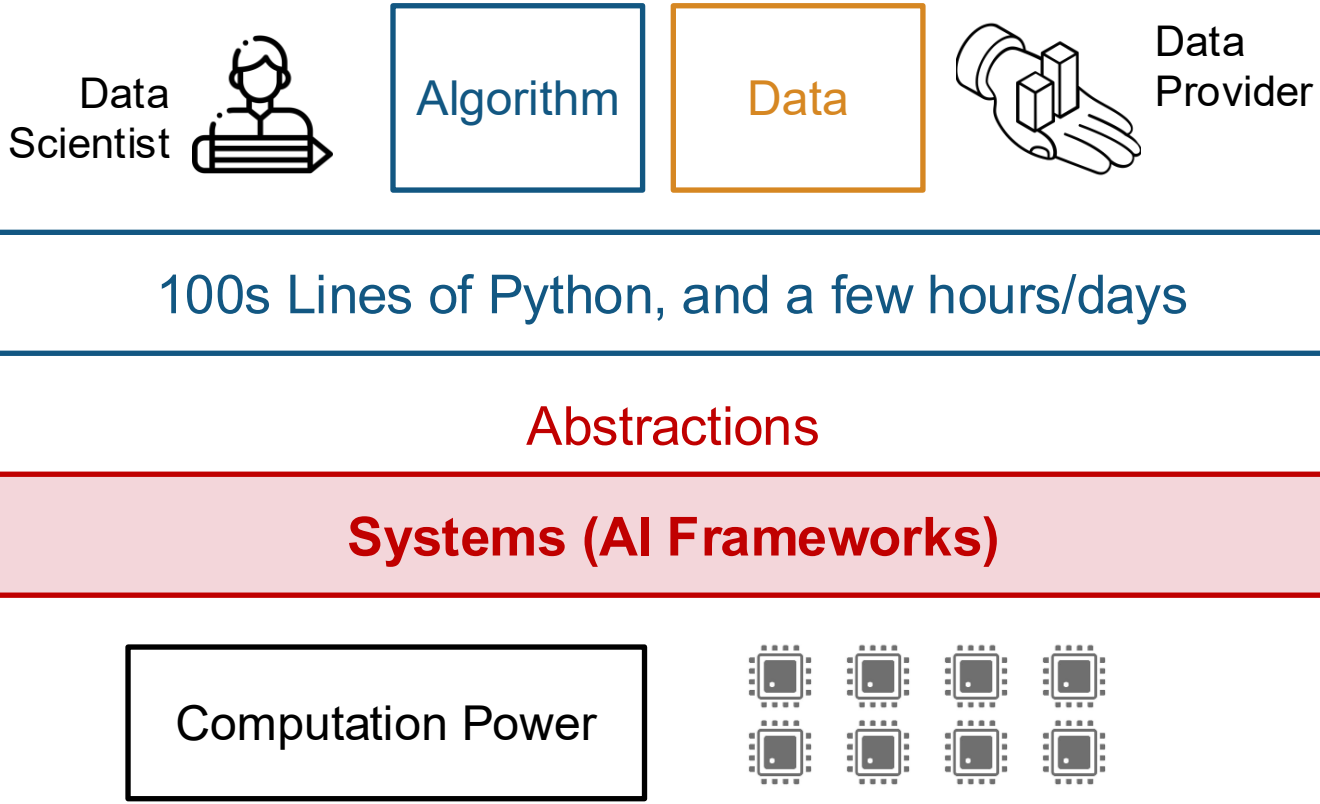
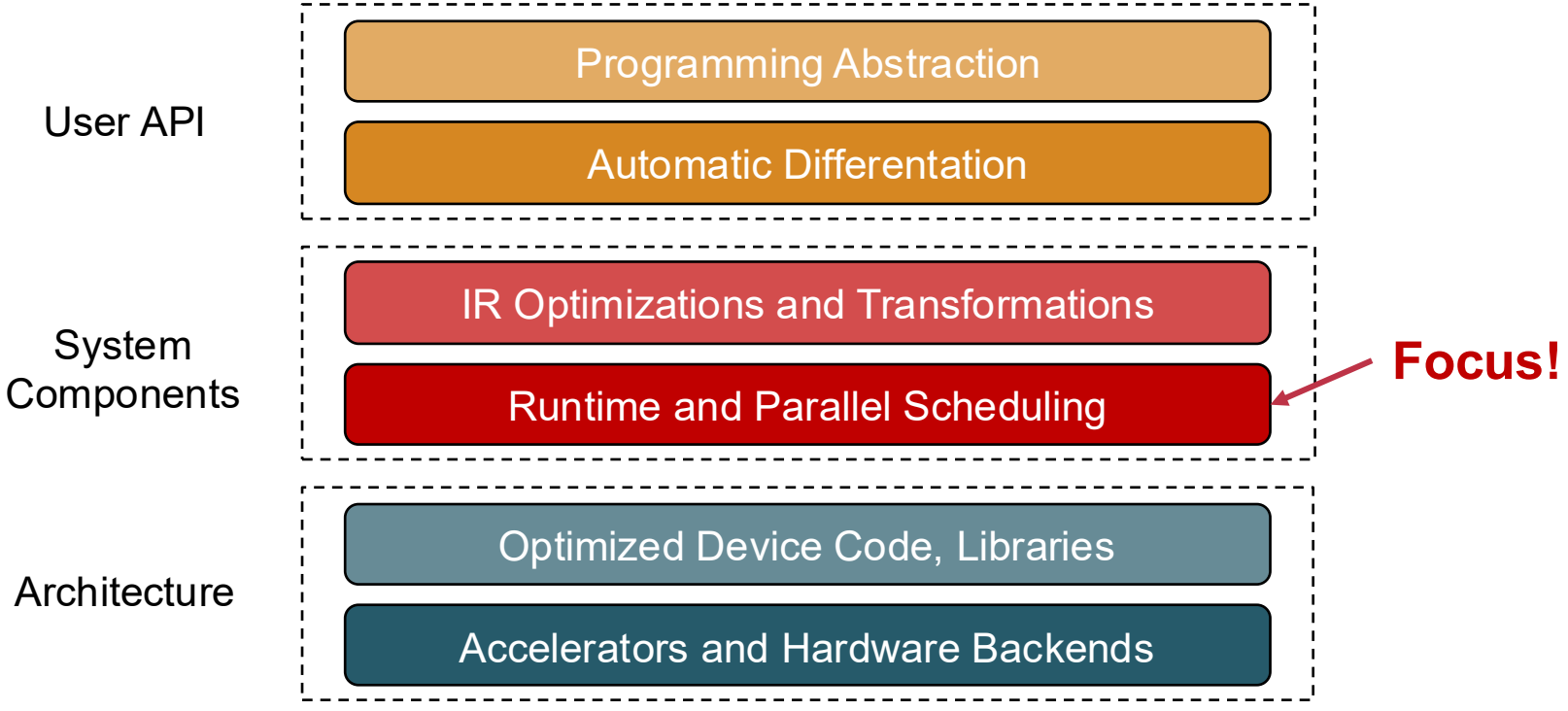
ACCELERATING DISTRIBUTED AI INFRASTRUCTURE SYSTEMS (I)

AI Systems



Credit: “Machine Learning Systems”, 15-884, CMU

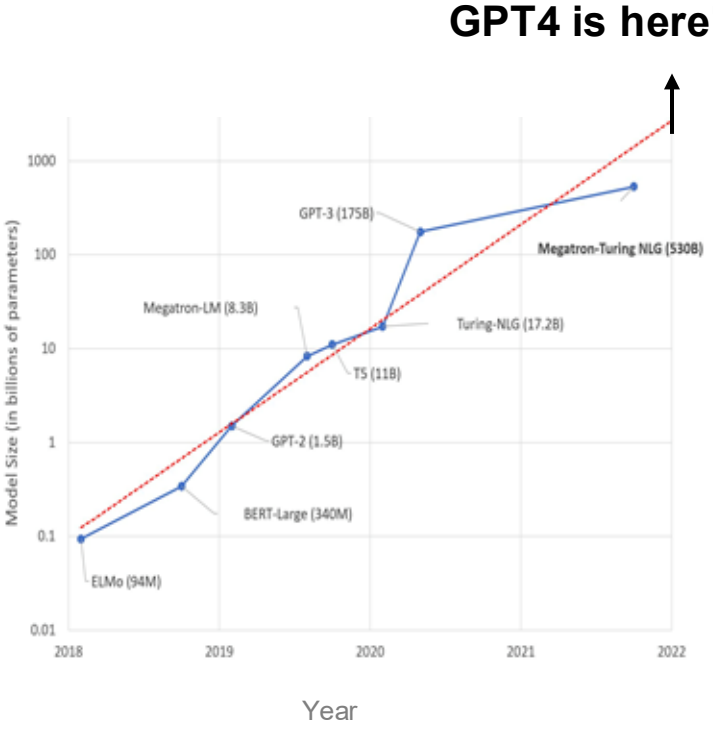
AI Systems

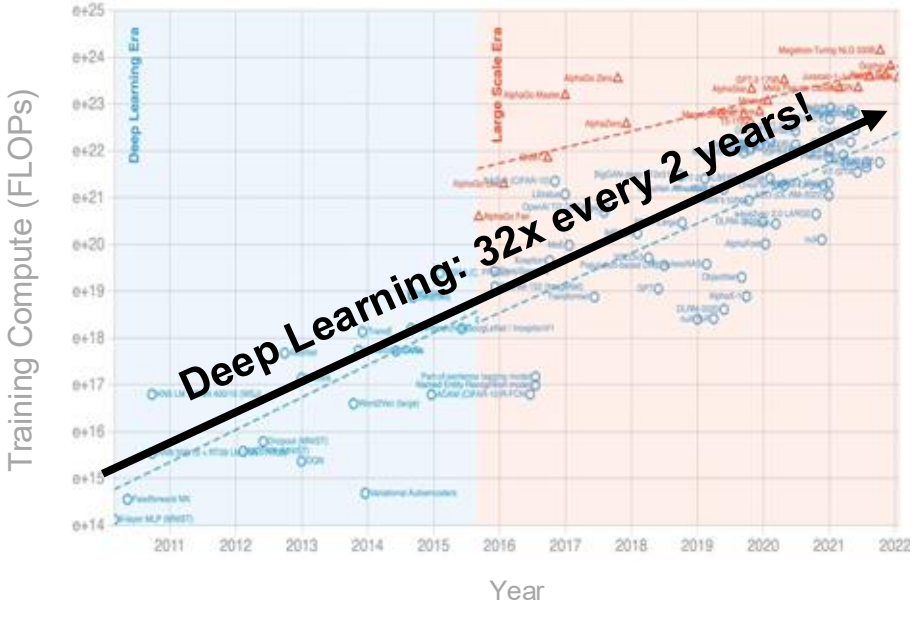


Key Question: How to parallelizing AI systems?

Credit: “Machine Learning Systems”, 15-884, CMU

Why to Parallelize AI Systems?



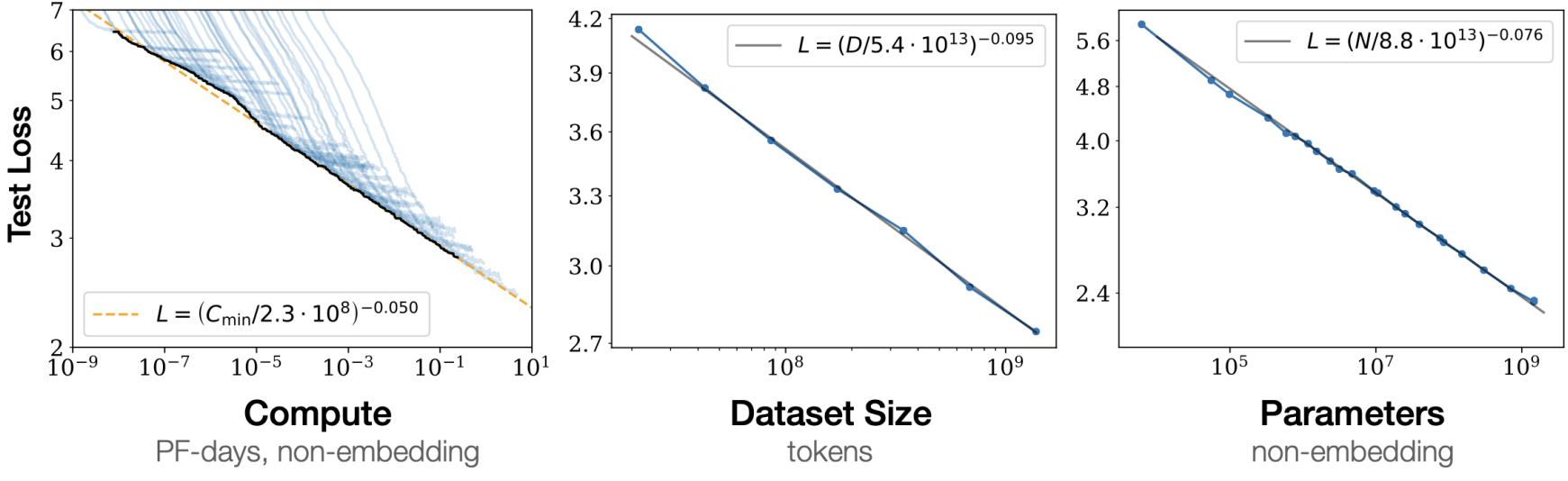


Why to Parallelize AI Systems?

