2022 Fall @ POSTECH

EECE695D: Efficient ML Systems

Distributed Training

(part 1)

Final Report. (Due: December 23rd, 23:59PM via PLMS)

- Pick a subject that is relevant to efficient ML
  (preferrably relevant to your research topic).

- Read 3~5 paper published in 2020~2022.
  (Don’t randomly pick papers; define a very narrow subtopic)

- Give a 3-page survey in ICML LaTeX format (https://icml.cc)

  - Max 3-page content, plus unlimited references.

  - Explain what each paper does, and combine them to give a nice storyline.

  - Must include the “potential future research direction”

- Example: https://arxiv.org/pdf/2004.05439.pdf

Distributed Training

What's that? Training a model using more than one computing devices (a.k.a., workers), e.g., GPUs

Why? There are several reasons...

- *Compute too large:* Training the largest GPT-3 requires 3.14×10^{23} FLOPs
  NVIDIA H100 SXM can process 6.7×10^{13} FLOP per second $\Rightarrow$ takes ≈ 150 years
  (Data parallelism)

- *Model too large:* The largest GPT-3 has 175B parameters ≈ 350 GB if all are 16bit!
  NVIDIA H100 has 80GBs of memory.
  (Model parallelism)

- *Data too large:* Private data, or just too large to store at a single place.
  (Critical consideration in "Federated Learning")

- Other issues, e.g., environmental considerations...

Key Challenges. Distributed training is difficult, and we don't get 100 times faster with 100 GPUs, because...

- Usually requires a careful coordination/scheduling of a central server.
  (rarely is fully decentralized; think about the communication cost!)
  (some devices could be very slower than others)

- Heavy communication between computing devices to transmit model/gradient information.
  (connected via InfiniBand? Remote location?)

- Latency / Communication budget / Data are heterogeneous from a GPU to GPU.

Data Parallelism

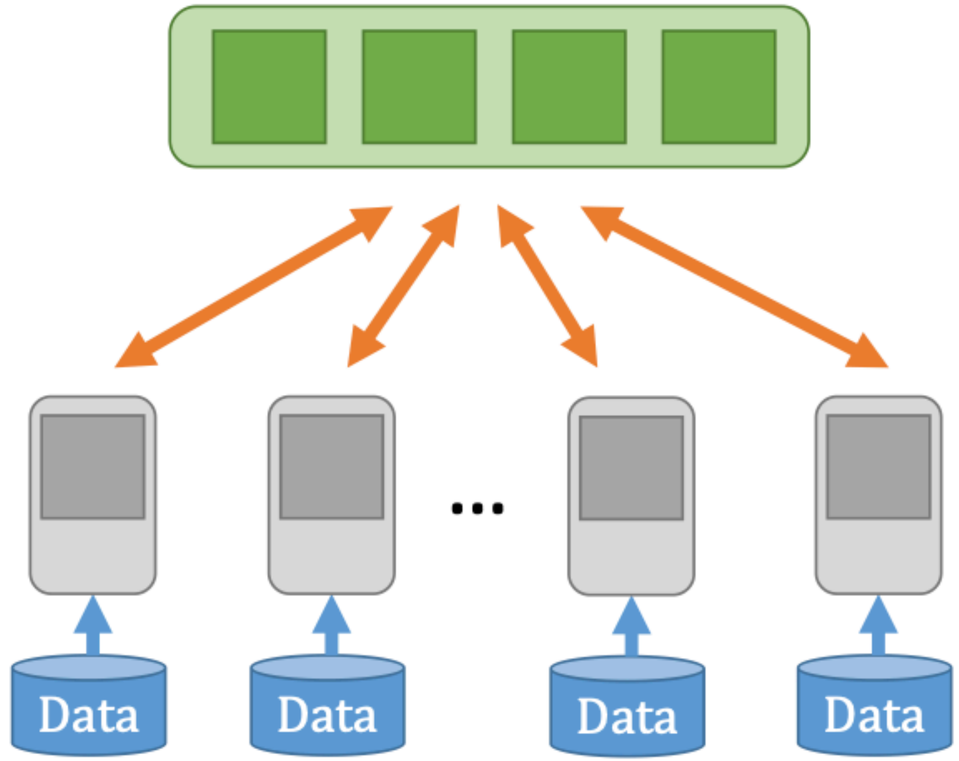
Copy the same parameter to multiple workers (i.e., GPUs), and assign different samples to each worker.

Here's a simplest way to implement this:

- At each iteration:
  - Master broadcasts the model param w.
  - Workers draw a data batch and compute the gradient $\nabla w^{(i)}$
  - Workers send $\nabla w^{(i)}$ to the parameter server.
  - Master averages the gradients to compute

$$w \leftarrow w - \alpha \cdot \left(\frac{1}{K} \sum_{i=1}^{K} \nabla w^{(i)} \right)$$

Parameter server (Master)
Coordinates all training



Workers
GPUs and data storage

Lin et al. (2018), "Deep gradient compression: Reducing the communication bandwidth for distributed training" ICLR 2018

Data Parallelism

Data. Usually evenly split into K workers—
but definitely possible for the workers to share the dataset,
(in which case master can broadcast data entries to train)
or dynamically fetching data during training.

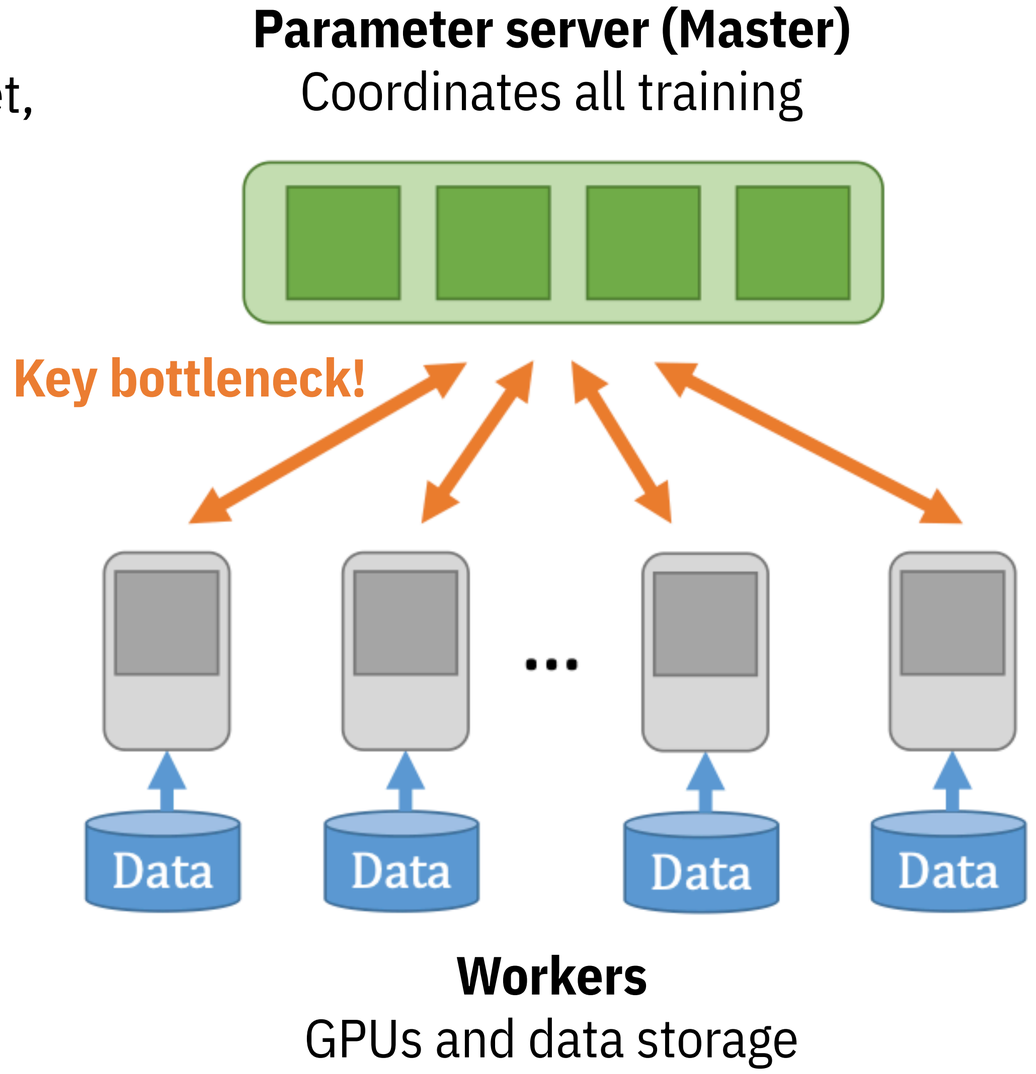
Communication cost.

