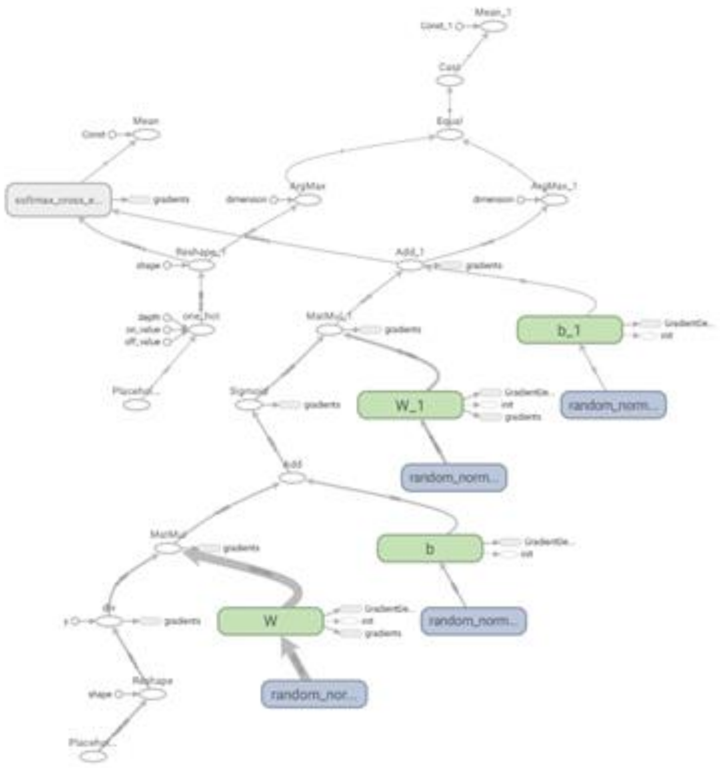
Learning-based methods

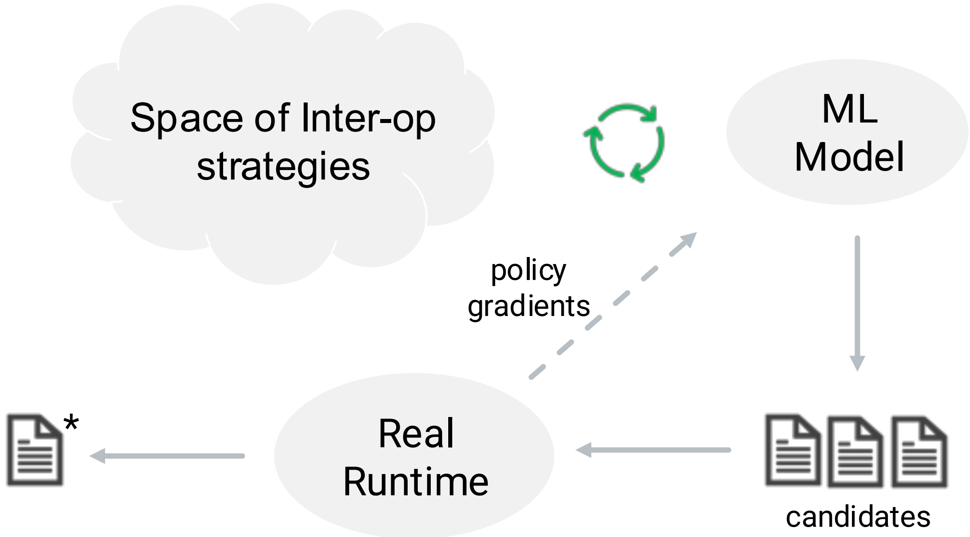
- Reinforcement Learning:
  - ➔ **[Mirhoseini et al., 2017]**
  - ➔ [Mirhoseini et al., 2018]
  - ➔ [Addanki, et al., 2019]
- ML-based cost model:
  - ➔ [Chen et al., 2018],
  - ➔ [Zhou et al., 2020],
  - ➔ [Zhang, 2020]
- Bayesian optimization:
  - ➔ [Sergeev et al., 2018],
  - ➔ [Peng et al., 2019]

Optimization-based methods

- Dynamic programming
  - ➔ [Wang, et al., 2018]
  - ➔ [Narayanan, et al., 2019]
  - ➔ [Li, et al., 2021]
  - ➔ [Narayanan, et al., 2012]
  - ➔ [Tarnawski, et al., 2020]
  - ➔ [Tarnawski, et al., 2021]
- Integer linear programming
  - ➔ [Tarnawski, et al., 2020]
- Hierarchical optimization
  - ➔ **[Zheng, et al., 2022]**

ColocRL (a.k.a. Device Placement Optimization)





Mirhoseini, et al. "Device Placement Optimization with Reinforcement Learning." *ICML* 2017.

ColocRL: Model

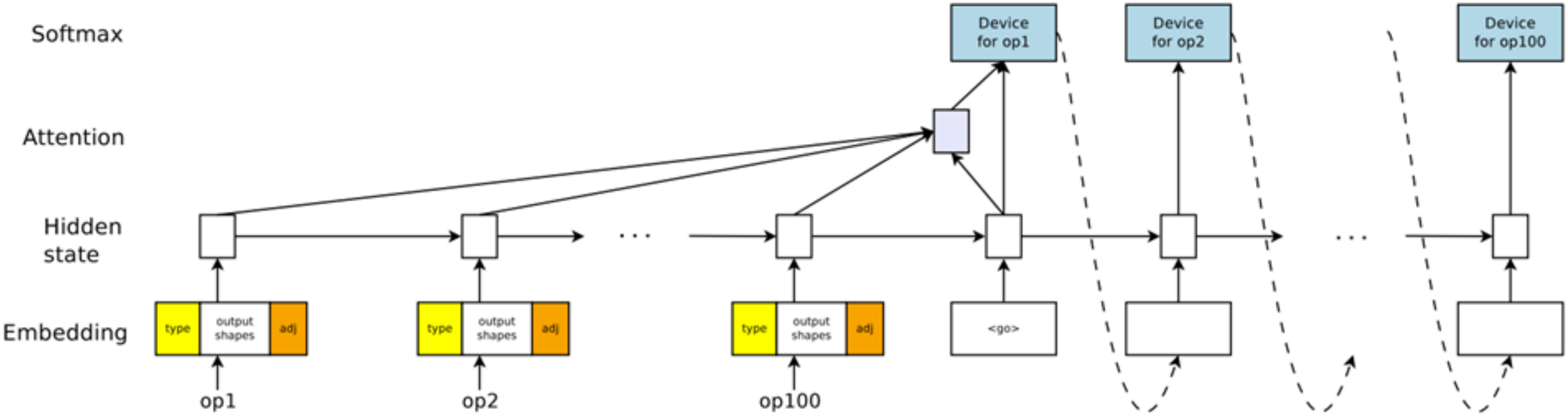


Figure from [Mirhoseini et al., ICML 2017]

Mirhoseini, et al. "Device Placement Optimization with Reinforcement Learning." *ICML* 2017.