AI performance improves from all forms of scale

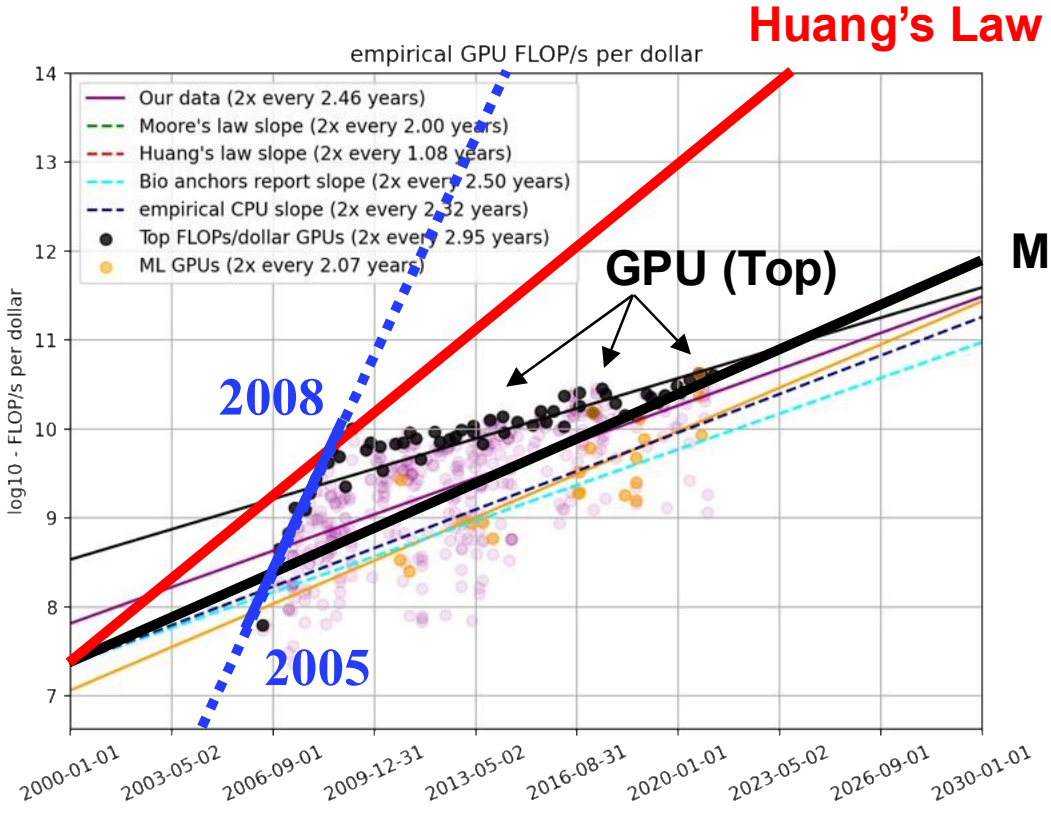
Massive computation power, data, and model

Scaling Law!



From: Scaling Laws for Neural Language Models.
Credit: “Parallelism Fundamentals”, CS 229S, Stanford

Why to Parallelize AI Systems?



Scaling-up is DEAD

Scaling-out is NOW

Parallelizing AI Systems

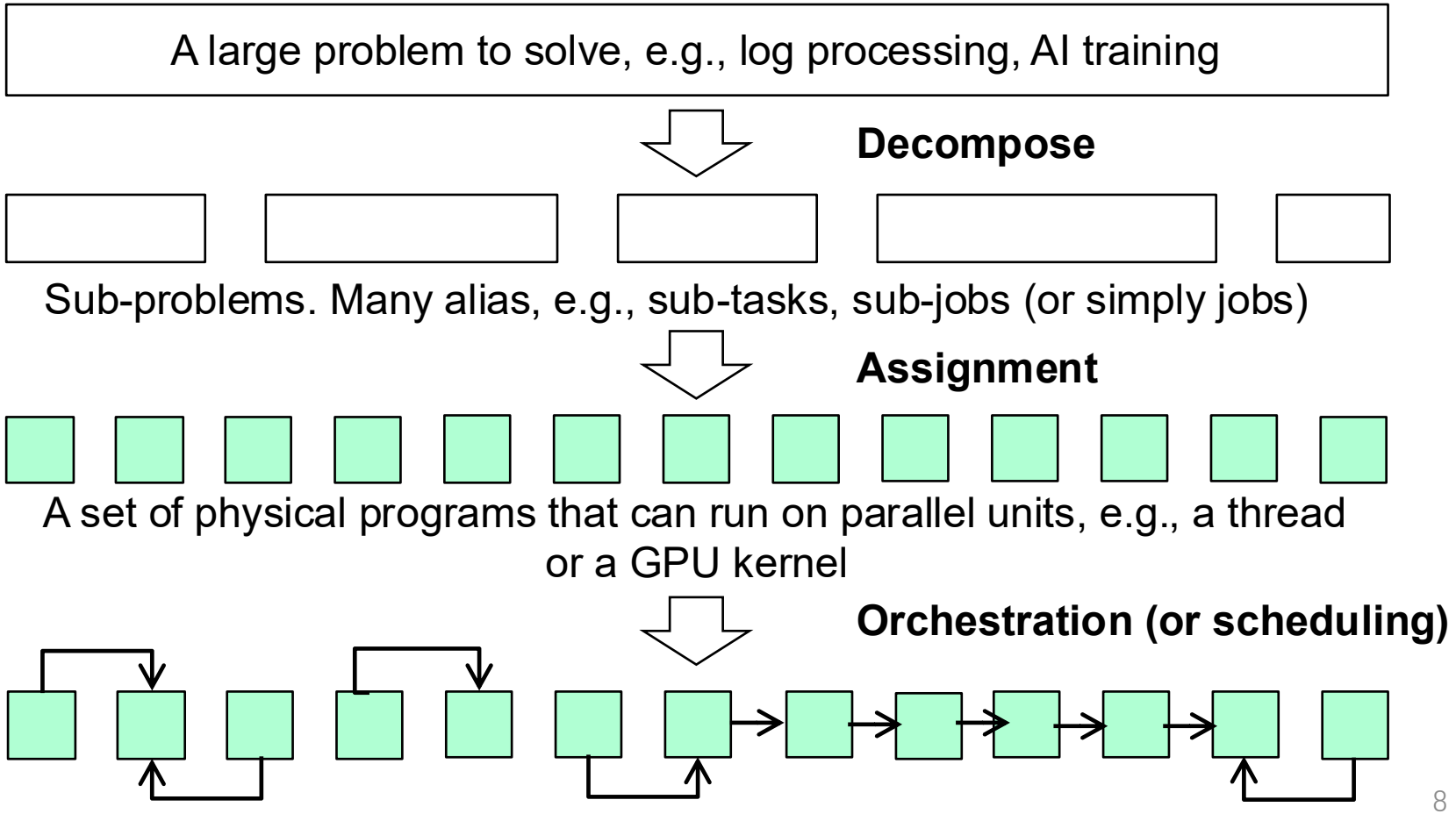
Similar characteristics in AI systems

- Massive data and resources
- Data dependency and update synchronization
- Iterative convergence

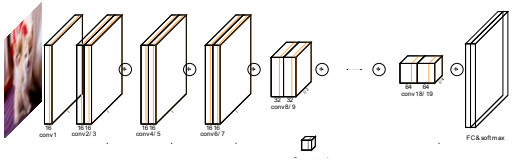
New characteristics in AI systems

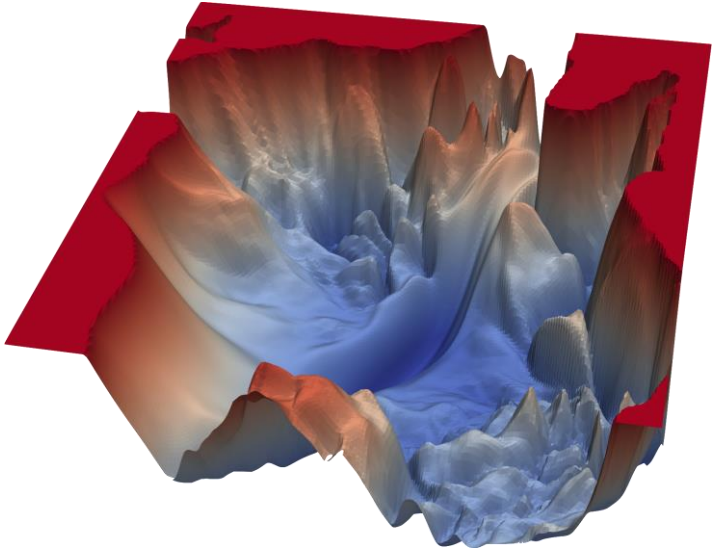
- **Compute-intensive**: large models (parameter scale: ~10^9)
- **Execution flow**: back-propagation, selective (MoE, sparse-activation)
- **New resources**: GPU and accelerators, NVLink/NVSwitch

Parallelizing AI Tasks



AI Training Workload





Stochastic Gradient Descent (SGD)

$$\min_{w} \mathcal{J}(w) = \frac{1}{N} \sum_{i=1}^{N} cost(w, x_i)$$

$$w^1 = w^0 - \underbrace{\frac{\alpha}{B} \sum_{i=1}^{B} \frac{\partial \mathcal{J}(w^0)}{\partial w}}_{\Delta w}$$

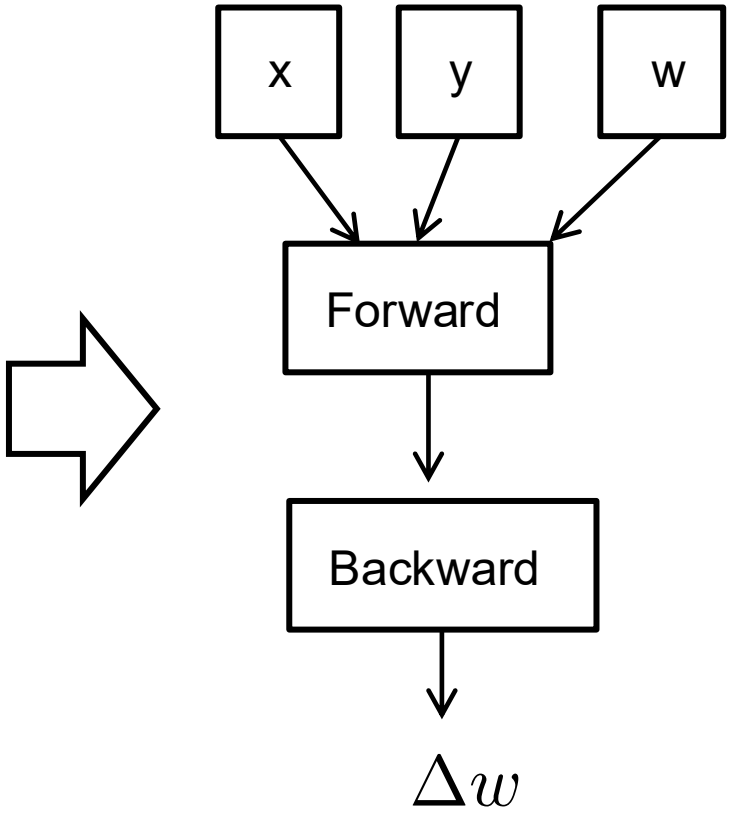
Stochastic Gradient Descent into Computation Graph

X: input features

Y: labels in supervised learning

$$\min_{w} \mathcal{J}(w) = \frac{1}{N} \sum_{i=1}^{N} cost(w, x_i)$$

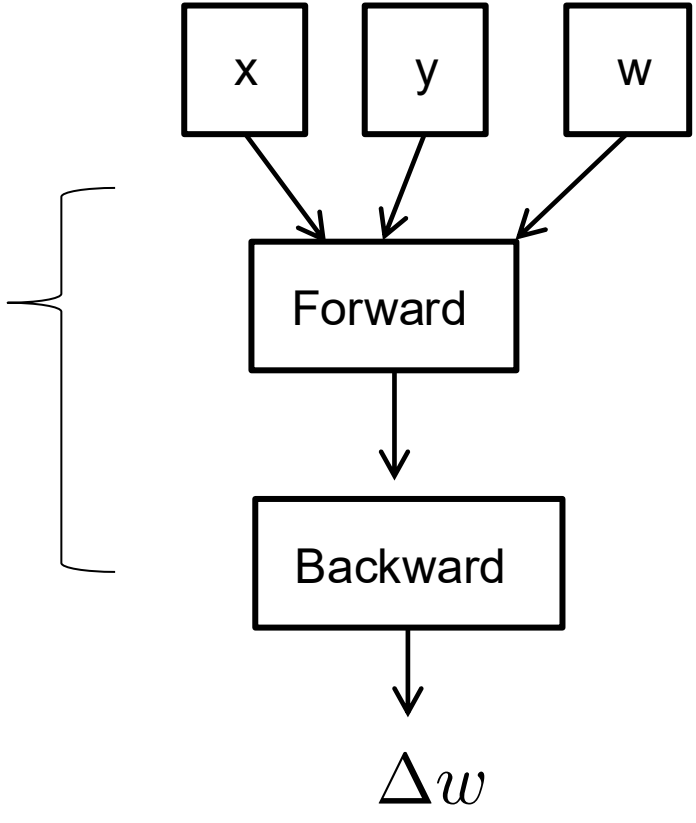
$$w^1 = w^0 - \underbrace{\frac{\alpha}{B} \sum_{i=1}^{B} \frac{\partial \mathcal{J}(w^0)}{\partial w}}_{\Delta w}$$