- Master: $K \cdot (\text{gradient size})_{\text{out}} + K \cdot (\text{model size})_{\text{in}}$
- Worker: $(\text{gradient size})_{\text{in}} + (\text{model size})_{\text{out}}$

Example) Training ResNet50 on V100

ResNet50 has model size = 97.5MB
Each gradient compute takes 1/3 seconds on each GPU
(with batch size 32)

⇒ Training with 256 workers requires $73.1\ \text{GB/s}$ bandwidth,
both inward and outward!



Lin et al. (2018), "Deep gradient compression: Reducing the communication bandwidth for distributed training" ICLR 2018

Trick. Use better communication scheme than centralized communication.

There are some standards for distributed communication (e.g., Sockets, MPI, ...)

DISTRIBUTED COMMUNICATION PACKAGE - TORCH.DISTRIBUTED

• NOTE

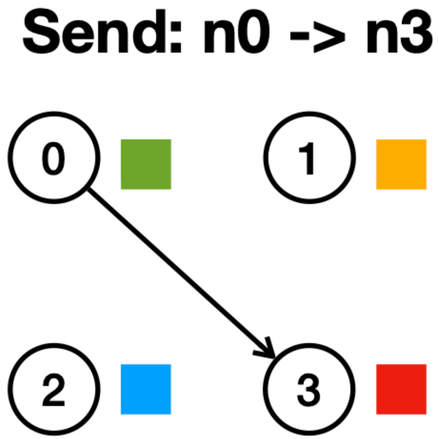
Please refer to PyTorch Distributed Overview for a brief introduction to all features related to distributed training.

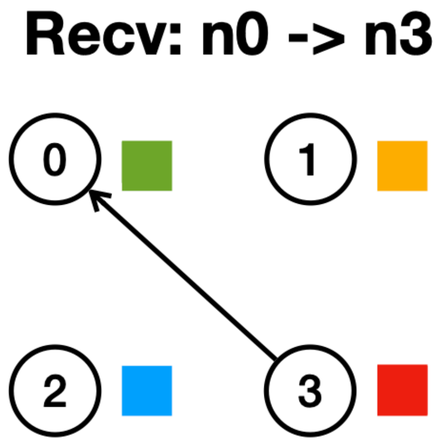
Backends

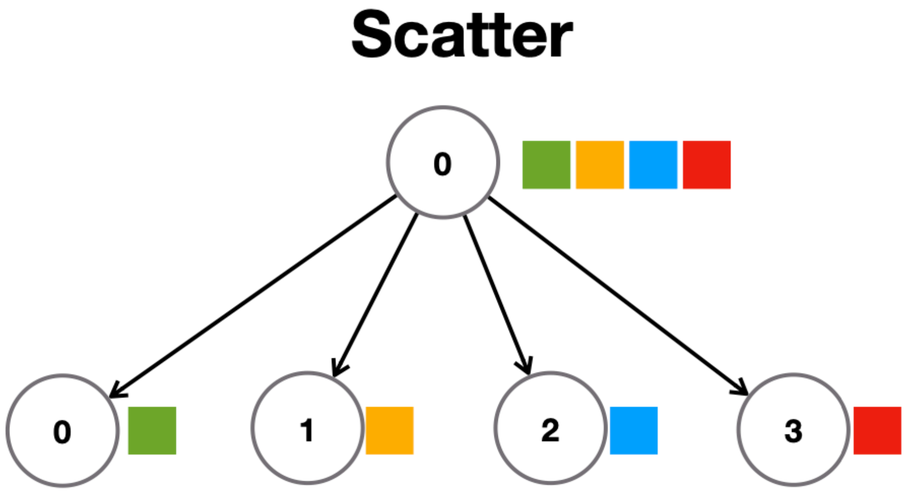
`torch.distributed` supports three built-in backends, each with different capabilities. The table below shows which functions are available for use with CPU / CUDA tensors. MPI supports CUDA only if the implementation used to build PyTorch supports it.

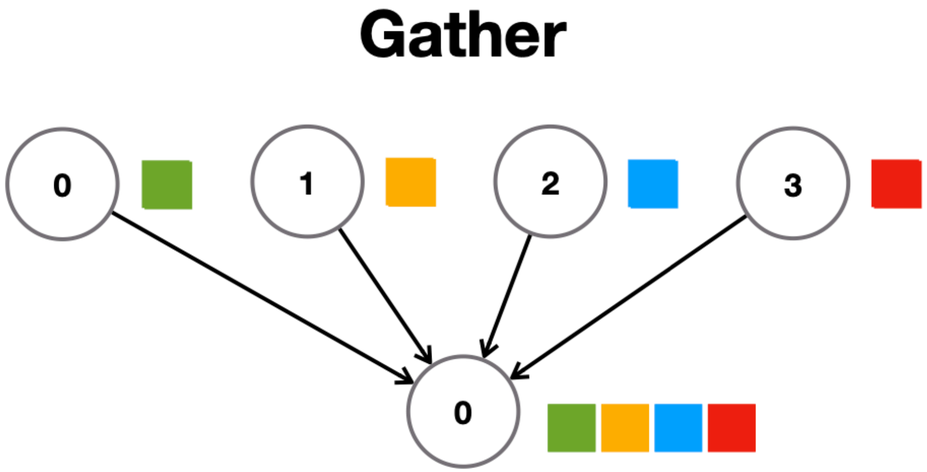
Trick. Use better communication scheme than centralized communication.

There are some standards for distributed communication (e.g., Sockets, MPI, ...)