ColocRL: Training

$$\mathbb{E}_{\mathcal{P}\sim\pi(\mathcal{P}\mid\mathcal{G};\theta)}[R(\mathcal{P})||\mathcal{G}]$$

$\mathcal{G}$: computational graph

$\mathcal{R}(\mathcal{P})$: Real runtime of a placement

$\pi(\cdot)$: output distributed of the RNN

Mirhoseini, et al. "Device Placement Optimization with Reinforcement Learning." *ICML* 2017.

ColocRL: Other Improvement

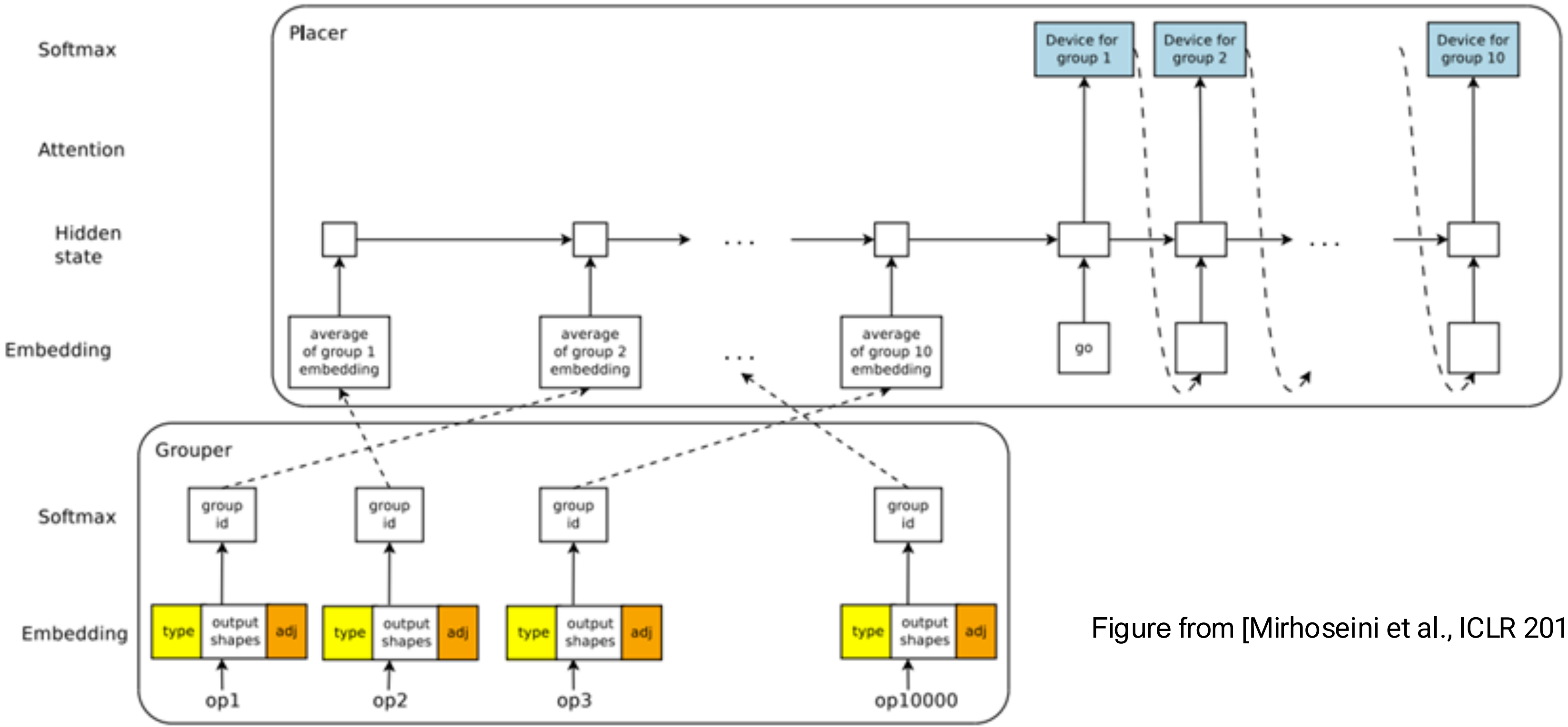
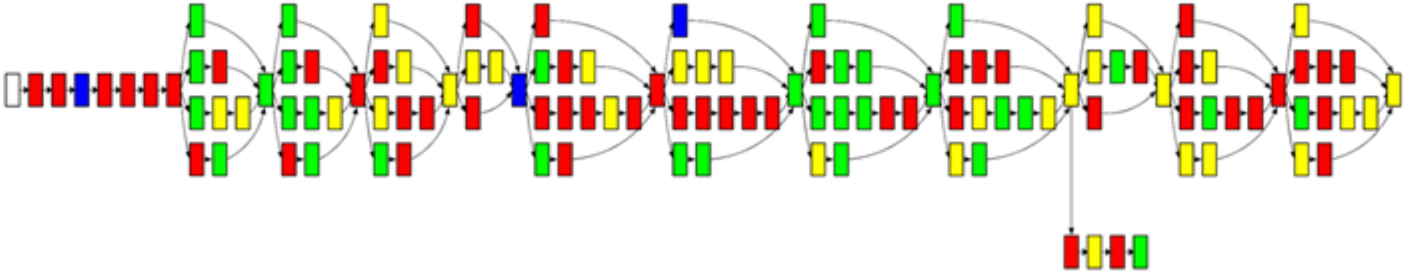


Figure from [Mirhoseini et al., ICLR 2018]

Mirhoseini, et al. "A Hierarchical Model for Device Placement." *ICLR* 2018.

Results Discussion



Tasks	Single-CPU	Single-GPU	#GPUs	Scotch	MinCut	Expert	RL-based	Speedup
RNNLM (batch 64)	6.89	**1.57**	2	13.43	11.94	3.81	**1.57**	0.0%
			4	11.52	10.44	4.46	**1.57**	0.0%
NMT (batch 64)	10.72	OOM	2	14.19	11.54	4.99	**4.04**	23.5%
			4	11.23	11.78	4.73	**3.92**	20.6%
Inception-V3 (batch 32)	26.21	**4.60**	2	25.24	22.88	11.22	**4.60**	0.0%
			4	23.41	24.52	10.65	**3.85**	19.0%

Figure and table from [Mirhoseini et al., ICML 2017]

Automatic Parallelization Methods

Search-based methods

- MCMC:
  - ➔ [Jia et al., 2018]
  - ➔ [Jia et al., 2019]
- Heuristics
  - ➔ [Fan et al., 2021]