Pseudocode for Stochastic Gradient Descent

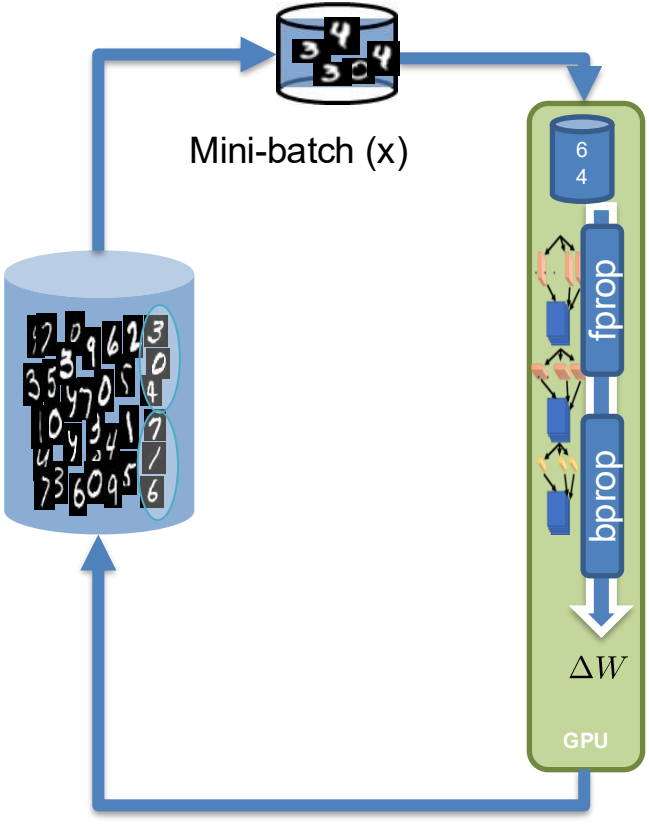
```
// execute on a master
for iter in num_iters:
```

$$w^{iter+1} = w^{iter} - \frac{\alpha}{B} \sum_{i=1}^{B} \frac{\partial \mathcal{J}(w^{iter})}{\partial w}$$



Overview of AI Training in Action

In every iteration of SGD we load a random mini-batch of training data and compute gradient



$$\min_w \mathcal{J}(w) = \frac{1}{N}\sum_{i=1}^{N} cost(w, x_i)$$

$$w^1 = w^0 \underbrace{- \frac{\alpha}{B}\sum_{i=1}^{B} \frac{\partial \mathcal{J}(w^0)}{\partial w}}_{\Delta w}$$

Parallelization Opportunities

Data Parallelism: *Distribute the processing of data to multiple PEs.*

$$w^1 = w^0 - \frac{\alpha}{B} \sum_{i=1}^{B} \frac{\partial \mathcal{J}(w^0)}{\partial w}$$

Model Parallelism: *Break the model and distribute processing of every layer to multiple PEs*

$$w^1 = w^0 - \frac{\alpha}{B} \sum_{i=1}^{B} \frac{\partial \mathcal{J}(w^0)}{\partial w}$$

For either approach it is also possible to use ***synchronous*** *or* ***asynchronous*** *updates*

$$w^1 = w^0 - \frac{\alpha}{B} \sum_{i=1}^{B} \frac{\partial \mathcal{J}(w^0)}{\partial w}$$

DATA PARALLELISM

Data Parallelism

Data Parallelism: *Distribute the processing of data to multiple PEs.*

$$w^1 = w^0 - \frac{\alpha}{B} \sum_{i=1}^{B} \frac{\partial \mathcal{J}(w^0)}{\partial w}$$

Model Parallelism: *Break the model and distribute processing of every layer to multiple PEs*

$$w^1 = w^0 - \frac{\alpha}{B} \sum_{i=1}^{B} \frac{\partial \mathcal{J}(w^0)}{\partial w}$$

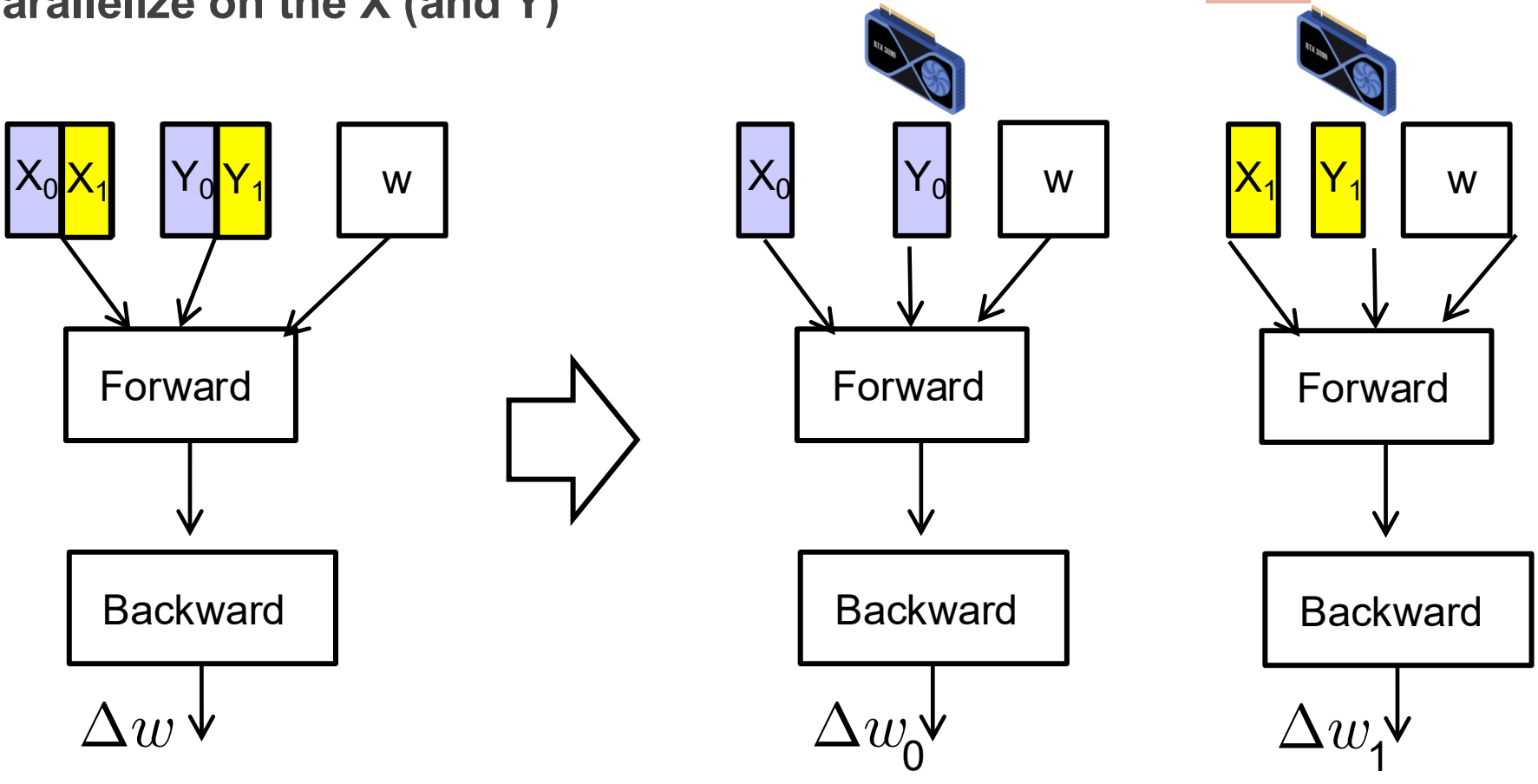
For either approach it is also possible to use ***synchronous*** *or* ***asynchronous*** *updates*

$$w^1 = w^0 - \frac{\alpha}{B} \sum_{i=1}^{B} \frac{\partial \mathcal{J}(w^0)}{\partial w}$$

Data Parallelism

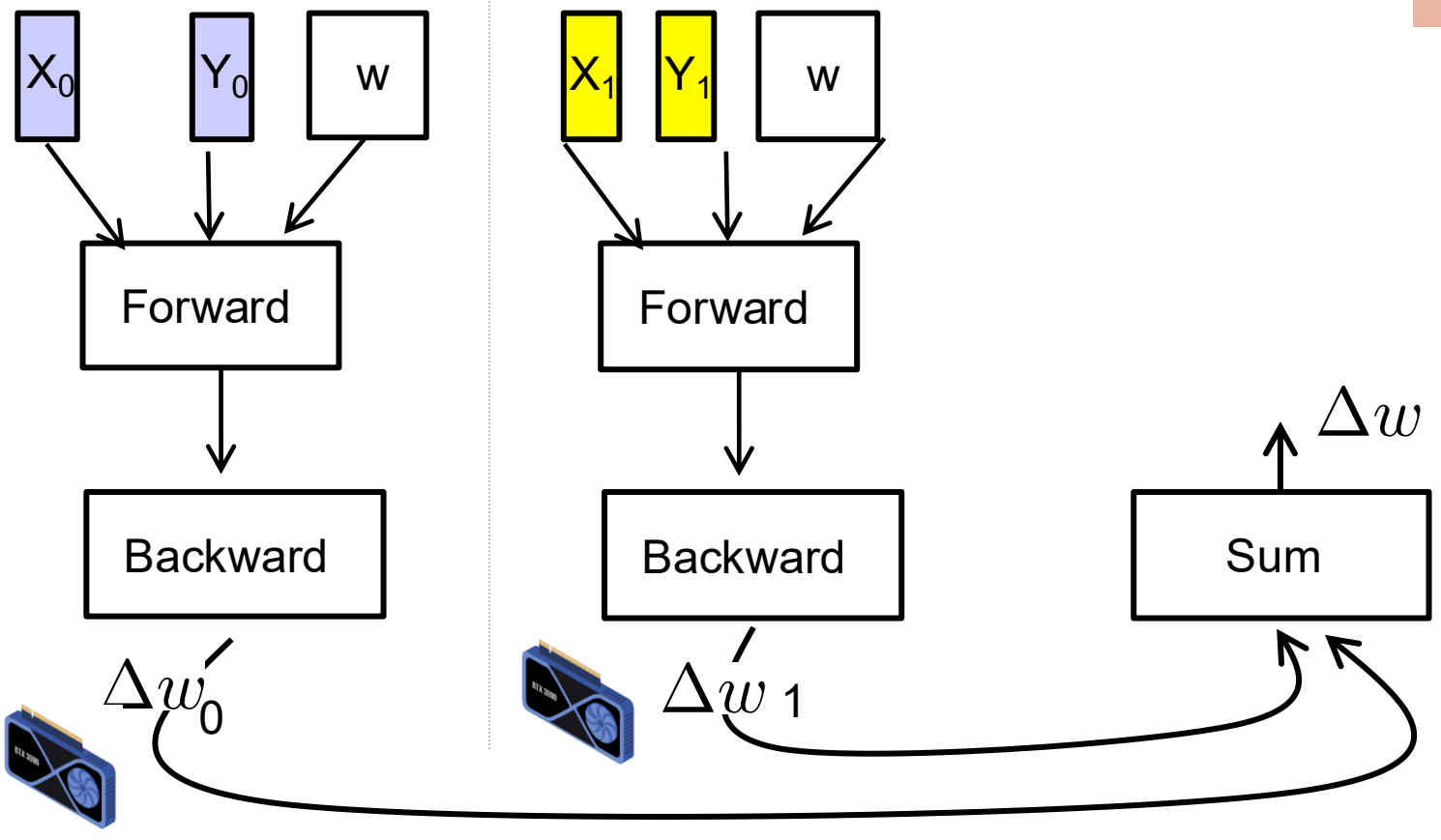
$$w^1 = w^0 - \frac{\alpha}{B} \sum_{i=1}^{B} \frac{\partial \mathcal{J}(w^0)}{\partial w}$$

Parallelize on the X (and Y)

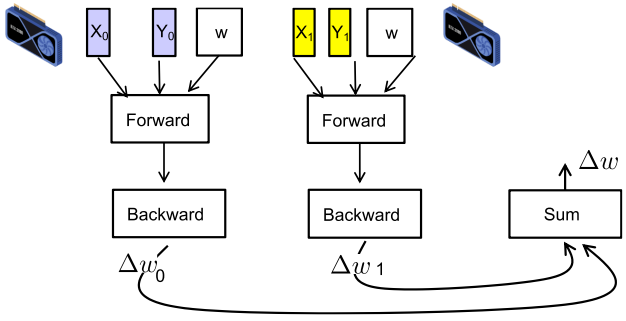


Data Parallelism

$$w^1 = w^0 - \frac{\alpha}{B}\sum_{i=1}^{B}\frac{\partial \mathcal{J}(w^0)}{\partial w}$$