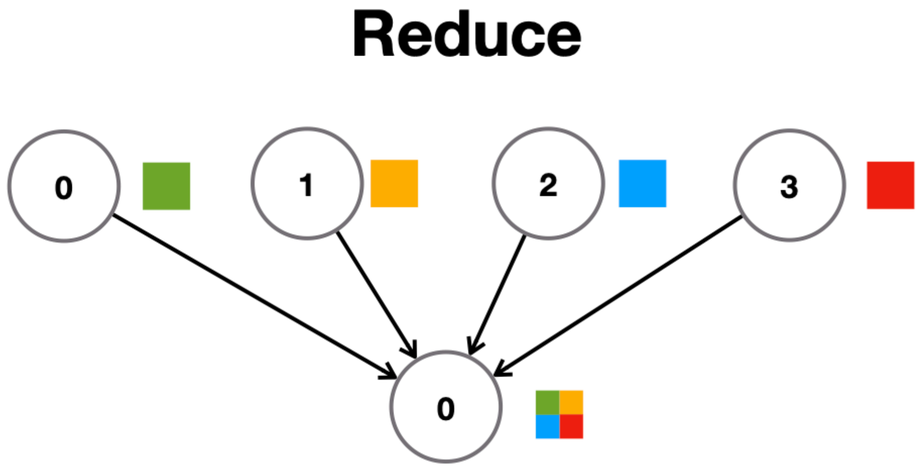


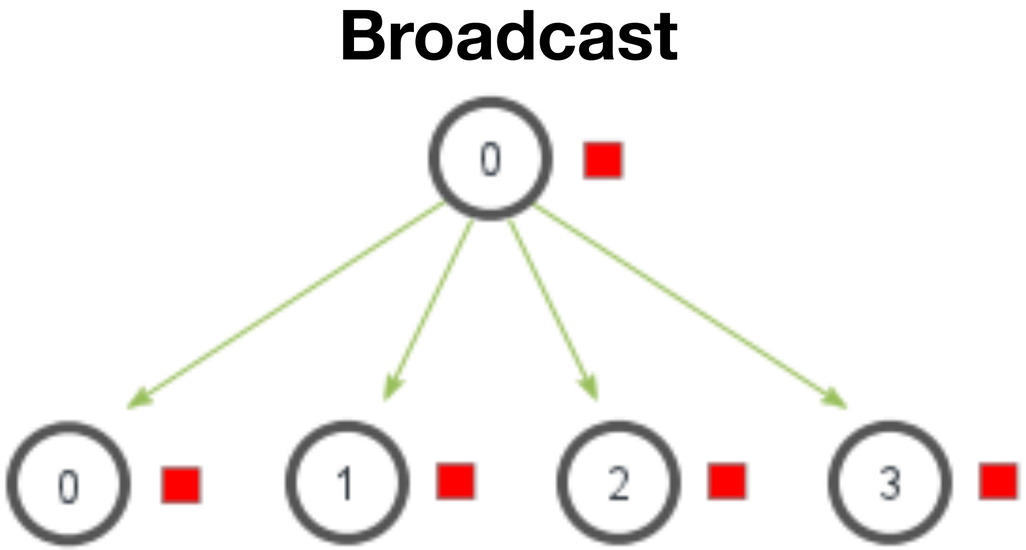


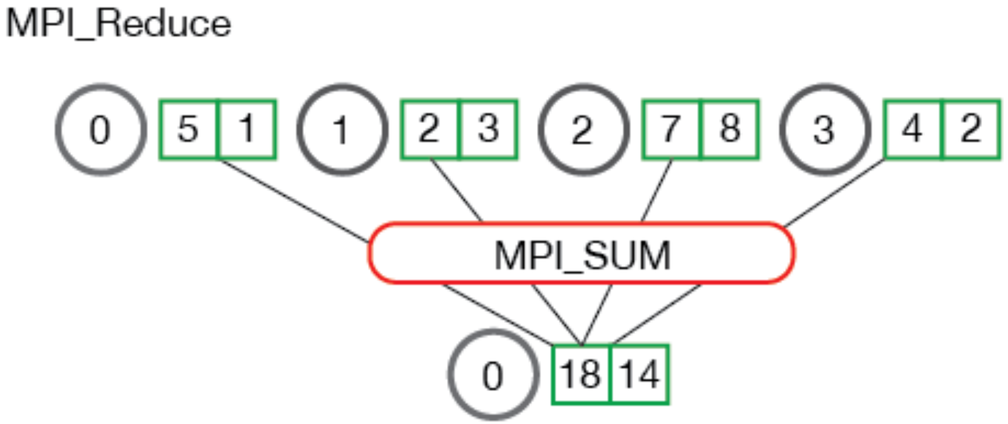




In the previous scenario, we were considering doing the “Reduce” operation, followed by “Broadcast”







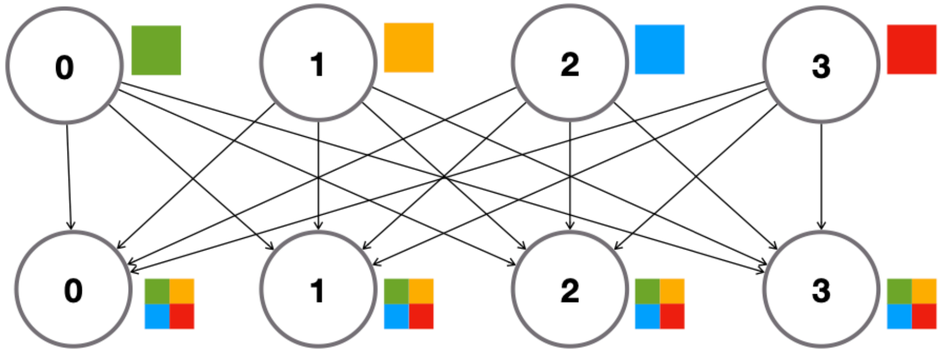
18 = 5 + 2 + 7 + 4
14 = 1 + 3 + 8 + 2

Time. $\mathcal{O}(1)$
Peak Bandwidth. $\mathcal{O}(K)$
Total Bandwidth. $\mathcal{O}(K)$

In fact, what we really wanted to do was so-called, "All-Reduce" operation.

There are many alternative ways to implement "All-Reduce" ...

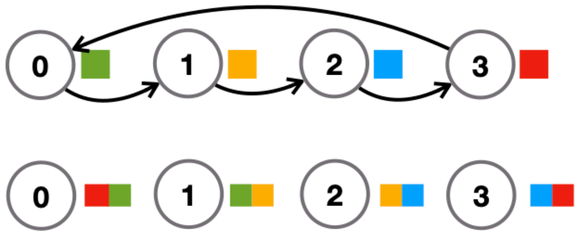
All-Reduce



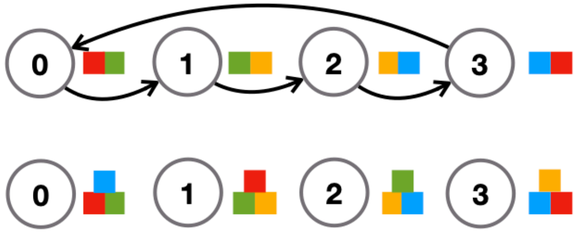
Time. $\mathcal{O}(1)$
Peak Bandwidth. $\mathcal{O}(K)$
Total Bandwidth. $\mathcal{O}(K^2)$

Ring All-Reduce

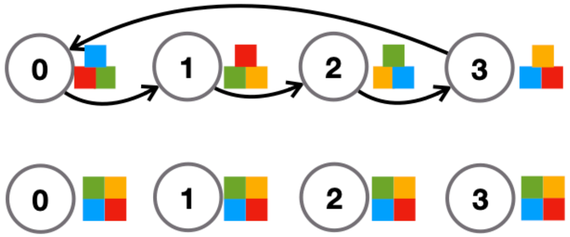
Step 1



Step 2

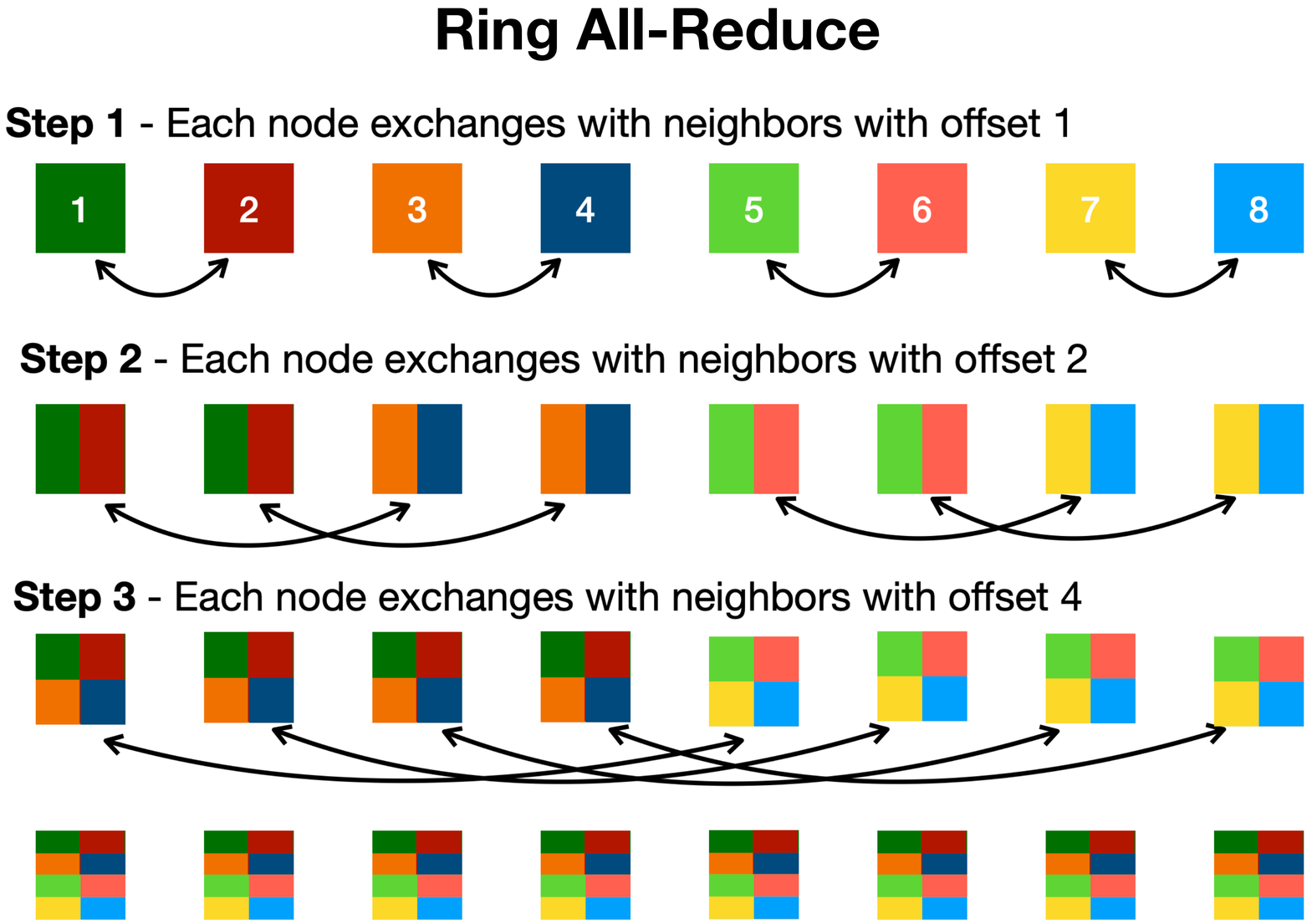


Step 3



Time. $\mathcal{O}(K)$
Peak Bandwidth. $\mathcal{O}(1)$
Total Bandwidth. $\mathcal{O}(K)$