Learning-based methods

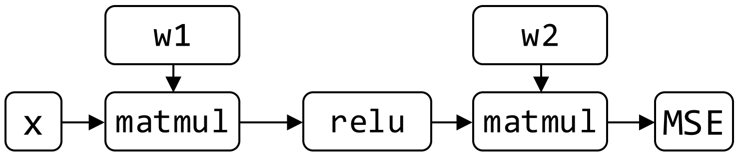
- Reinforcement Learning:
  - ➔ [Mirhoseini et al., 2017]
  - ➔ [Mirhoseini et al., 2018]
  - ➔ [Addanki, et al., 2019]
- ML-based cost model:
  - ➔ [Chen et al., 2018],
  - ➔ [Zhou et al., 2020],
  - ➔ [Zhang, 2020]
- Bayesian optimization:
  - ➔ [Sergeev et al., 2018],
  - ➔ [Peng et al., 2019]

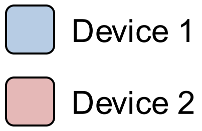
Optimization-based methods

- Dynamic programming
  - ➔ [Wang, et al., 2018]
  - ➔ [Narayanan, et al., 2019]
  - ➔ [Li, et al., 2021]
  - ➔ [Narayanan, et al., 2012]
  - ➔ [Tarnawski, et al., 2020]
  - ➔ [Tarnawski, et al., 2021]
- Integer linear programming
  - ➔ [Tarnawski, et al., 2020]
- Hierarchical optimization
  - ➔ **Alpa [Zheng, et al., 2022]**

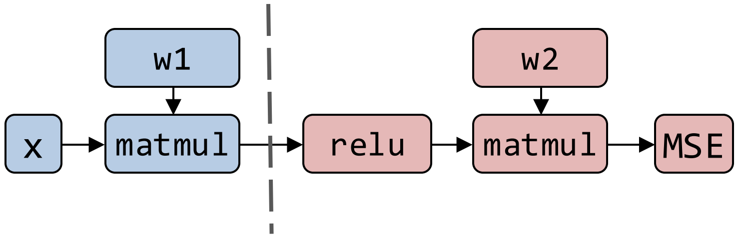
The complete list of references is available on the tutorial website

Optimization-based Method: Alpa

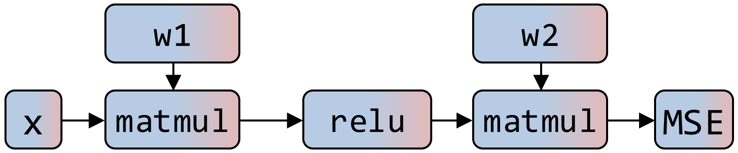




Inter-op parallelism



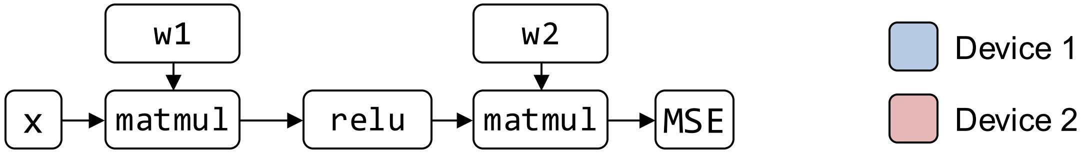
Intra-op parallelism



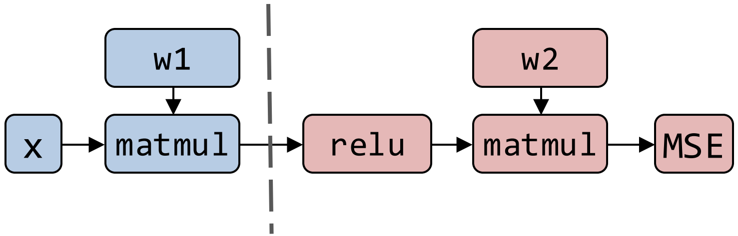
Trade-off

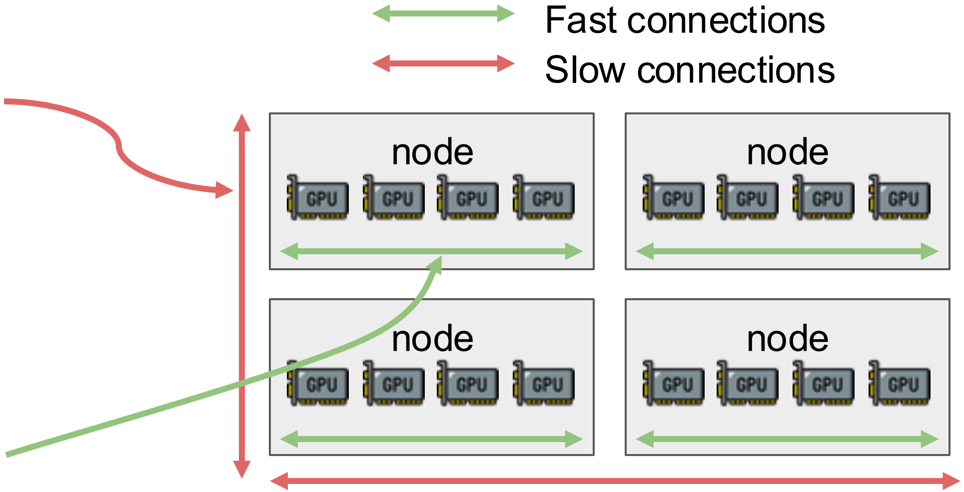
	Inter-operator Parallelism	**Intra-operator Parallelism**
Communication	Less	More
Device Idle Time	More	Less