Workflow of Data Parallelism



Model Replication

– Replicate model across multiple GPUs

Data Splitting

– Split input data into smaller batches and send to different GPUs

Parallel Computation

– Perform forward and backward passes on each GPU independently

Gradient Aggregation

– Gather and average Gradients from all GPUs

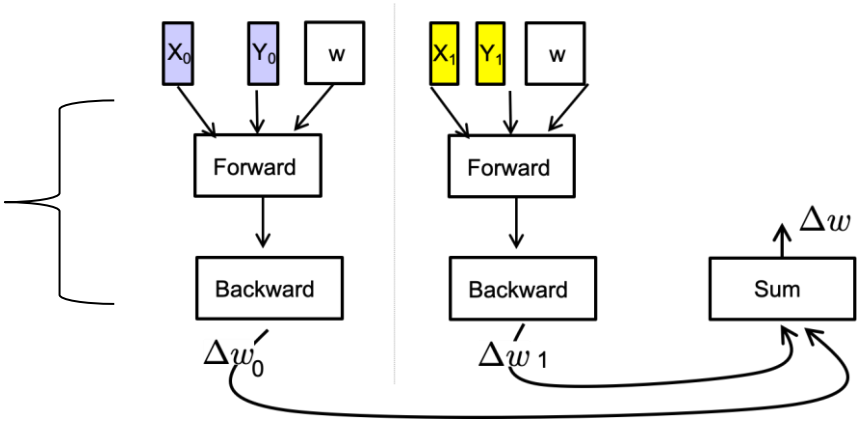
Model Update

– Update the model parameters with the averaged gradients

Pseudocode for (Iterative) Data Parallelism in Action

```
// execute on a master
for iter in num_iters:
```

$$w^{iter+1} = w^{iter} - \frac{\alpha}{B} \sum_{i=1}^{B} \frac{\partial \mathcal{J}(w^{iter})}{\partial w}$$



The master will do the coordination

- Similar to the Job manager in Dryad

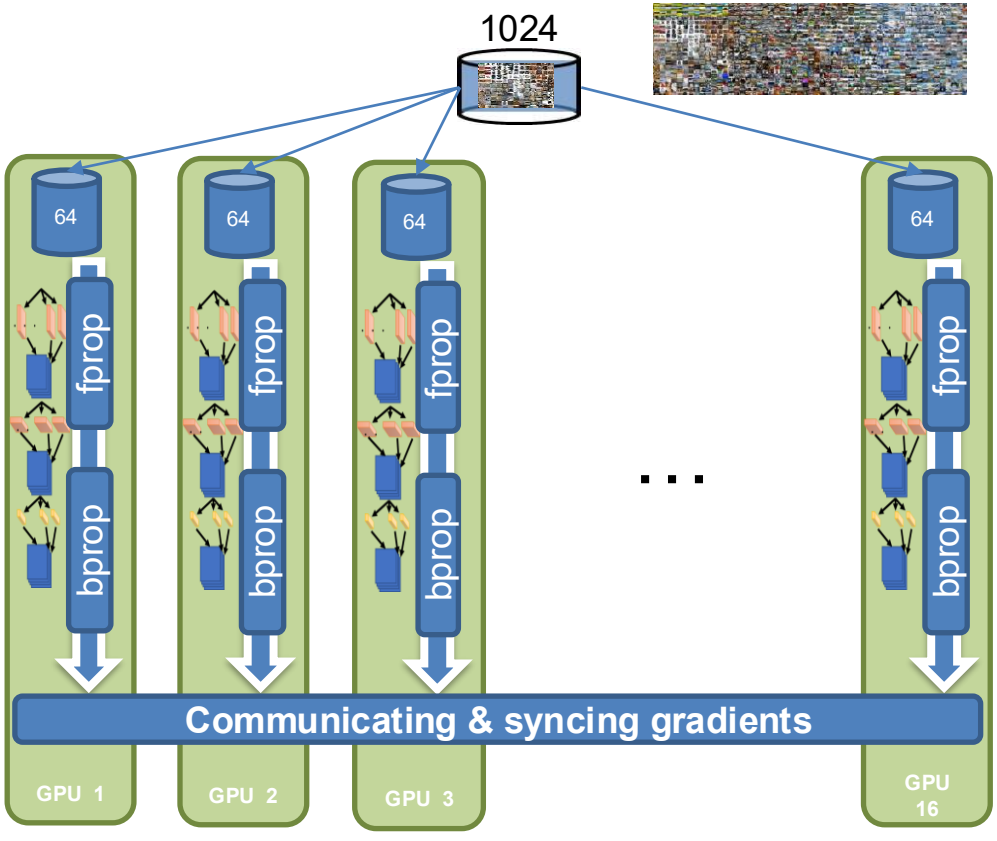
Data Parallelism w/ BSP

Store the entire model (W) on each processor

- Then distribute the batch evenly across each processor
- E.g., 64 images per GPU

Question: how to coordinate between iterations?

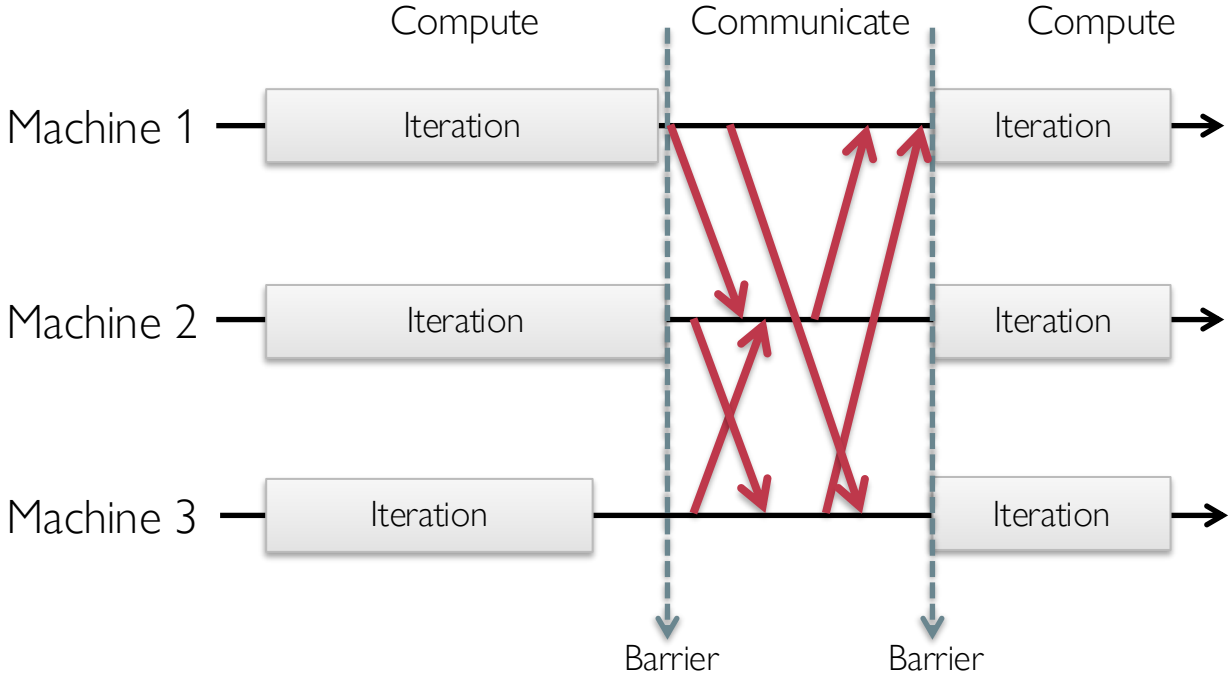
$$w^1 = w^0 - \frac{\alpha}{B} \sum_{i=1}^{B} \frac{\partial \mathcal{J}(w^0)}{\partial w}$$



How to Parallelize Partitioned Graphs?

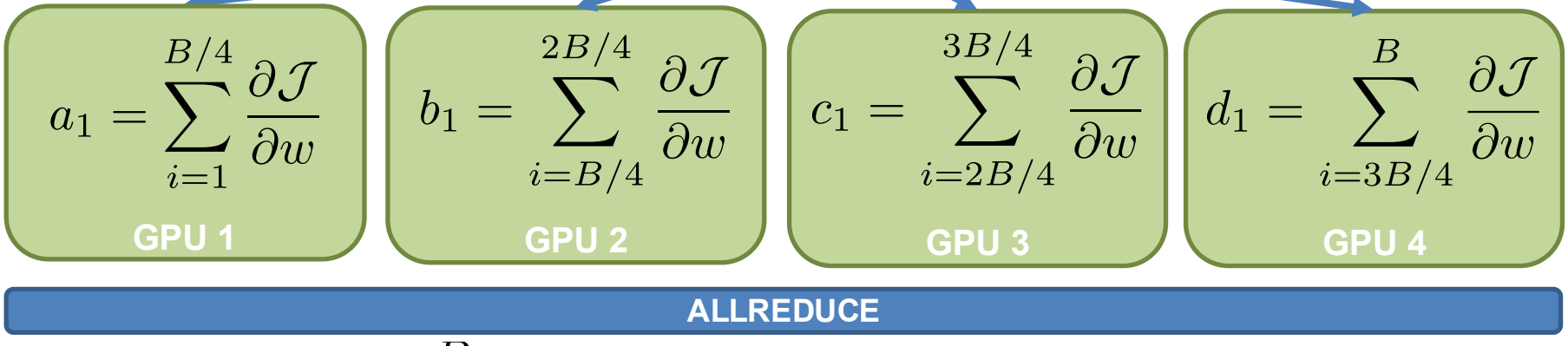
Bulk Synchronous Parallel (BSP) Execution

- Using a strict barrier to coordinator processes in a distributed computing
- Barrier can be implemented in a centralized way (master) or decentralized



Key Operator: AllReduce

$$w^1 = w^0 - \frac{\alpha}{B} \sum_{i=1}^{B} \frac{\partial \mathcal{J}(w^0)}{\partial w}$$



$$\sum_{i=1}^{B} \frac{\partial \mathcal{J}}{\partial w} = a_1 + b_1 + c_1 + d_1$$

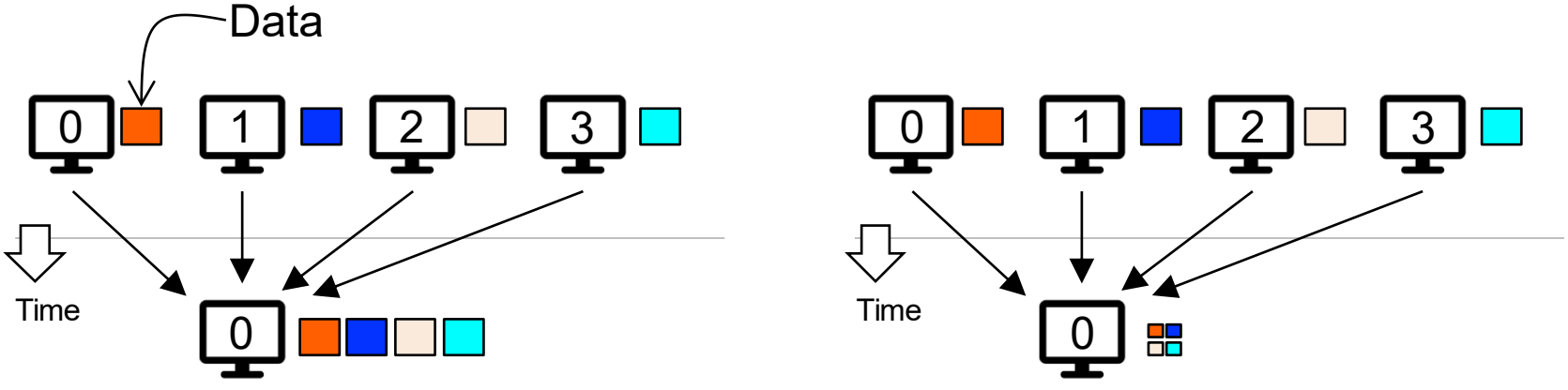
Communication Primitives in Distributed Computing

Gather: Collect data from all the other processes

- Example: [1][2][3][4] -> (gather) -> [1,2,3,4]

Reduce: Gather and aggregate a piece of data from all the other processes

- Example: [1][2][3][4] -> (reduce) -> [1+2+3+4] = [10]



Gather vs. Reduce

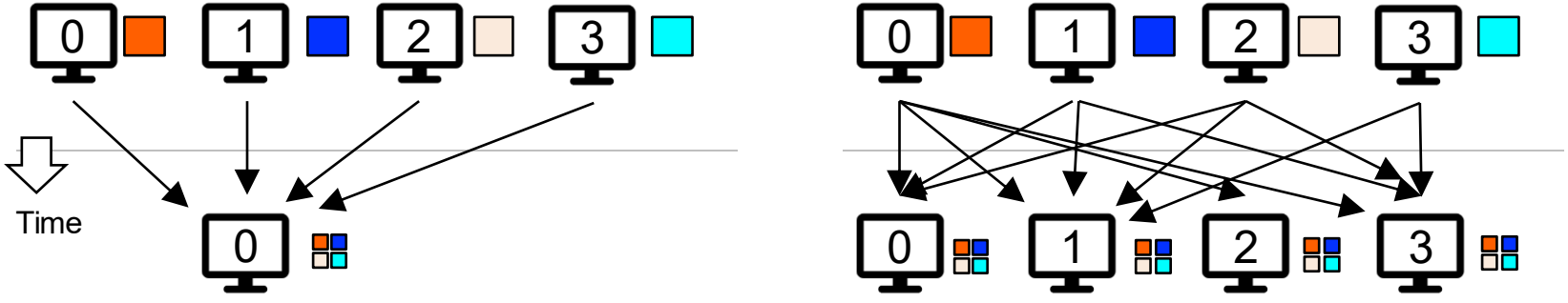
Communication Primitives in Distributed Computing

Reduce: Gather and aggregate a piece of data from all the other processes

– Example: [1][2][3][4] -> (reduce) -> [1+2+3+4] = [10]

Allreduce = reduce on all processes

Question: how to efficiently implement allreduce?