A neat method is called “Recursive Having All-Reduce”



Time. $\mathcal{O}(\log K)$

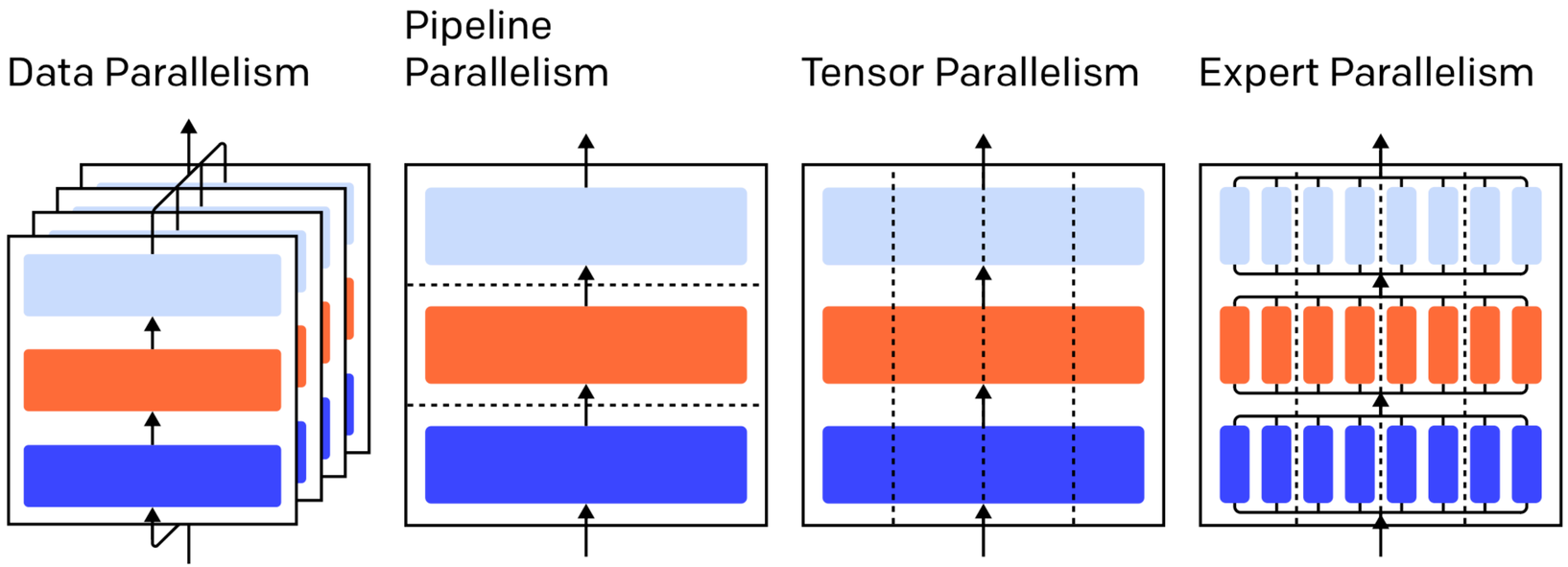
Peak Bandwidth. $\mathcal{O}(1)$

Total Bandwidth. $\mathcal{O}(K)$

Model Parallelism

Instead of spreading the data, we spread the model parameters across a set of workers.

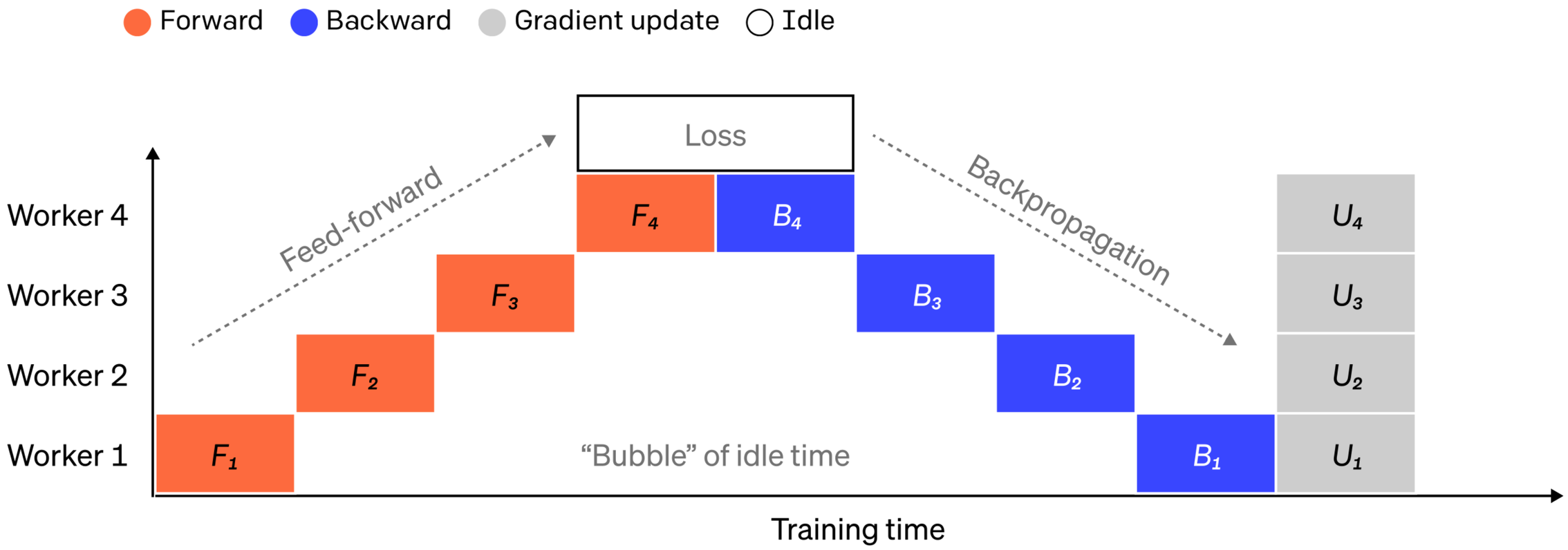
- *Pipeline Parallelism:* Data gets sequentially processed by GPU0, GPU1, GPU2, ... # No speedup?
- *Tensor Parallelism:* Layerwise tensors are divided into multiple workers.
- *Expert Parallelism:* Conditional computation, e.g., mixture-of-experts.



Pipeline Parallelism. Data gets sequentially processed by GPU0, GPU1, GPU2, ...

A caveat is that, when naïvely implemented, this does not lead to any speedup—
Not fully utilizing all workers, as workers need another worker's output as an input.

For instance, consider a four-layer model divided into four workers...
(Very rough figure—backward takes almost twice more time!)