Alpa Rationale

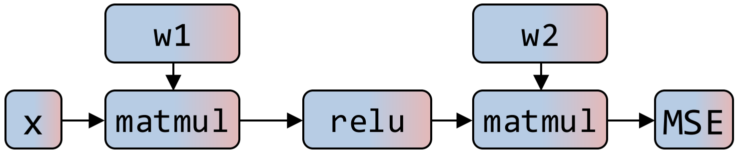


Inter-op parallelism



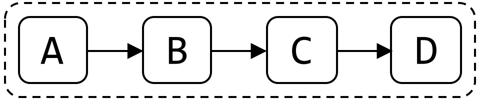


Intra-op parallelism



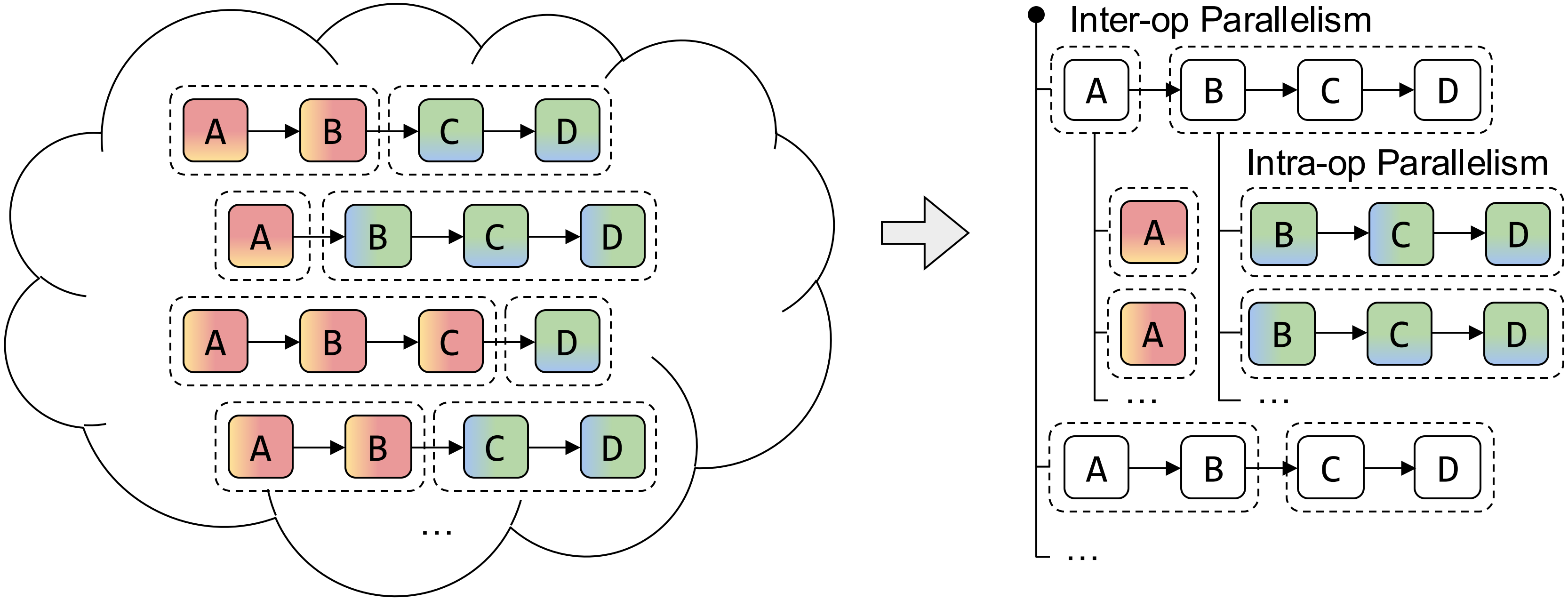
Search Space

Computational Graph

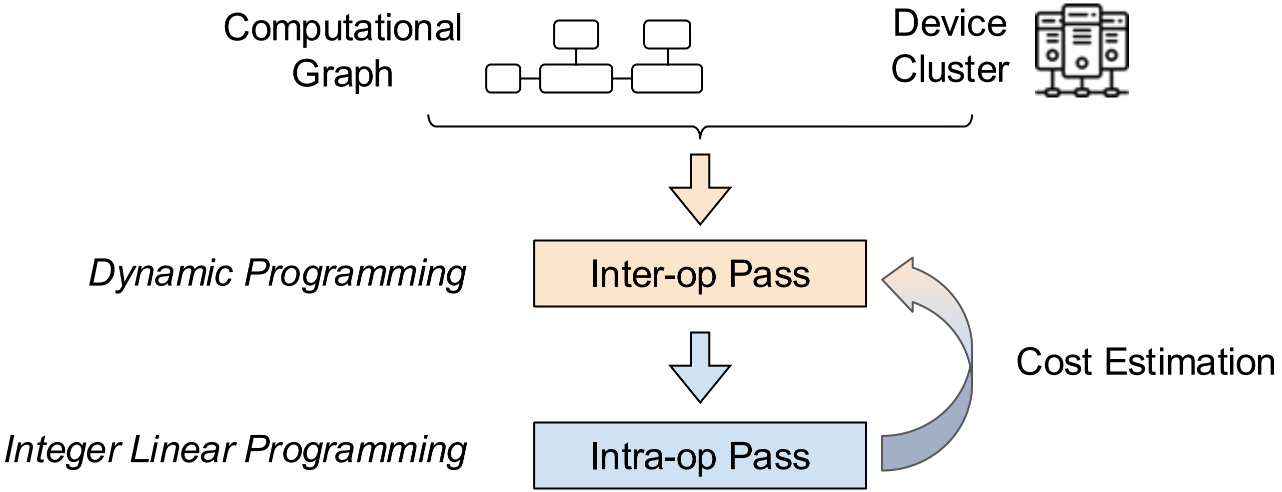