Reduce vs. AllReduce

System Requirements for AllReduce

Overhead analysis: computation + network

The overhead of computation is typically small

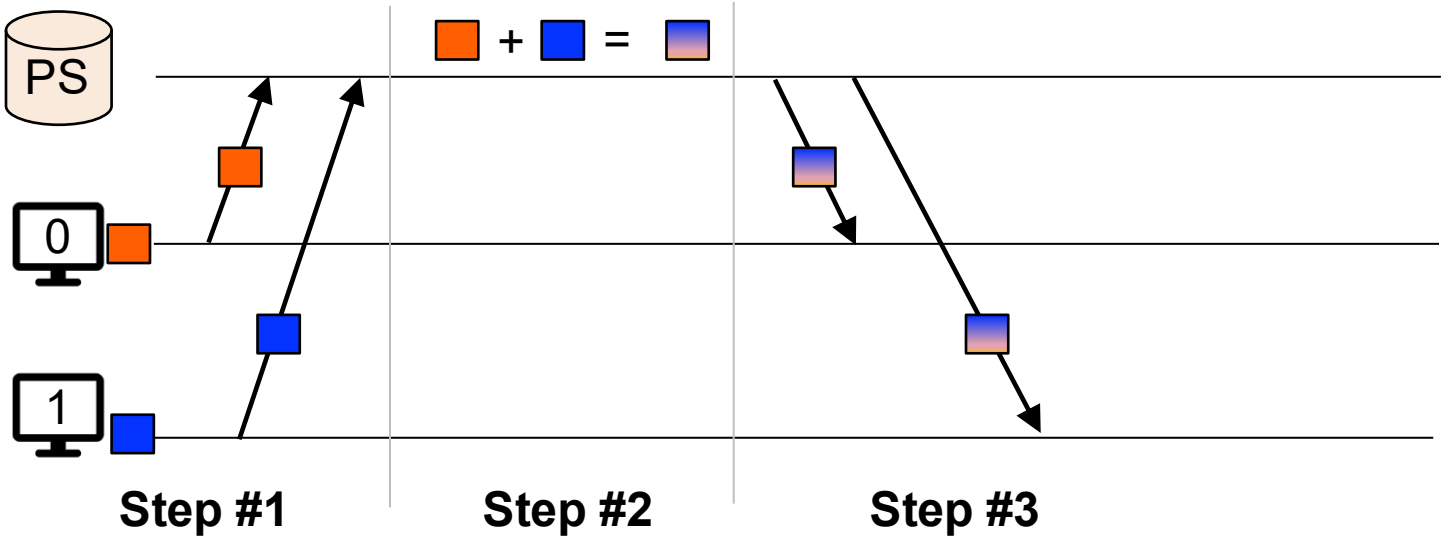
- Compute a sum operation on device is highly optimized, e.g., SIMD, GPU, TPU, etc.

Goal: reduce network overhead

- Reduce bytes sent per-message
- Reduce concurrent messages sent to a server

Try #1: Parameter Server

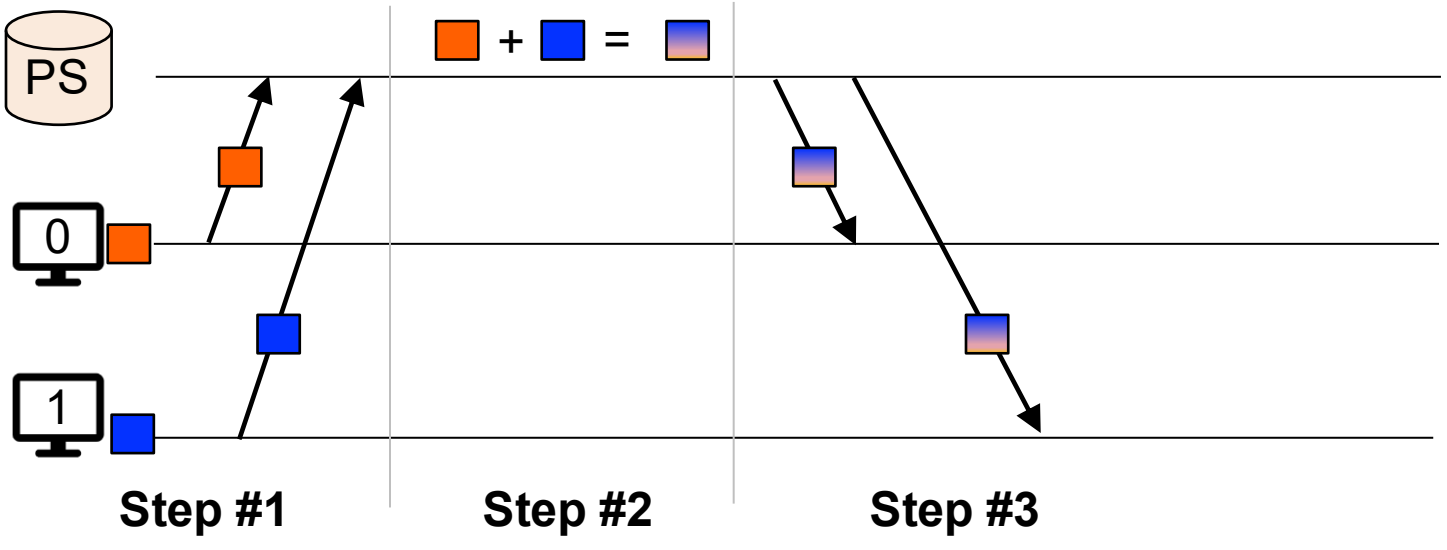
1. Each process pushes the data to the parameter server (PS)
2. After receiving all data, aggregate the data at the PS
3. The PS pushes the data back to the processes



AllReduce with A Single Parameter Server (PS)

Question: what is the network requirements at each stage?

- Step #1: O(1) at each process, O(n) at the PS
- Step #3: O(1) at each process, O(n) at the PS



AllReduce with A Single Parameter Server (PS)

Problem: huge bandwidth requirement at the PS (O(P * N))

- As well as huge contention at the master machine

Example: training ResNET50 on V100

- ResNET50: 24.4 M params ~= 97.5 MB storage
- On V100, each node can train 3 iterations / second (forward + backward path), w/o communication
- #Processing Node: 256
- Total requirement bandwidth: 256 * 3 * 97.5 = 73.1GB/s

The state-of-the-art NIC: 400Gbps RDMA ~= 50GB/s

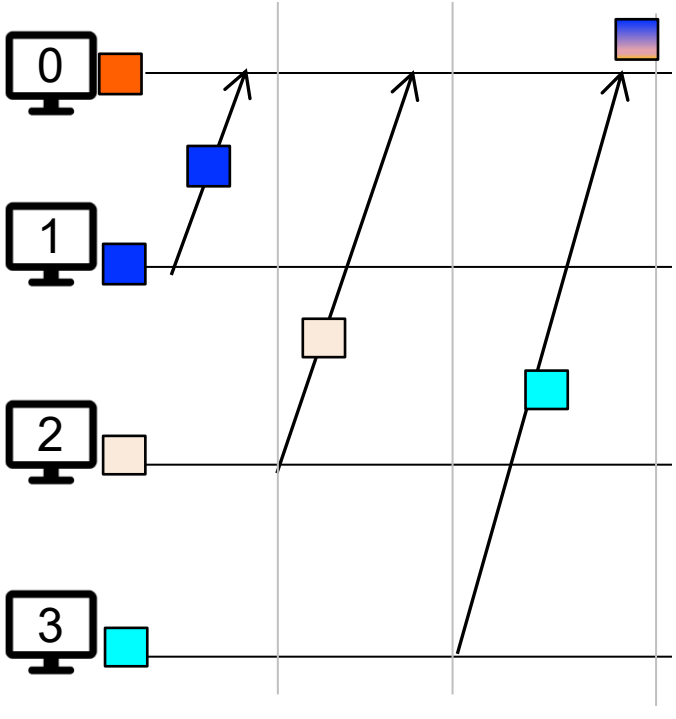
- The network would become the bottleneck!

Goal: reduce messages send to a centralized server

Try #2: Decentralized Approach for AllReduce

Each node finishes its allreduce, one-by-one

– E.g., with a careful designed scheduler



Analysis of Naïve Decentralized AllReduce

N: total parameters, P: # processors

Total communication per-machine

- **N * (P – 1) data (Bad …)**
- O(1) fan-in (Good!)

Total communication rounds: O(P * P)

- Much higher than the parameter server approach (O(1))

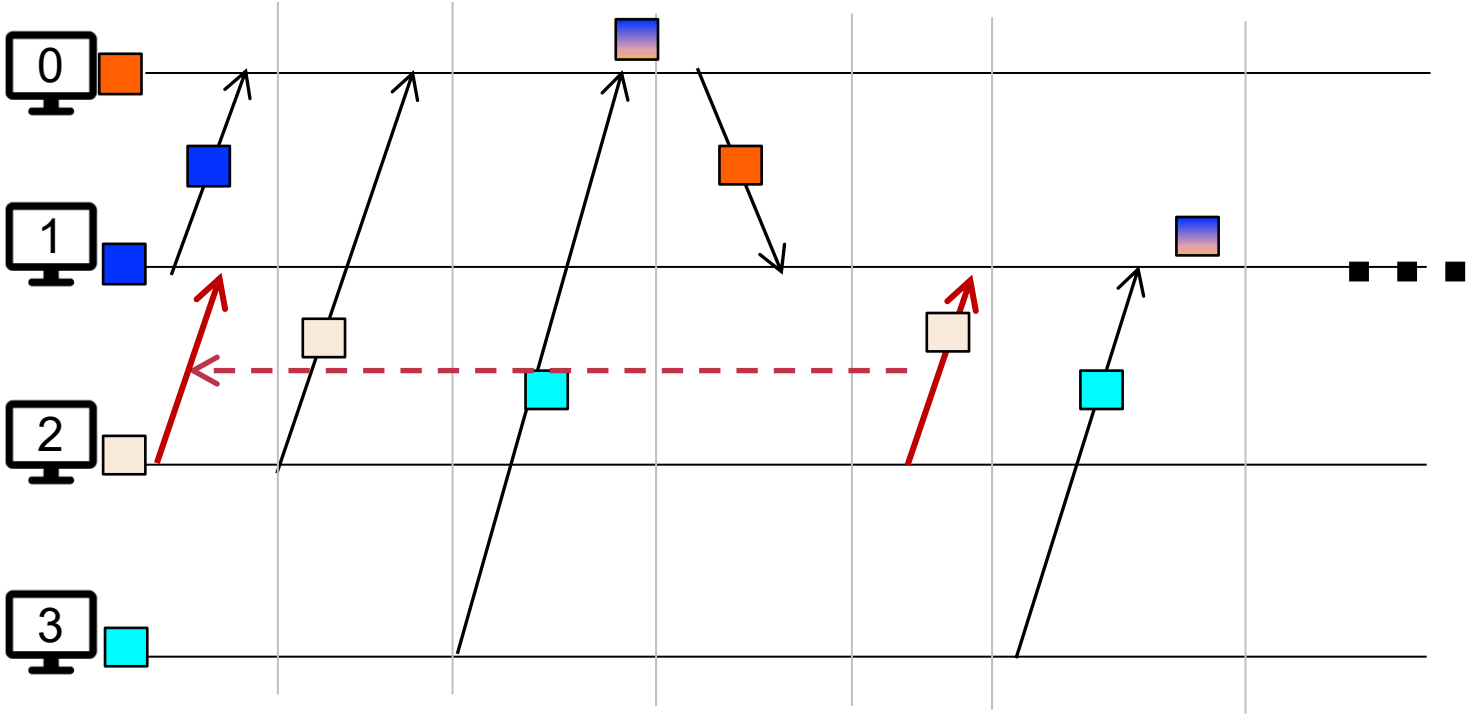
Question

- Can we do better?