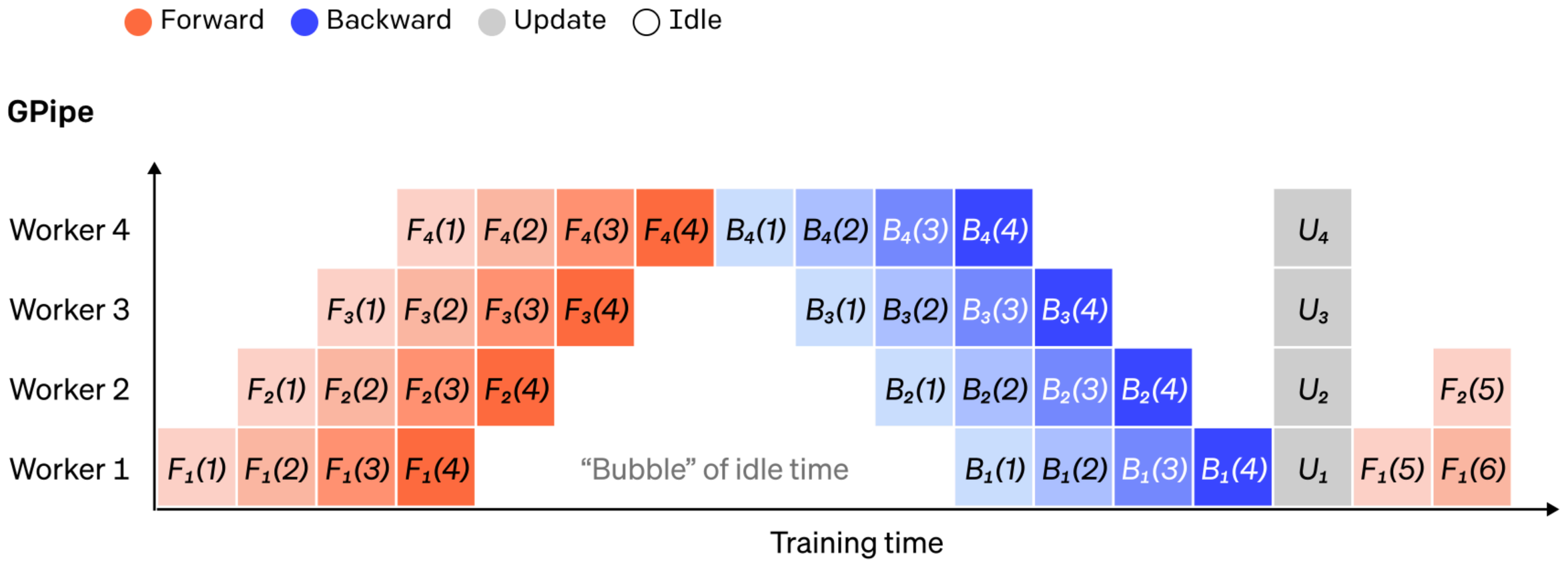


Solution. We can reuse aspects of data parallelism!

In other words, process the multiple micro-batch simultaneously and then update.

GPipe (Huang et al., 2019) have the workers do the forward and backward passes consecutively, and then aggregate gradients of each micro-batch to update at the end.

(Note: As we have seen long time ago, some tensors for backprop can be precomputed—these fill up the bubble, too.)



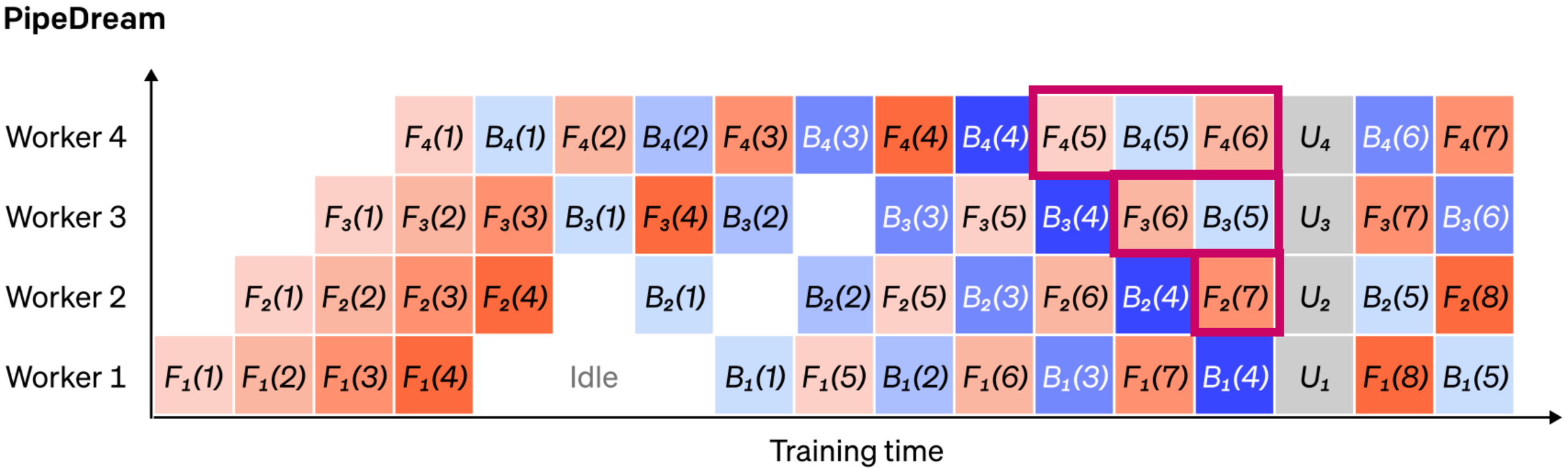
Huang et al. (2019), "GPipe: Efficient training of giant neural networks using pipeline parallelism" NeurIPS 2019.

PipeDream (Narayanan et al., 2019) brings this pipeline scheduling to a level of art.

They allow for an inter-batch pipelining as well—and do the scheduling automatically!

Note that some data of the second batch are processed before properly updating the tensors based on the first batch of data! (we call these outdated parameters “stale”).

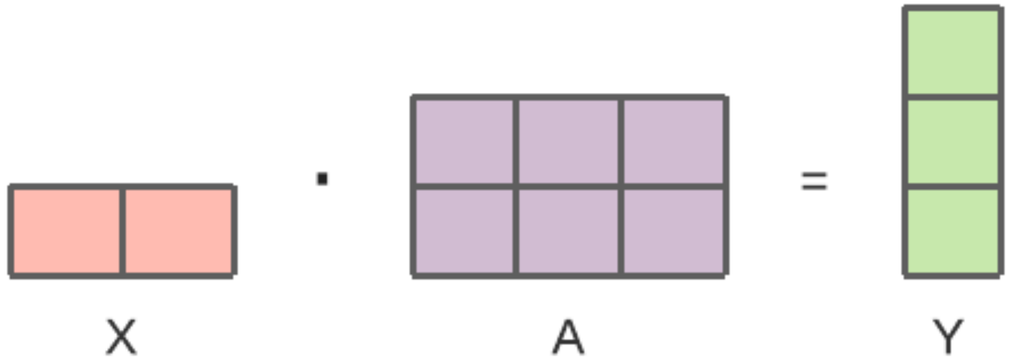
They also explore many interesting ideas, such as version control of weights (weight stashing) …



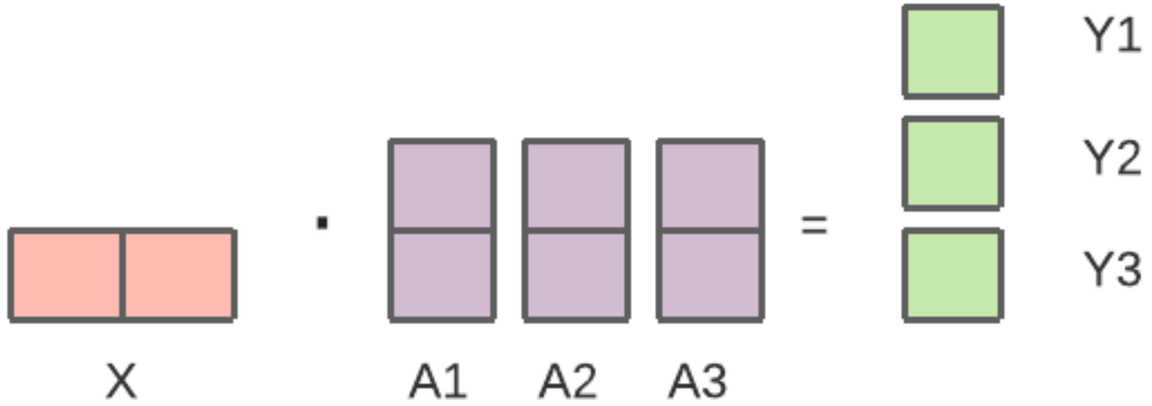
Narayanan et al. (2019), “PipeDream: Generalized pipeline parallelism for DNN training” SOSP 2019.

Tensor Parallelism. Split a tensor into several pieces, and distribute subtensors to several workers. This is possible because many operations in NN are parallelizable.

Example) Matrix multiplication.