Whole Search Space

Alpa Hierarchical Space

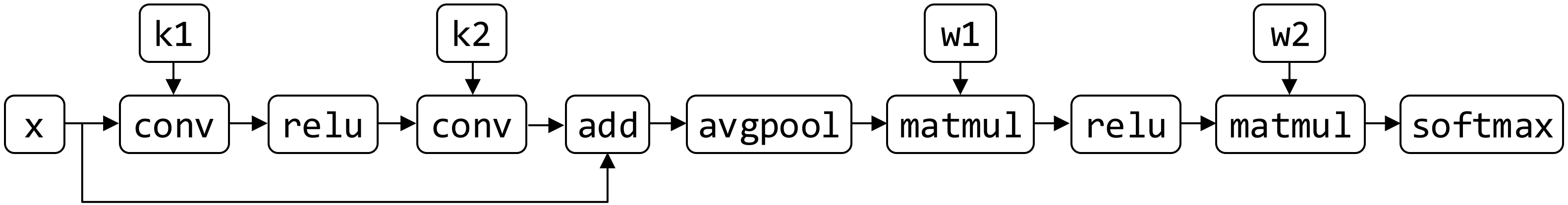


Alpa Compiler: Hierarchical Optimization

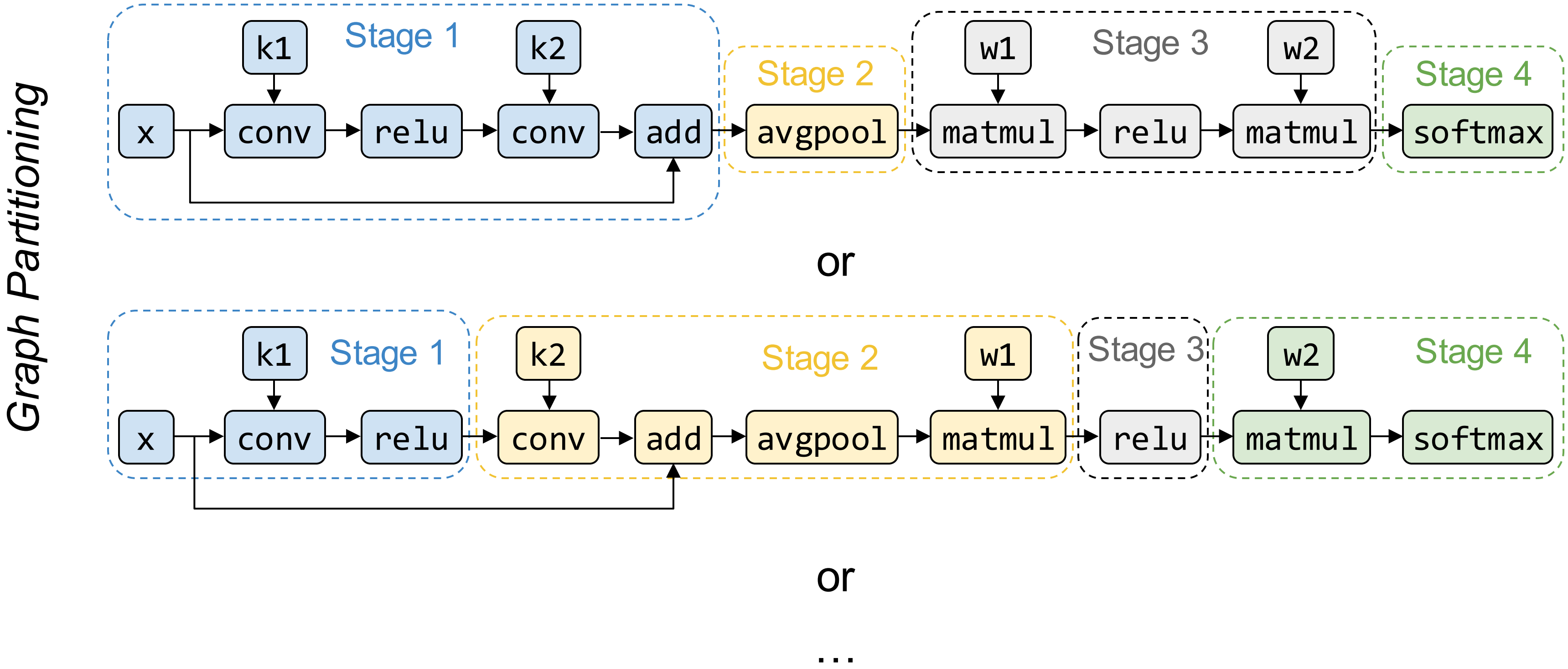


Inter-op Pass

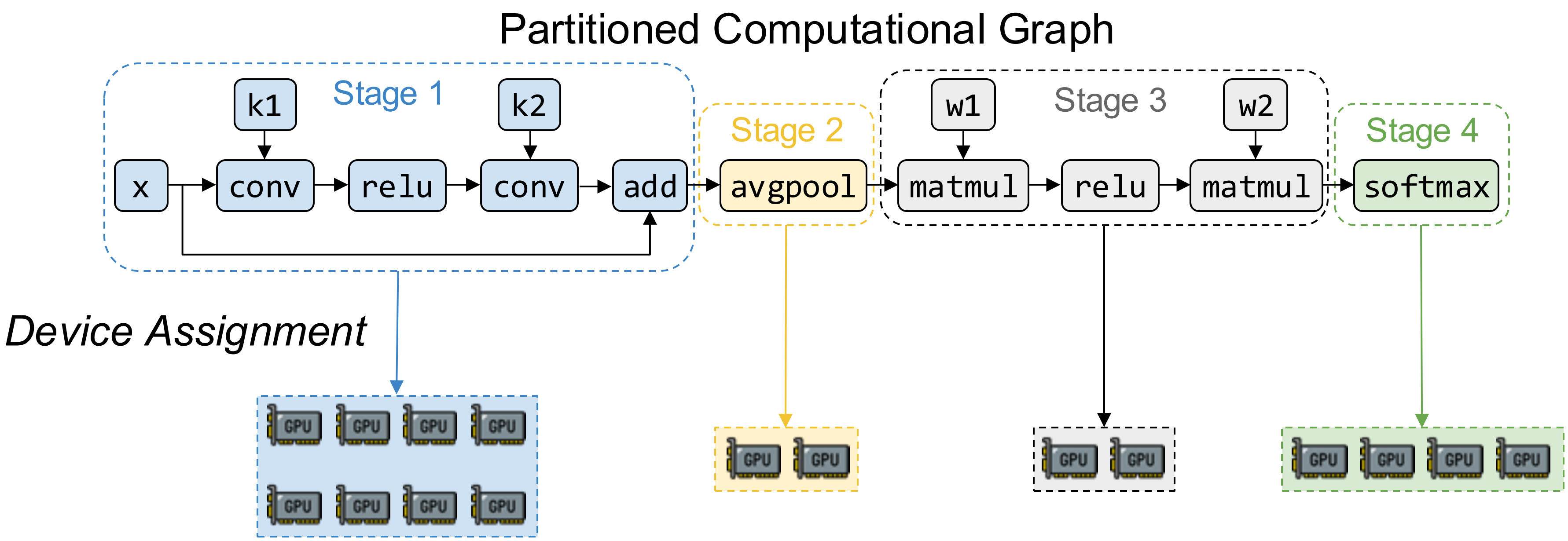
Computational Graph



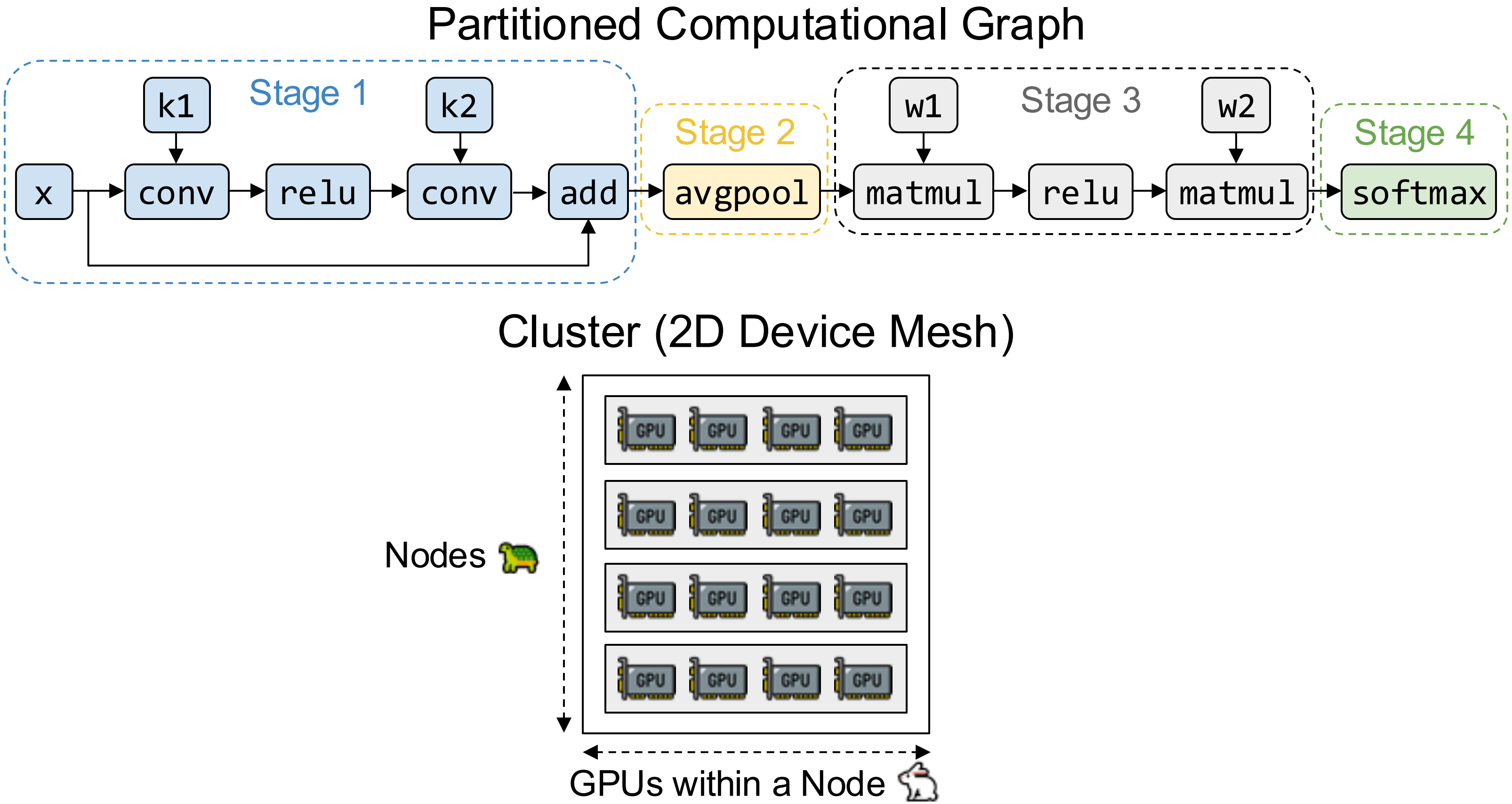