Optimized Decentralized AllReduce

Each node finishes its allreduce, one-by-one

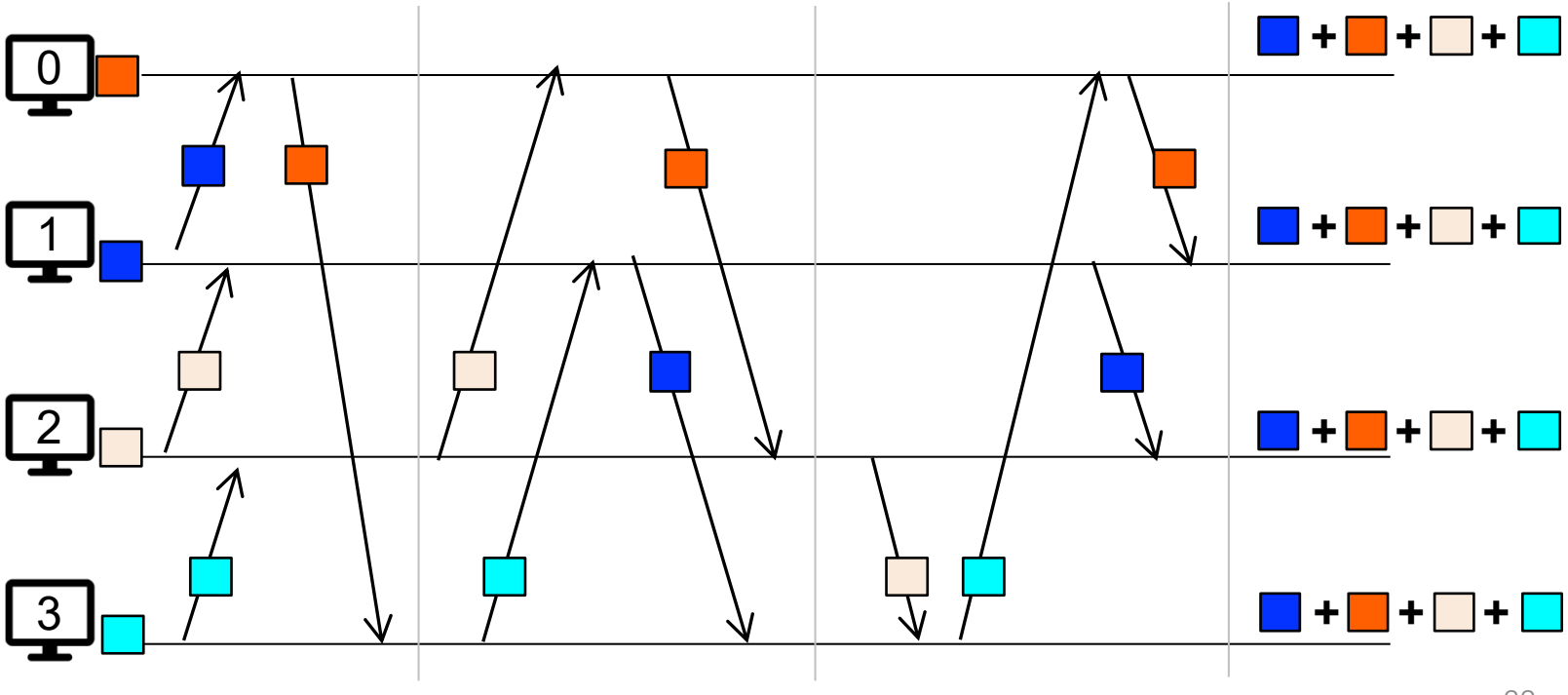
– Communications from future rounds can be moved earlier if w/o contention



Optimized Decentralized AllReduce

Each node finishes its allreduce, one-by-one

– Communications from future rounds can be moved earlier if w/o contention



Analysis of Optimized Decentralized AllReduce

N: total parameters, P: # processors

Total communication per-machine

- **N * (P – 1) data (Bad …)**
- O(1) fan-in (Good!)

Total communication rounds: ~~O(P * P)~~ → O(P)

- Higher than the parameter server approach (O(1)) , but w/o contention

Question

- Can we do better?

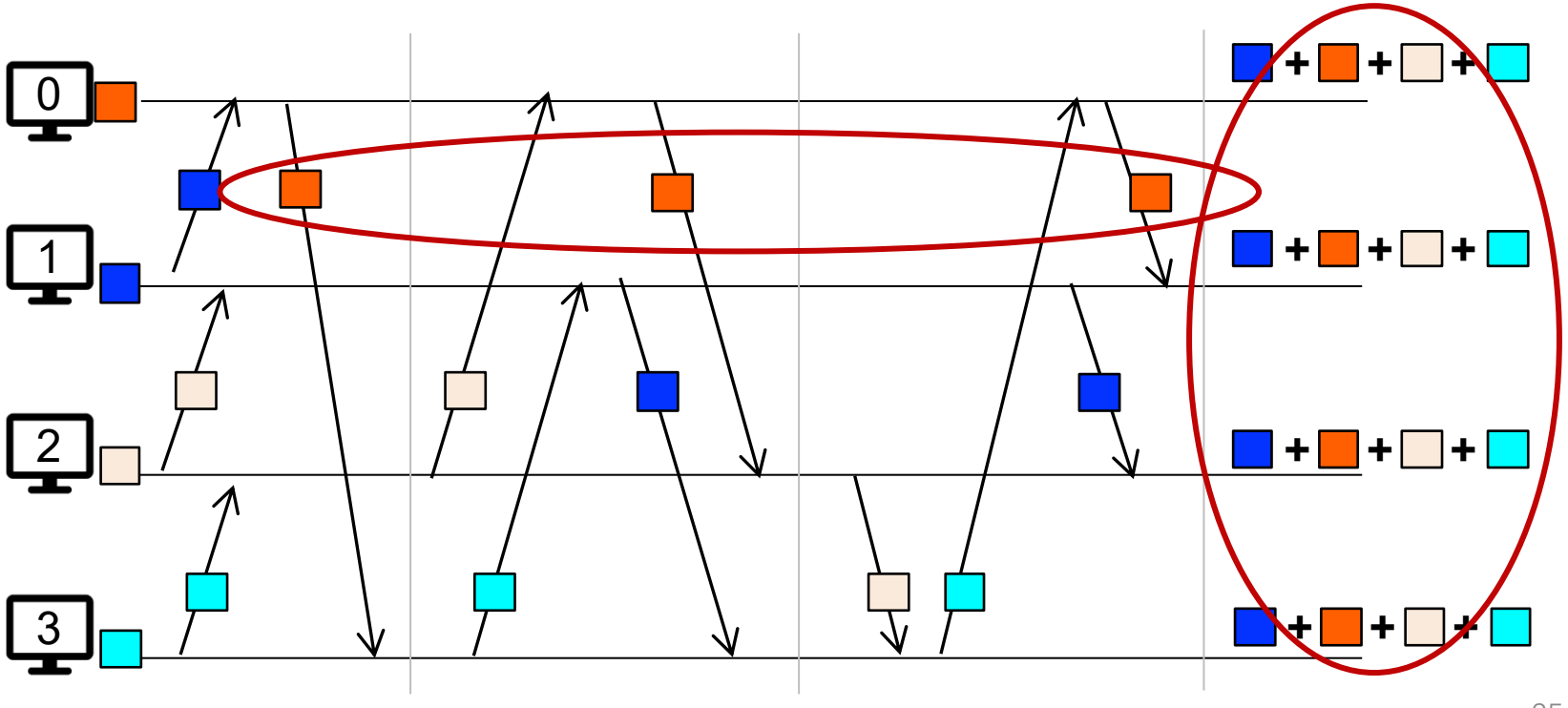
Goal: reduce the payload sent per-process

Idea: Partition Data to Reduce Traffic/Computation

Each node finishes its (a partition of its) allreduce, one-by-one

- Finally, using a decentralized approach to gather all the data

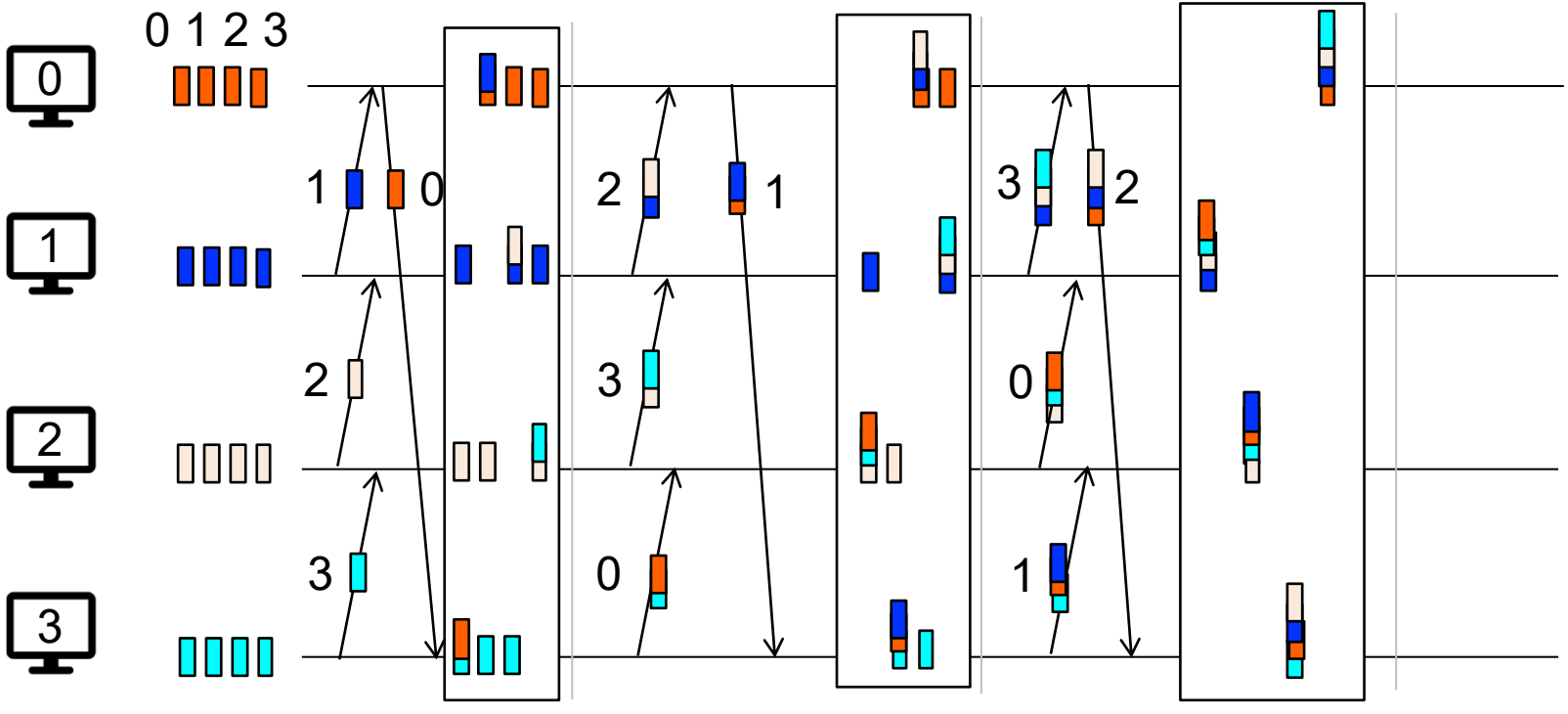
Redundant computations



Try #3: Ring AllReduce

Decentralized reduce, each process reduce on a partitioned data (prev/next to it)

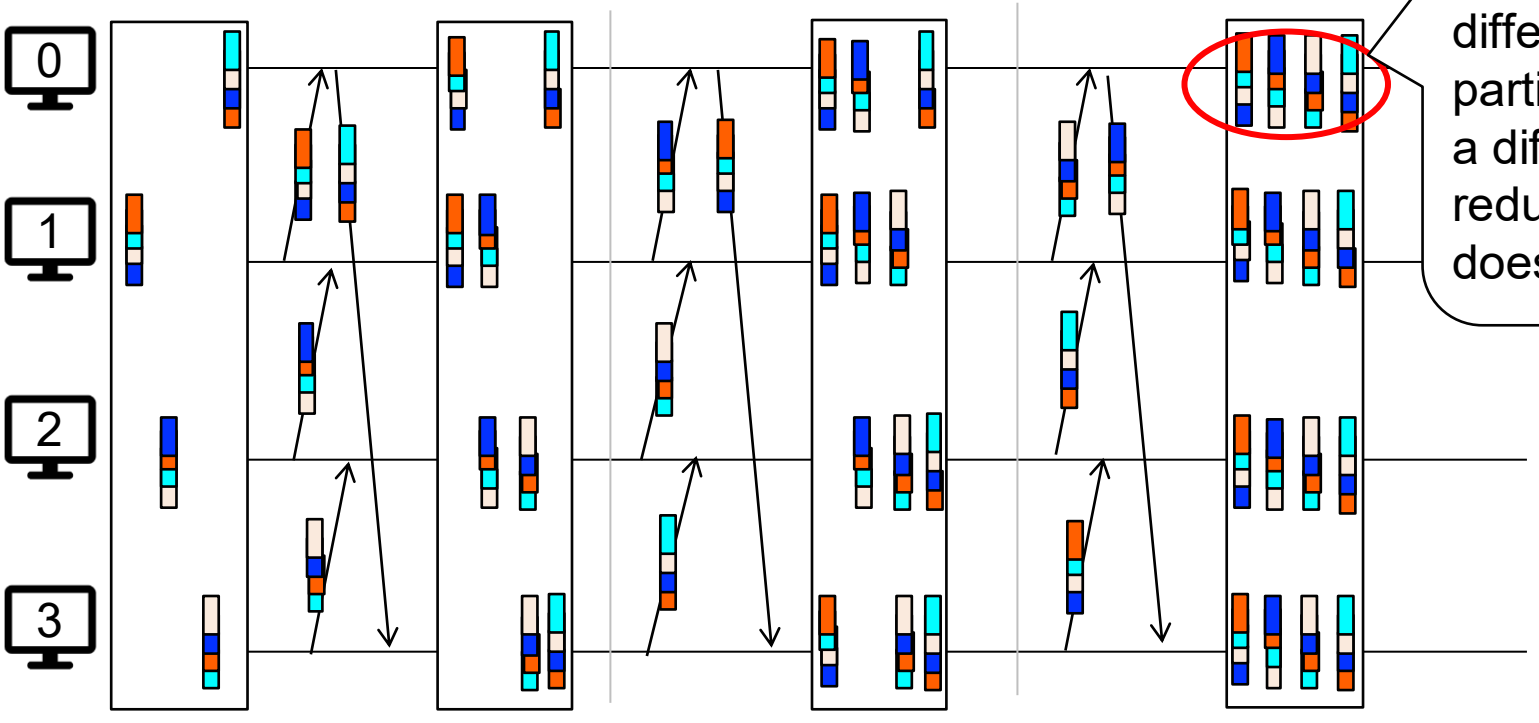
– i.e., each process send one partition data sequentially



Ring AllReduce

After the individual reduce, each process can scatter to collect results

– Eventually, all the processes will have the same reduced data



Question: different partition has a different reduce order, does it ok?