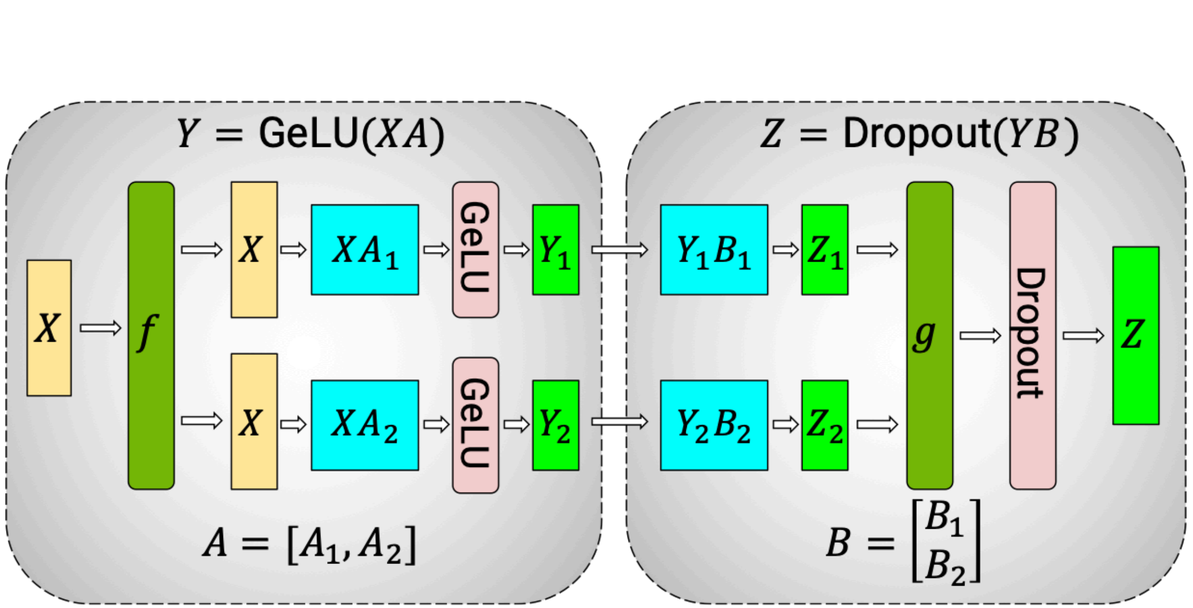


is equivalent to

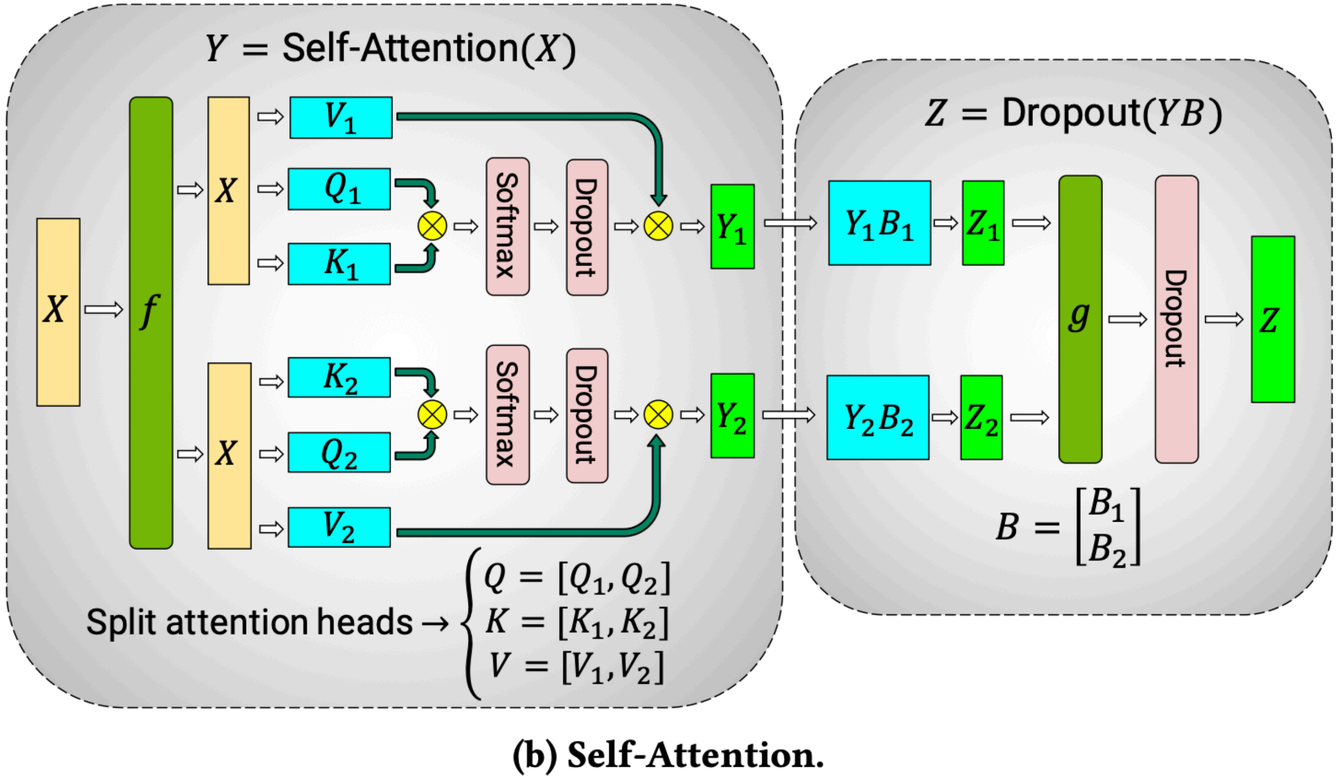


Such tensor parallelism has been widely exploited in the Megatron-LM (Shoeybi et al., 2019) to give an easy-to-use model parallelism framework that does not require any change in C++ base or compilers.

unlike mesh-tensorflow or GPipe



(a) MLP.



(b) Self-Attention.

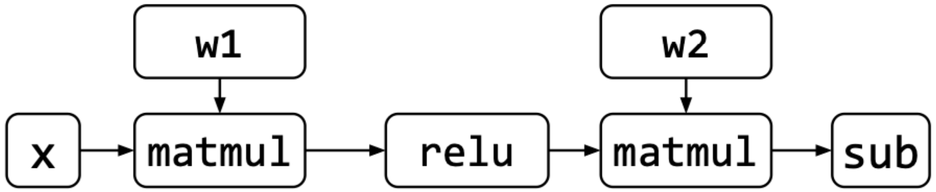
f: identity in forward pass, all-reduce in backward pass

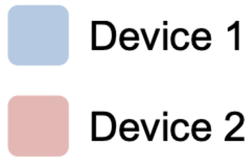
g: all-reduce in forward pass, identity in backward pass

Note. Dropout / Normalization are difficult to handle this way... Recent Megatron update (Korthikanti et al., 2022) introduces "sequence parallelism"

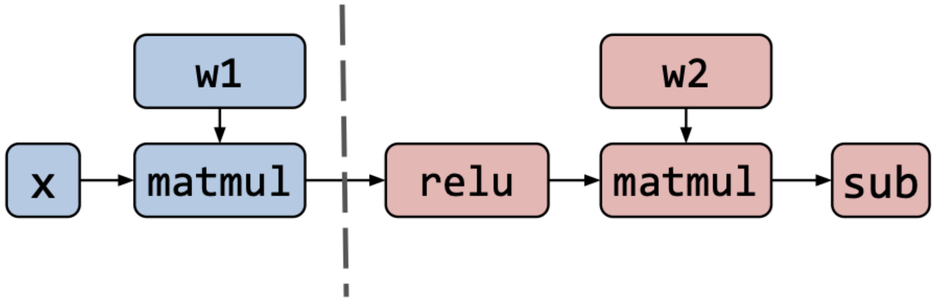
Shoeybi et al. (2019), "Megatron-LM: Training multi-billion parameter language models using model parallelism" arXiv 2019.
Korthikanti et al. (2022), "Reducing activation recomputation in large transformer models," arXiv 2022.