Inter-op Pass



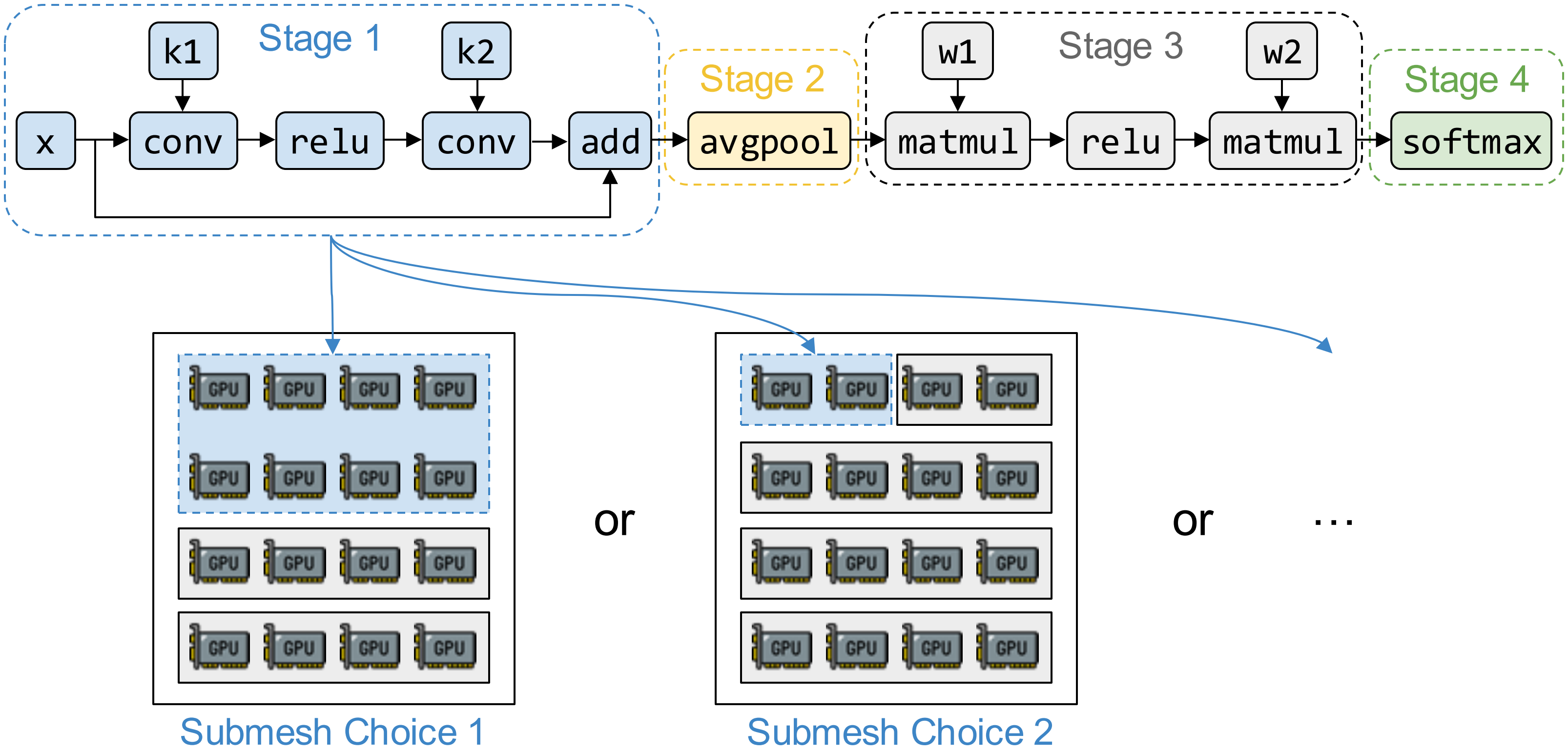
Inter-op Pass



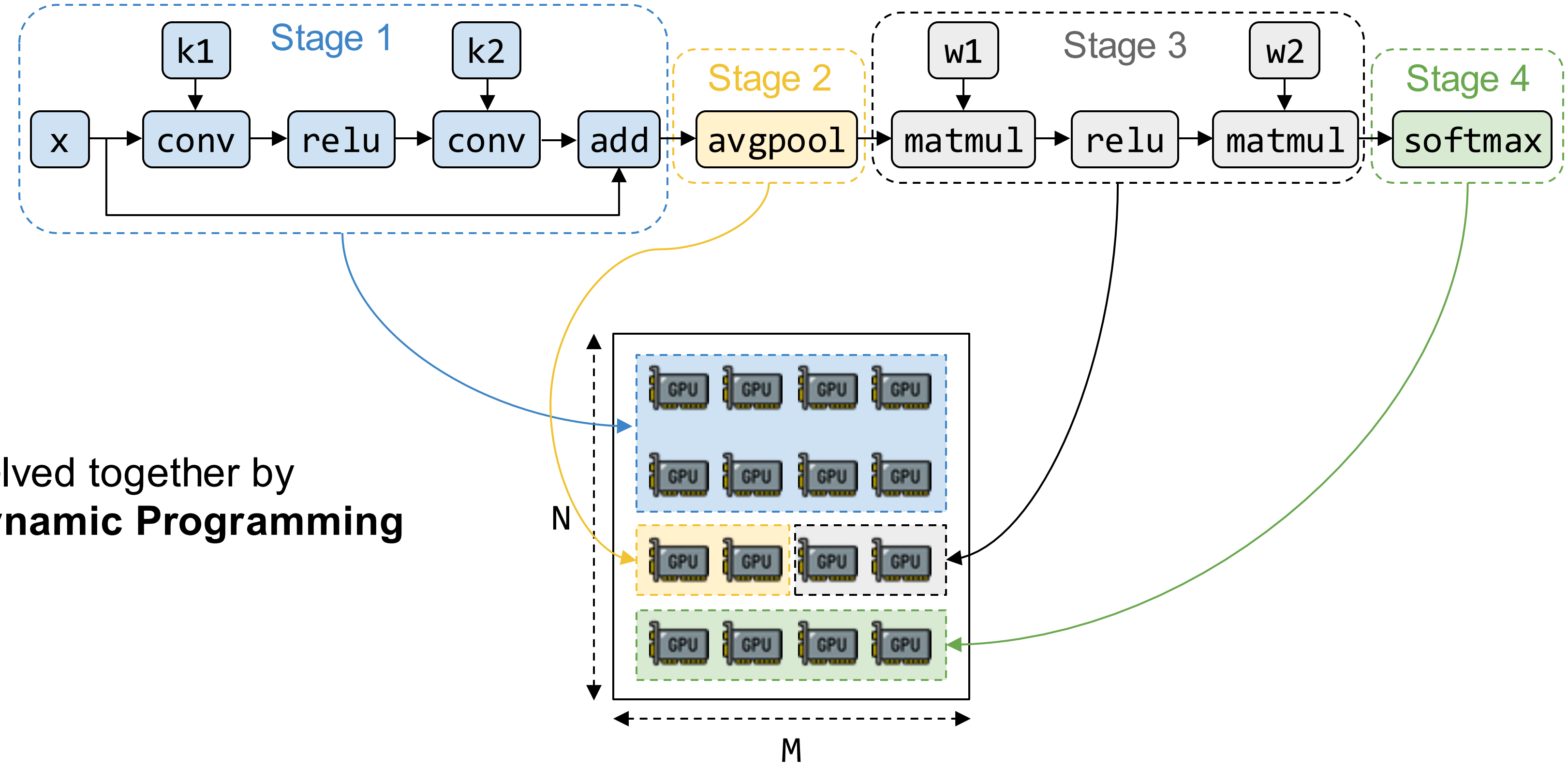
Inter-op Pass



Inter-op Pass

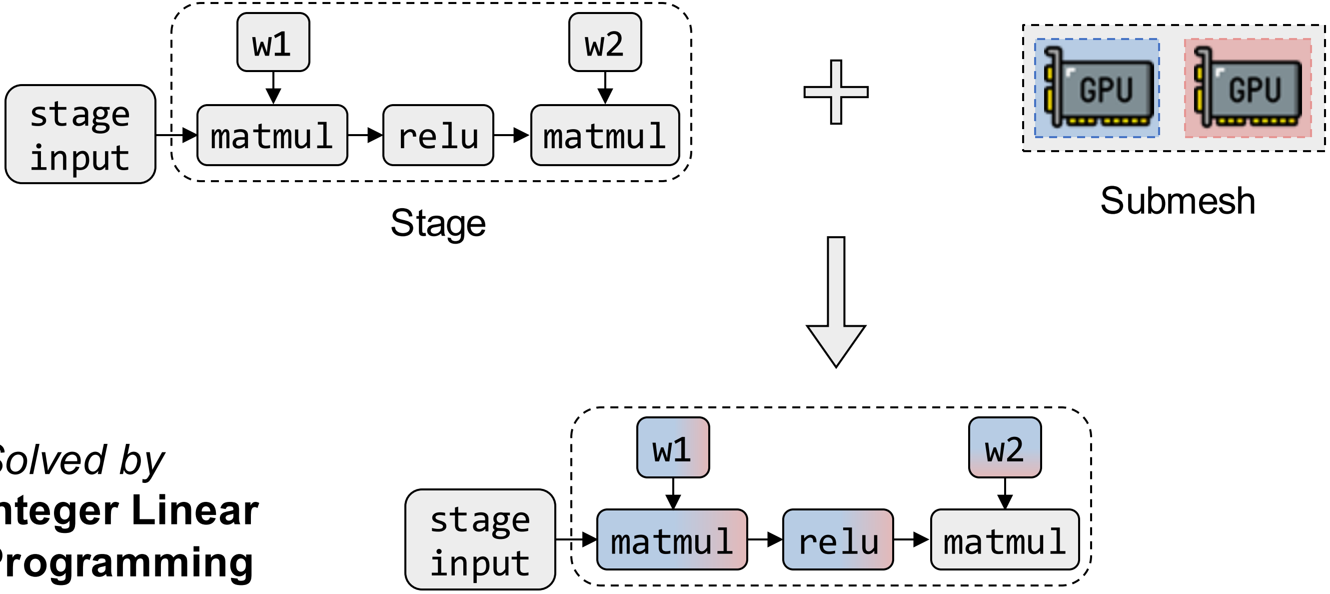


Inter-op Pass



Solved together by
Dynamic Programming

Intra-op Pass