Why Ring AllReduce Can Work?

Assumption: reduce op is associative & communicative

- So we can reorder them to reduce sudden loads at a node
- E.g., we don't need to collect all the data to do the reduce

Otherwise, we cannot reorder the computation

Analysis of Ring AllReduce

N: total parameters, P: # processors

Total communication per-machine

- ~~**N * (P – 1)**~~ → **2 * (P-1) * N / P**: the same as partitioned
- O(1) fan-in

Total communication rounds: O(2 * P)

- Much higher than the parameter server approach (O(1))
- A trade-off for reducing network contention due to high fan-in

What are the drawback?

- High latency due to extra round

AllReduce: Put It All Together

Trade-off between rounds vs. fan-in performance

	Round	Peak node bandwidth	Per-node fan-in
Parameter server (PS)	**O(1)**	**O(2N)**	O(P)
Decentralized Allreduce	O(P)	O(N*P)	**O(1)**
Ring allreduce	O(2*P)	**O(2N)**	**O(1)**

More Reduce Methods Exist

Key design goals

- Reduce payload sent per-node
- Reduce fan-in to avoid contention on network resource
- (new) Reduce rounds (grow linearly w/ the number of processors)
- (new) Better suits the underlying hardware (Cloud & HPC)

Still an active research field

- E.g., what if the hardware speed between different machines diff?
- What if the link speed changes overtime?

Summary: Data Parallelism

Data Parallelism: Distribute the processing of data to multiple PEs

Pros	Cons
Scalability (Large-scale Data)	Synchronization Overhead
Easy Implementation	Model Duplication

Drawback: model (parameters) are replicated on all PEs

Target: larger dataset but smaller models

MODEL PARALLELISM

Model Parallelism

Data Parallelism: *Distribute the processing of data to multiple PEs.*

$$w^1 = w^0 - \frac{\alpha}{B} \sum_{i=1}^{B} \frac{\partial \mathcal{J}(w^0)}{\partial w}$$

Model Parallelism: *Break the model and distribute processing of every layer to multiple PEs*

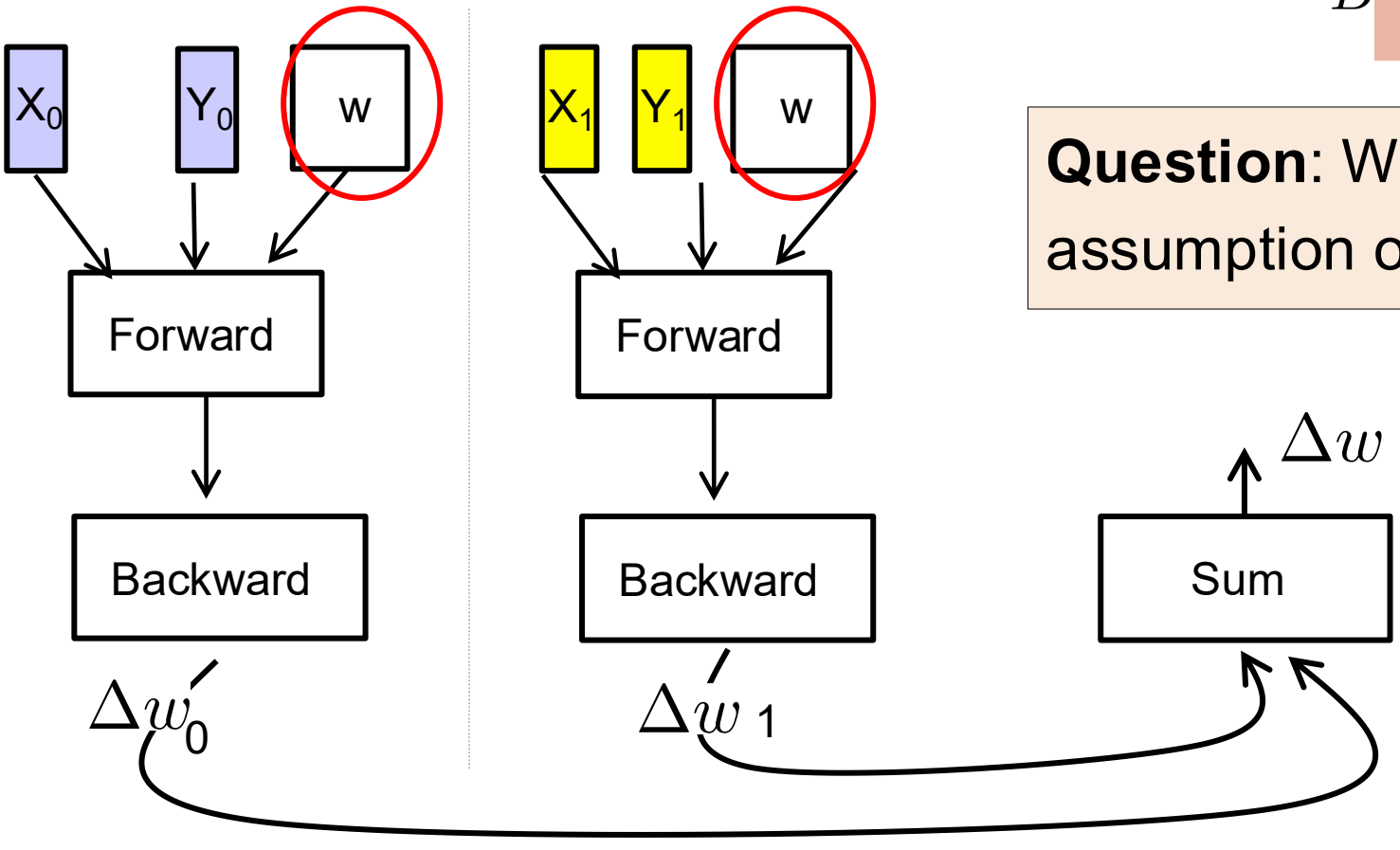
$$w^1 = w^0 - \frac{\alpha}{B} \sum_{i=1}^{B} \frac{\partial \mathcal{J}(w^0)}{\partial w}$$

For either approach it is also possible to use ***synchronous*** *or* ***asynchronous*** *updates*

$$w^1 = w^0 - \frac{\alpha}{B} \sum_{i=1}^{B} \frac{\partial \mathcal{J}(w^0)}{\partial w}$$

Drawback of Data Parallelism

$$w^1 = w^0 - \frac{\alpha}{B}\sum_{i=1}^{B}\frac{\partial \mathcal{J}(w^0)}{\partial w}$$



Question: What is the assumption of W?

Drawback of Data Parallelism: Replicated Model

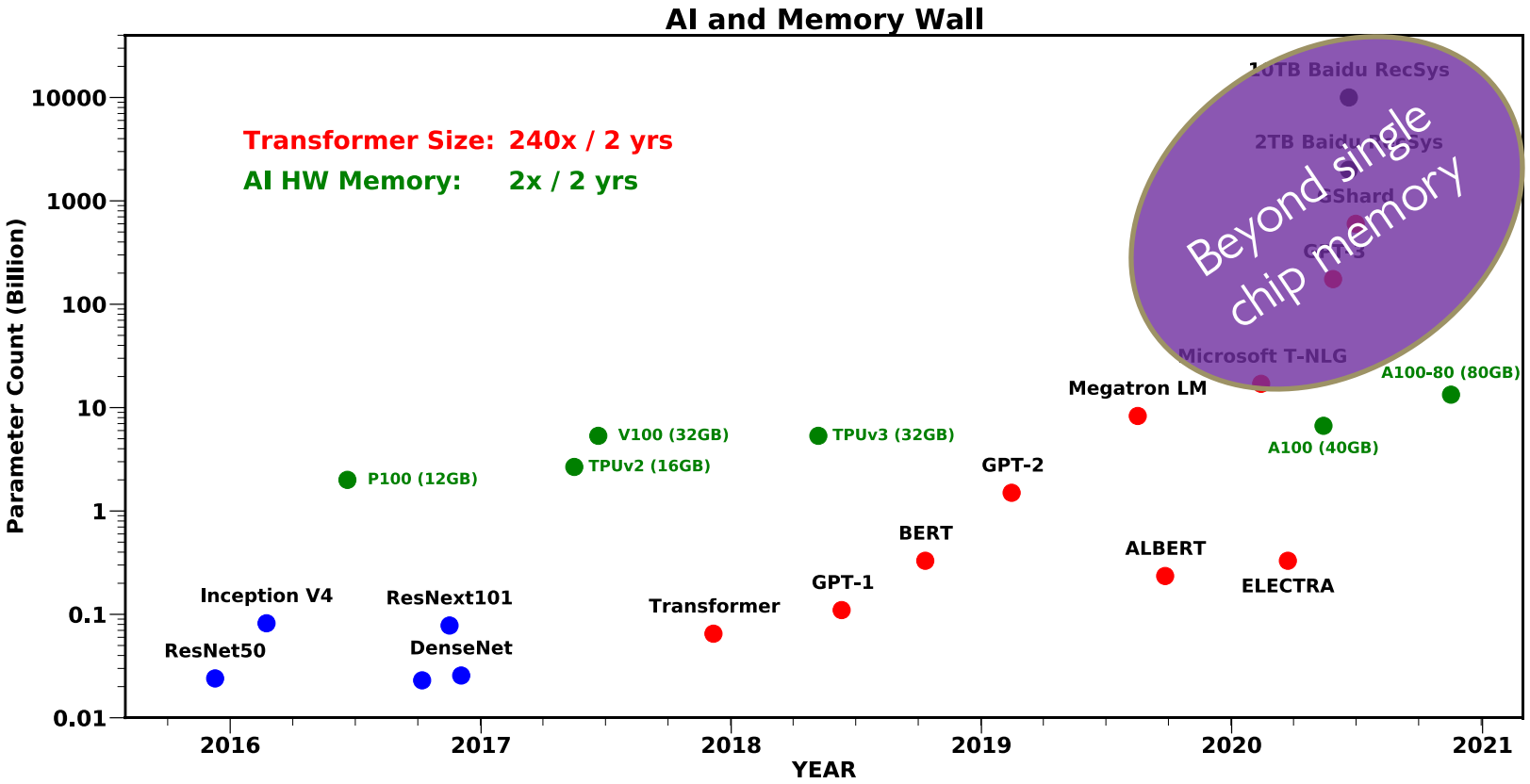
Data parallelism assumes the model (parameters) are replicated on all the processes

- Such that they can do the forward & backward pass concurrently
- What if the model cannot fit onto the device?

Current trends: models are becoming larger and larger

- No country (or company) can tolerate the risk of falling behind in the AI race

Trend: Training Large Models



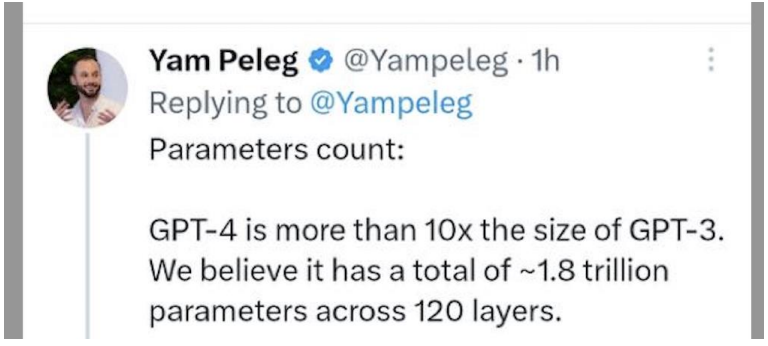
Amir Gholami, Zhewei Yao, Sehoon Kim, Michael W. Mahoney, Kurt Keutzer, AI and Memory Wall, Riselab Medium Blogpost, 2021.

Trend: Training Large Models

Typically, also challenging because AI models are really really large

Example: GPT-4

- **1.8 trillion parameters**
- **120 layers**
- Assuming 4B for each parameter, we have 60 GB per-layer



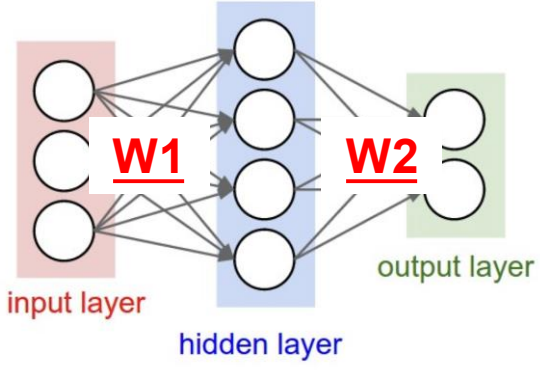
What are the memory capacity of GPU?