Mixed Parallelism. A recent paper proposes "Alpa" (Zheng et al., 2022) which automatically searches for a best mix of tensor parallelism (intra-operator) and pipeline parallelism (inter-operator)

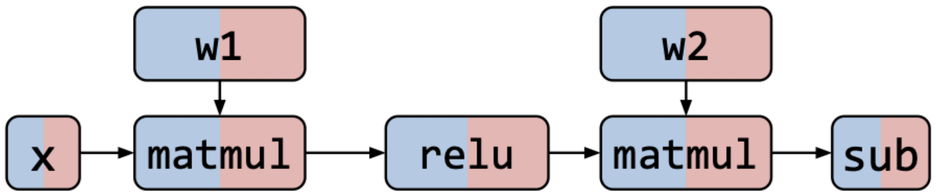




Strategy 1: Inter-operator Parallelism



Strategy 2: Intra-operator Parallelism



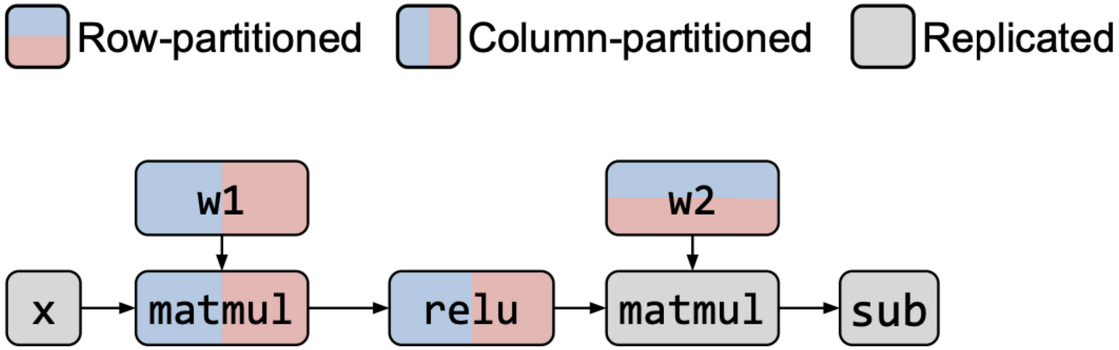
Trade-off

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

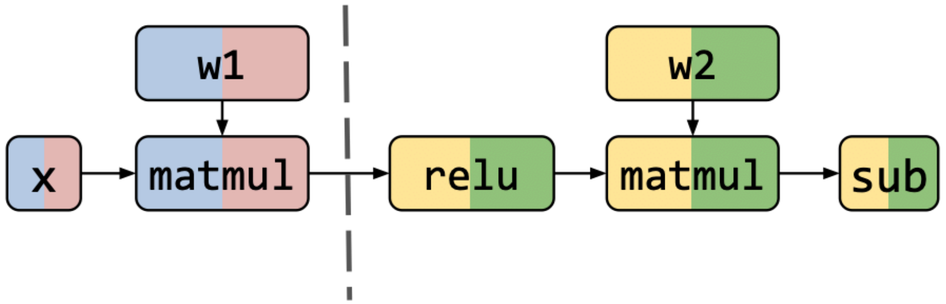
Zheng et al. (2022), "Alpa: Automating inter- and intra-operator parallelism for distributed deep learning," OSDI 2022.

Mixed Parallelism. A recent paper proposes “Alpa” (Zheng et al., 2022) which automatically searches for a best mix of tensor parallelism (intra-operator) and pipeline parallelism (inter-operator)

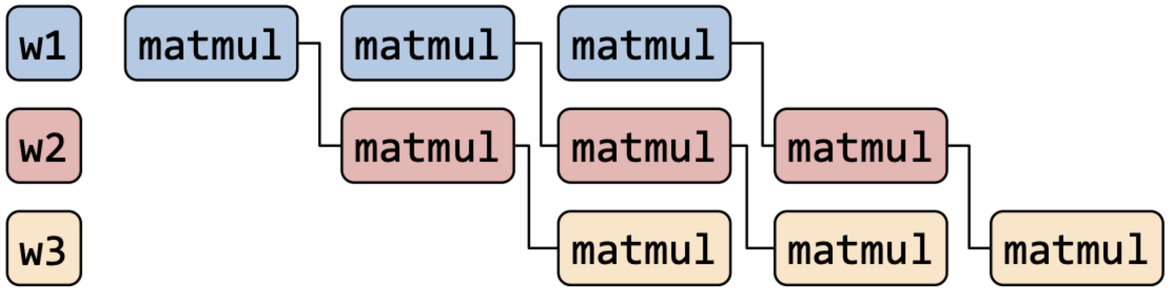
Multiple intra-op strategies for a single node

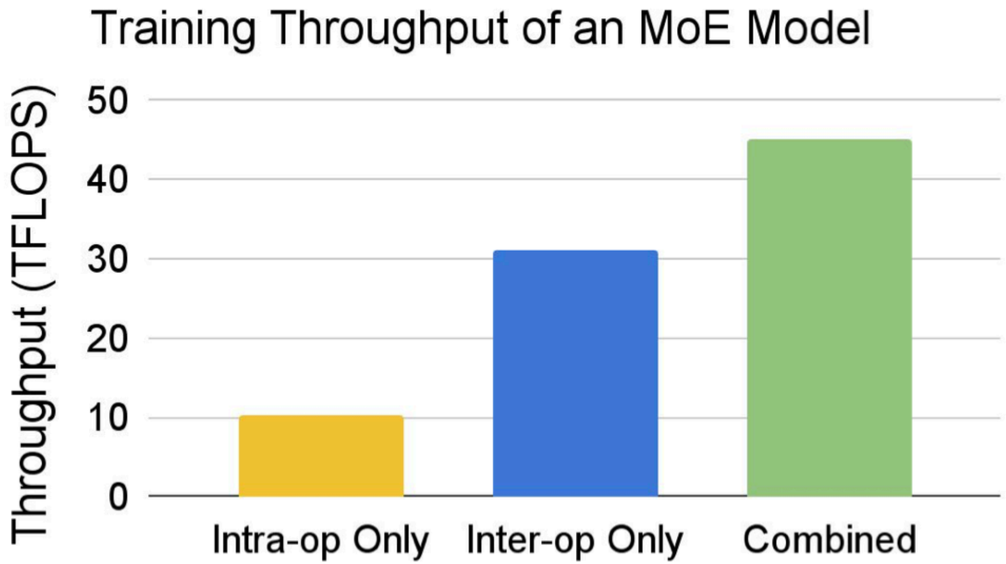


Combine Intra-op and Inter-op



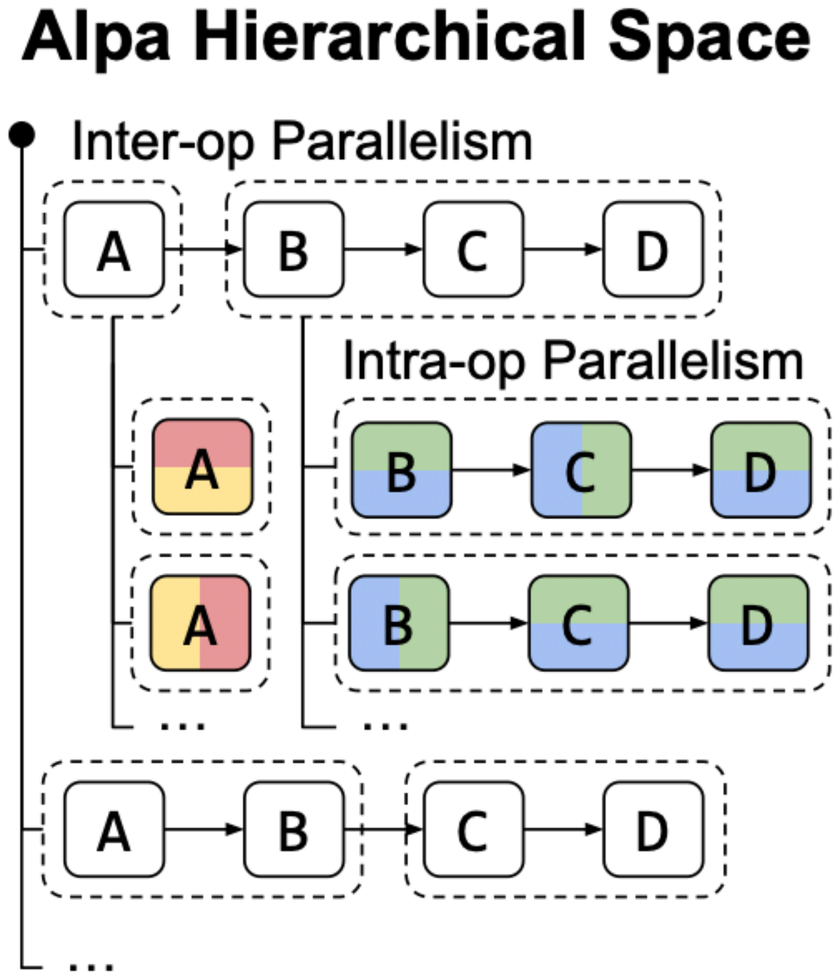
Pipeline the execution for inter-op parallelism