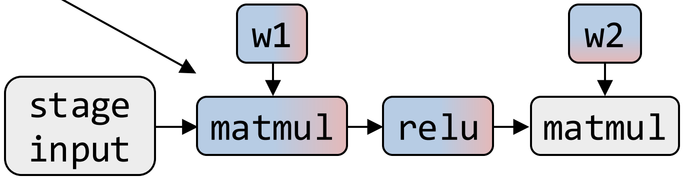
Solved by
Integer Linear Programming

Stage with intra-operator parallelization

Intra-op Pass

Integer Linear Programming Formulation

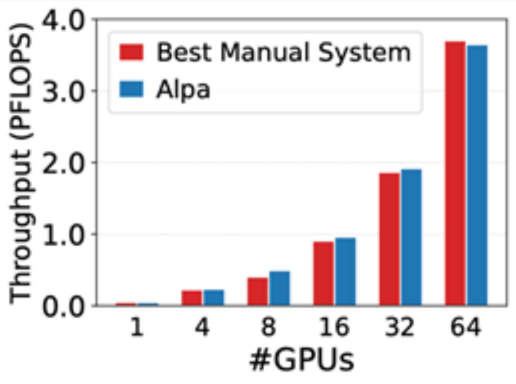
Decision vector
Parallel strategies of each operator