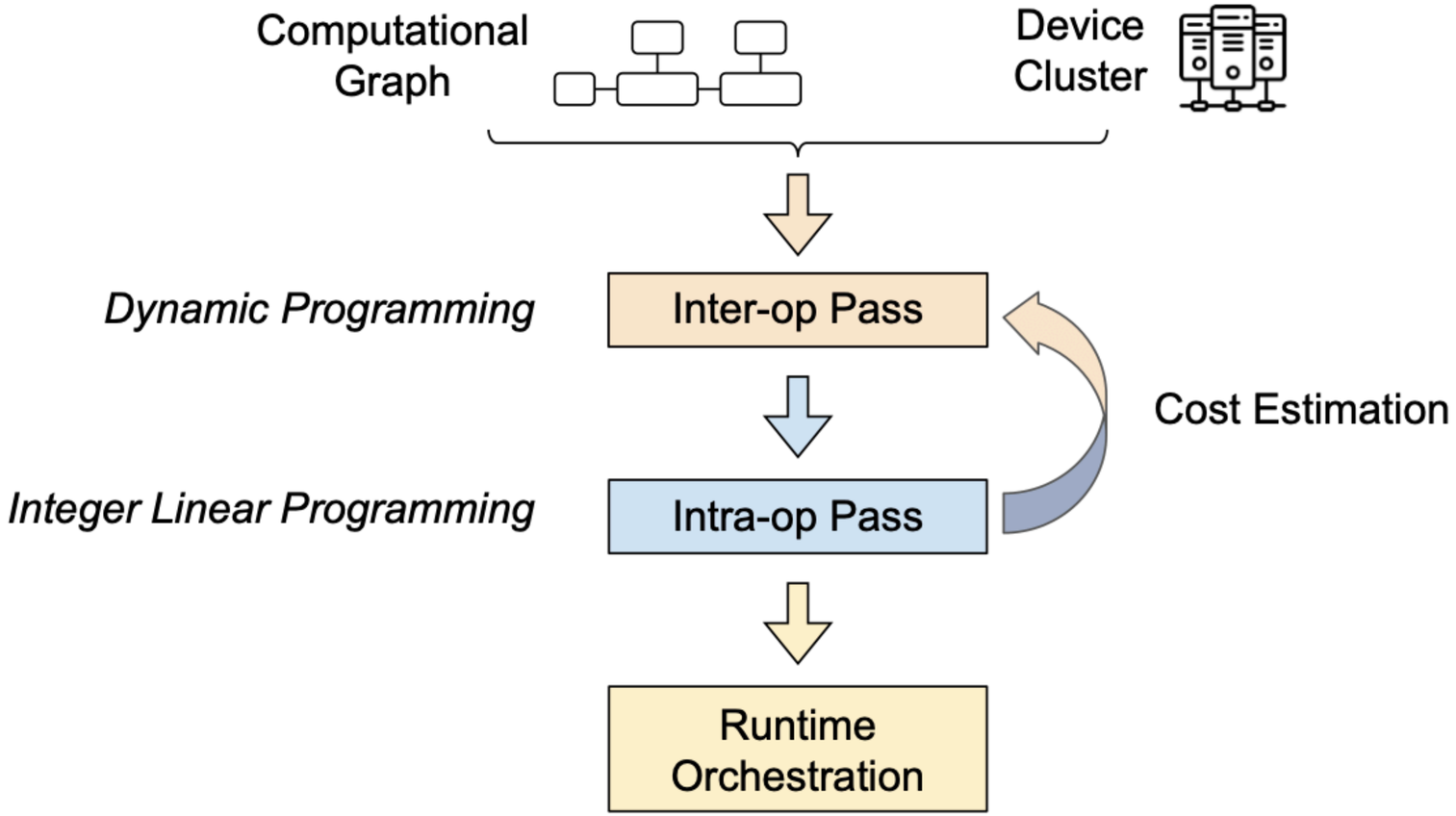




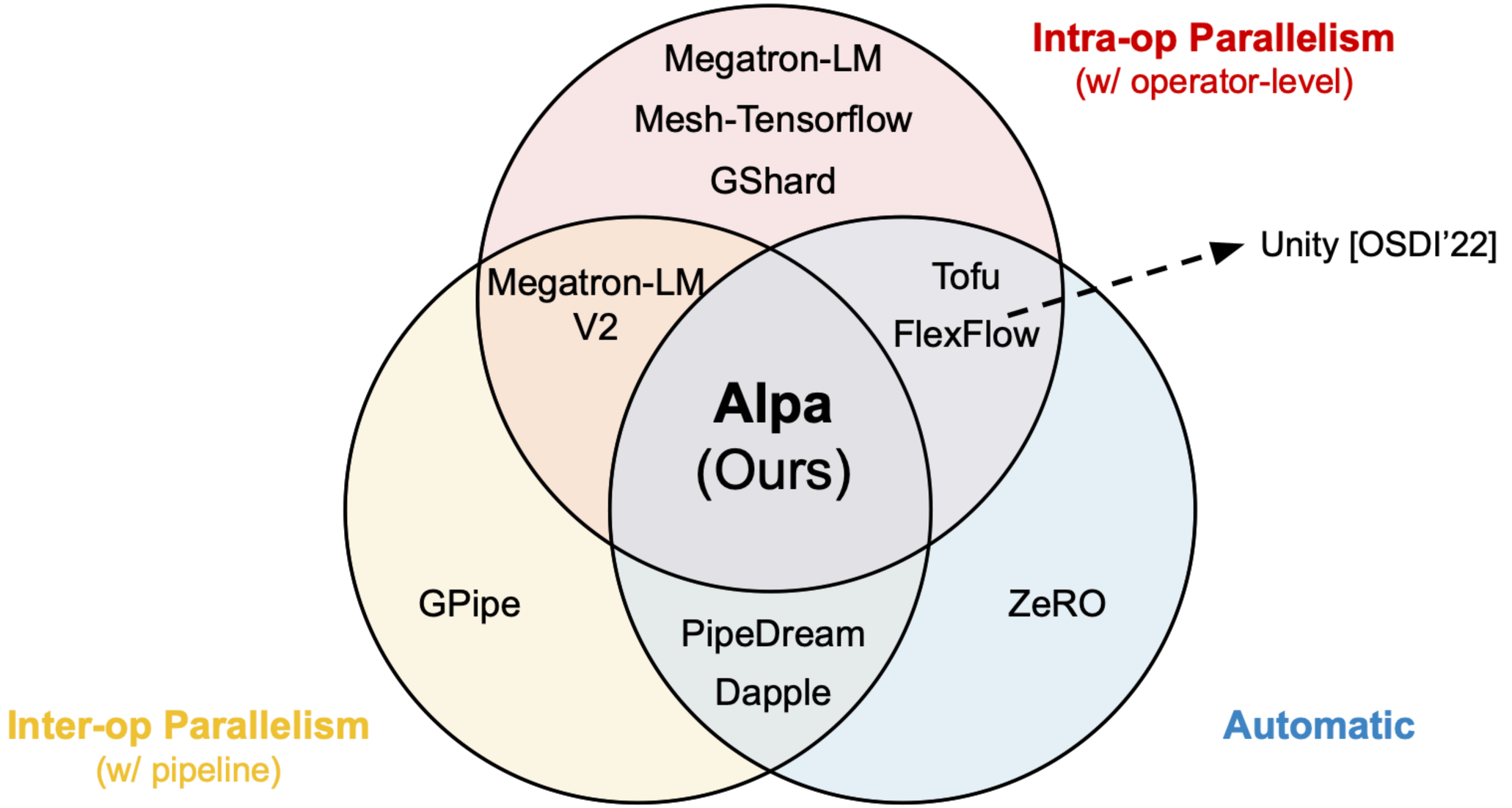
Zheng et al. (2022), “Alpa: Automating inter- and intra-operator parallelism for distributed deep learning,” OSDI 2022.

Mixed Parallelism. A recent paper proposes “Alpa” (Zheng et al., 2022) which automatically searches for a best mix of tensor parallelism (intra-operator) and pipeline parallelism (inter-operator)





Zheng et al. (2022), “Alpa: Automating inter- and intra-operator parallelism for distributed deep learning,” OSDI 2022.



Zheng et al. (2022), "Alpa: Automating inter- and intra-operator parallelism for distributed deep learning," OSDI 2022.

Next Class (Virtual).
A little more focused on data parallelism.
Gradient compression techniques.
More on asynchronicity.