Minimize Computation cost + Communication cost

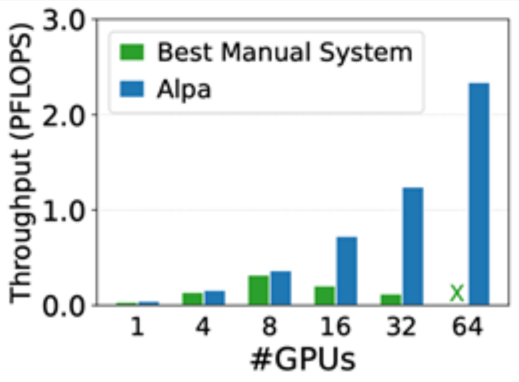
Evaluation: Comparing with Previous Works

GPT (up to 39B)



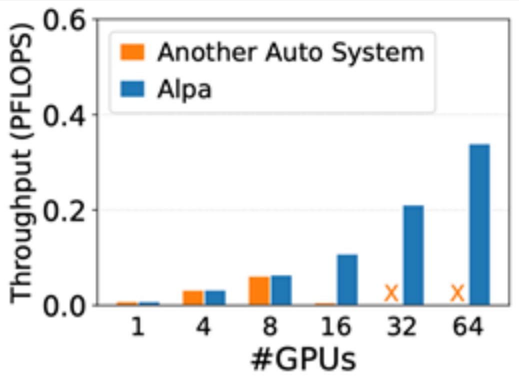
Match specialized manual systems.

GShard MoE (up to 70B)



Outperform the manual baseline by up to 8x.

Wide-ResNet (up to 13B)



Generalize to models without manual plans.

Weak scaling results where the model size grow with #GPUs.
Evaluated on 8 AWS EC2 p3.16xlarge nodes with 8 16GB V100s each (64 GPUs in total).

Automatic Parallelization Methods

Search-based methods

- ✅ Easy to extend the search space
- ✅ No training cost
- ❌ High inference cost
- ❌ Not explainable
- ❌ No optimality guarantee

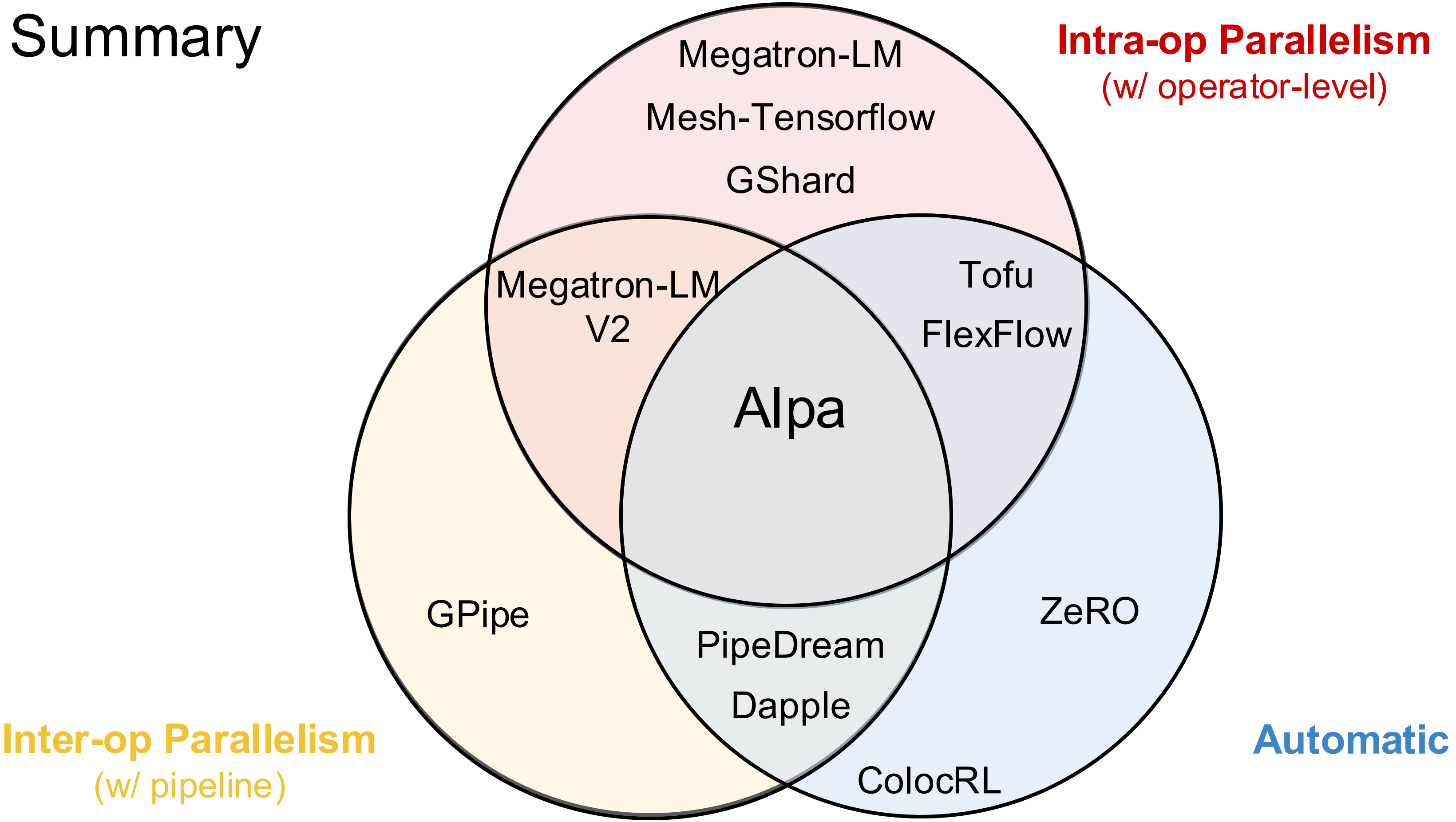
Learning-based methods

- ✅ Easy to extend the search space
- ❌ High training cost
- ✅ Low inference cost
- ❌ Not explainable
- ❌ No optimality guarantee

Optimization-based methods

- ❌ Non-trivial to extend the search space
- ✅ No training cost
- ✅ Medium inference cost
- ✅ Explainable
- ✅ Some optimality guarantee

Summary


Intra-op Parallelism
(w/ operator-level)
Inter-op Parallelism
(w/ pipeline)
Automatic
Megatron-LM
Mesh-Tensorflow
GShard
Megatron-LM V2
Tofu
FlexFlow
Alpa
GPipe
PipeDream
Dapple
ZeRO
ColocRL