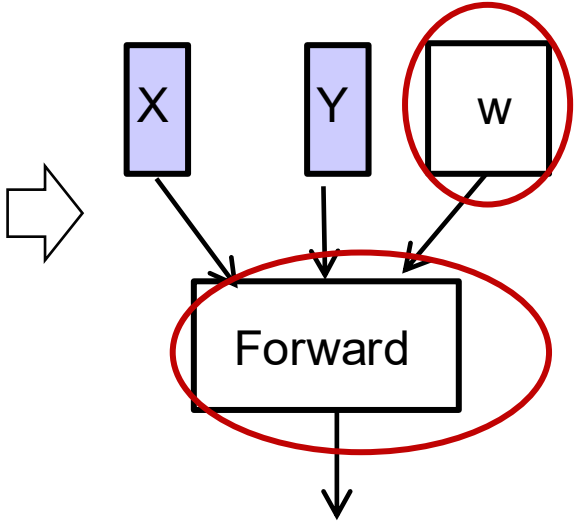
- 80GB for H100, 140GB for H200
- Note that we also need to store **activations & input data (& optimization state)**

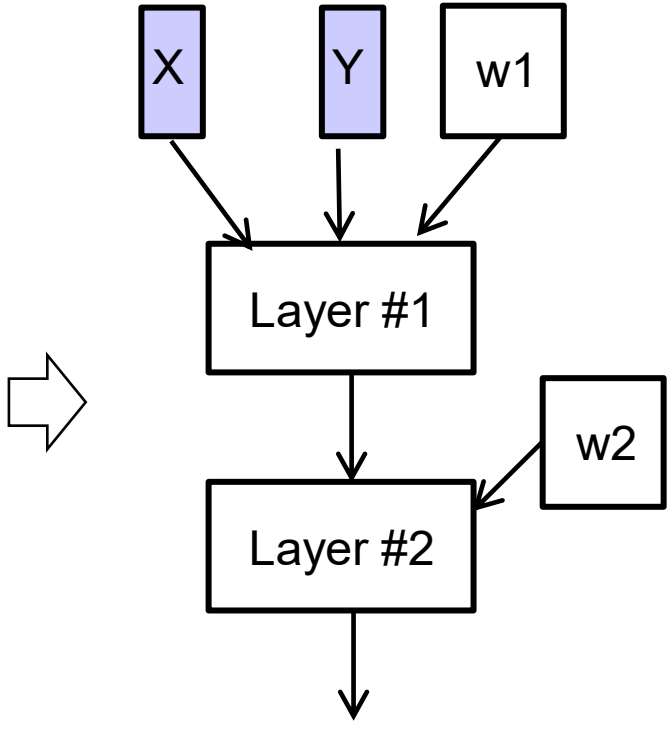
Model Parallelism

Partition model parameters, typically done in two ways:

- **Partition on the W**: tensor parallelism
- **Partition on the layer**: pipeline parallelism



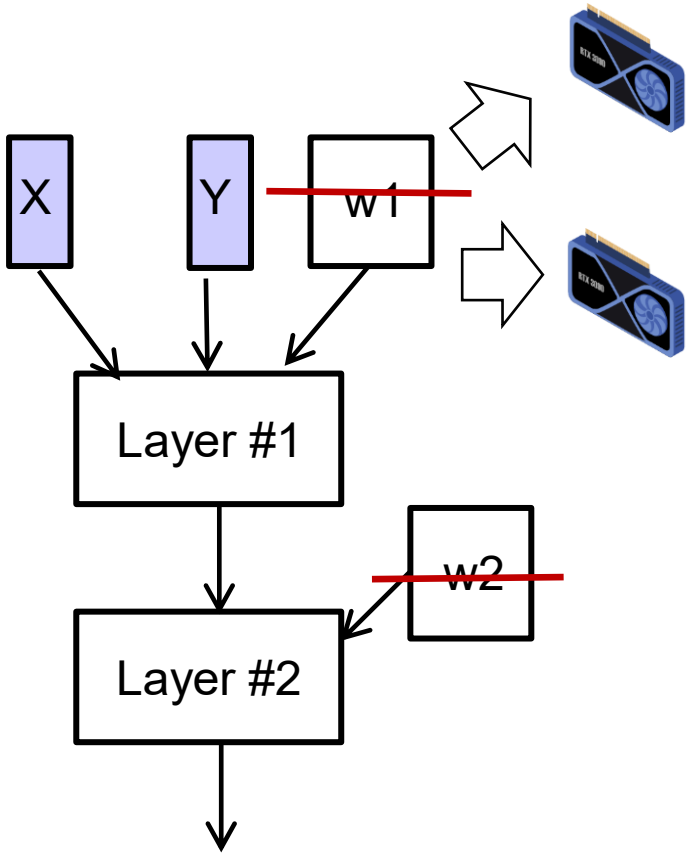




Tensor Parallelism

Partition the parameters of a layer

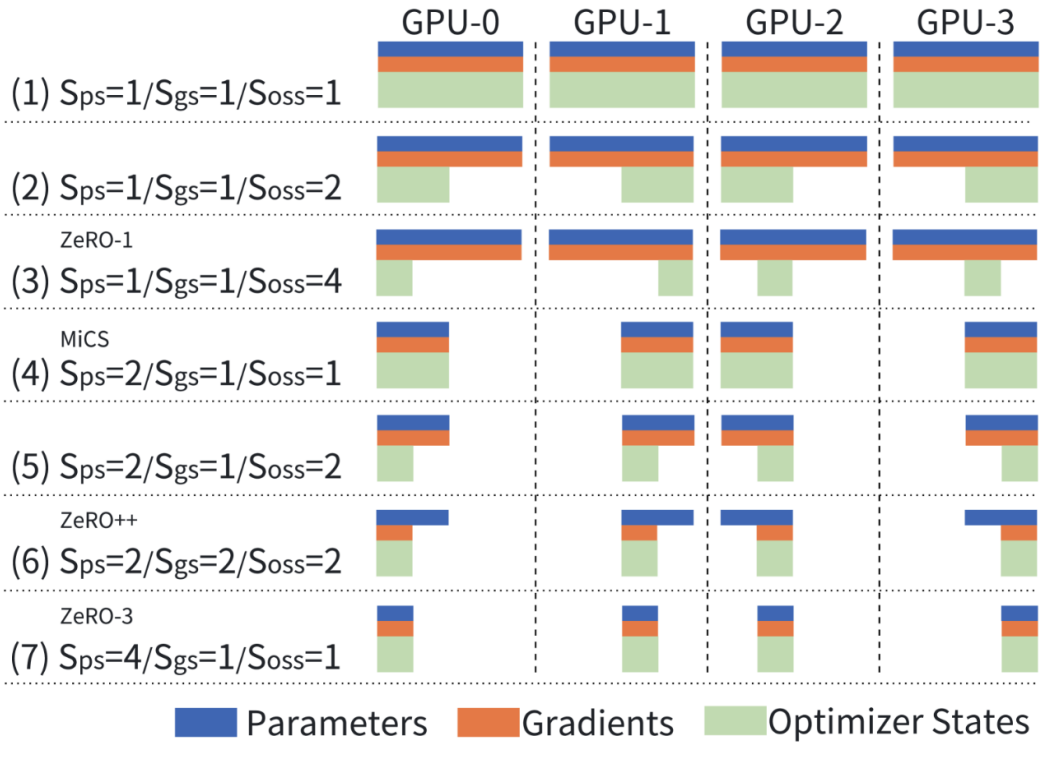
- Each partition is deployed on a separate GPU
- **Reduced Memory Constraints**
- Allow for training **larger models** by leveraging multiple devices



Tensor Parallelism

What to partition?

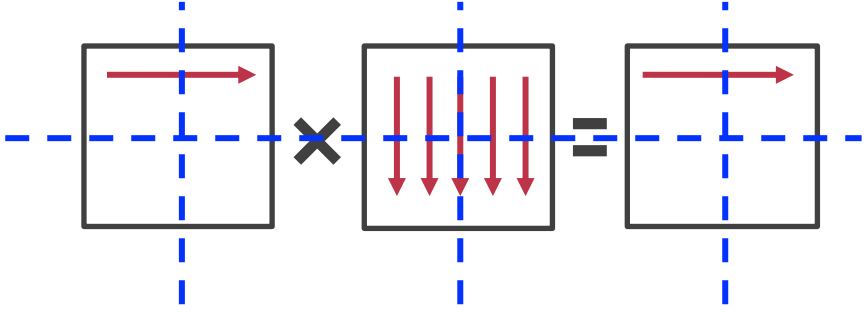
- Parameters
- Gradients
- Optimizer States
- Activations
- Operators
- . . .

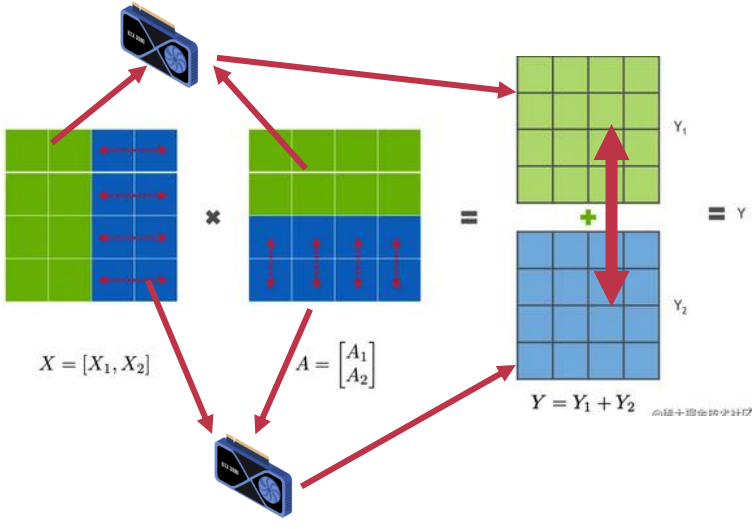


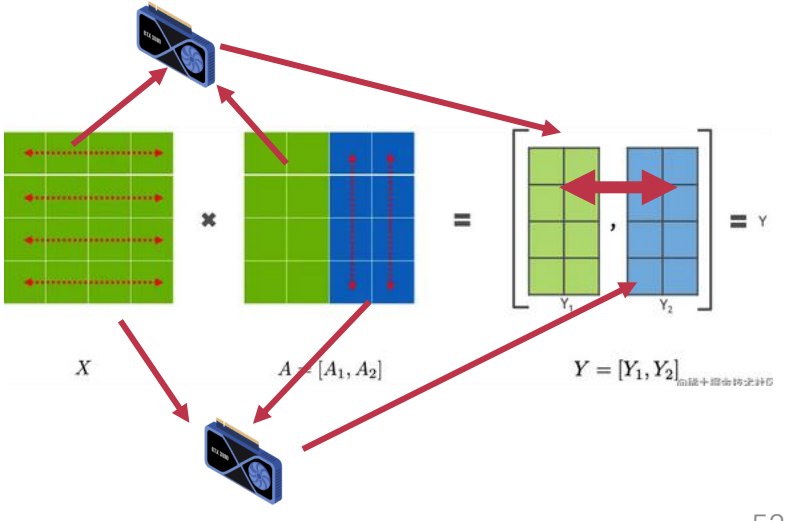
Tensor Parallelism

How to partition?

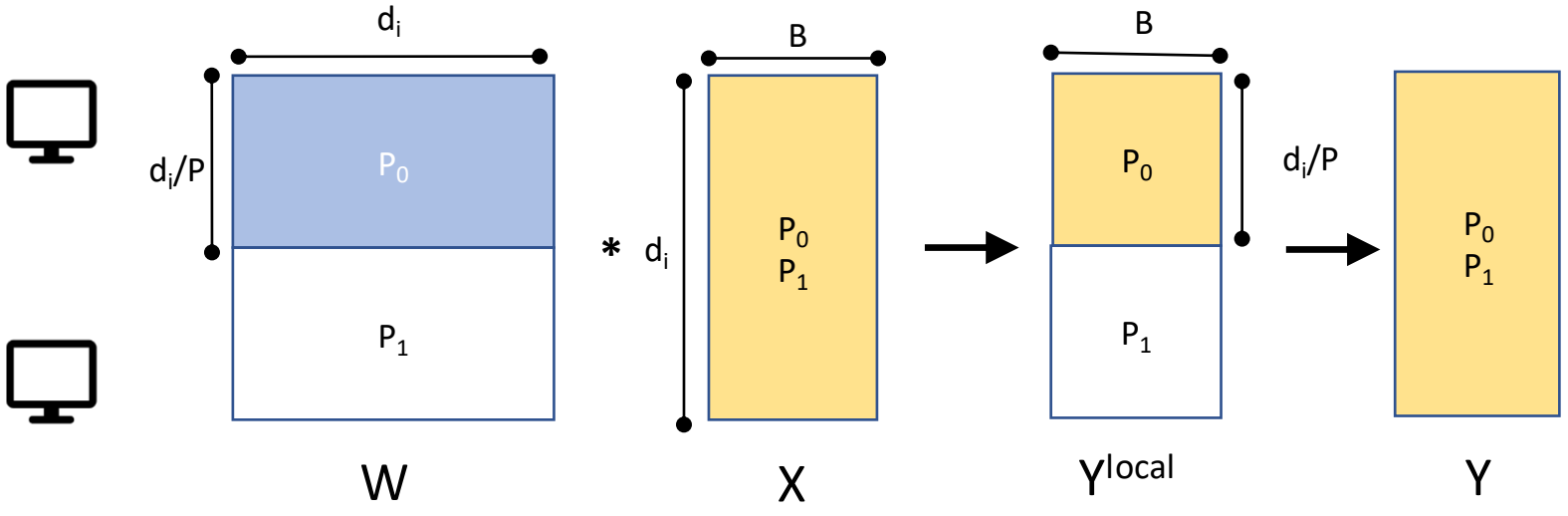
- Row/Column-wise partitioning
- Others: block-wise, tiling, 1D/2D, ..
- Trade-off: memory, communication, …





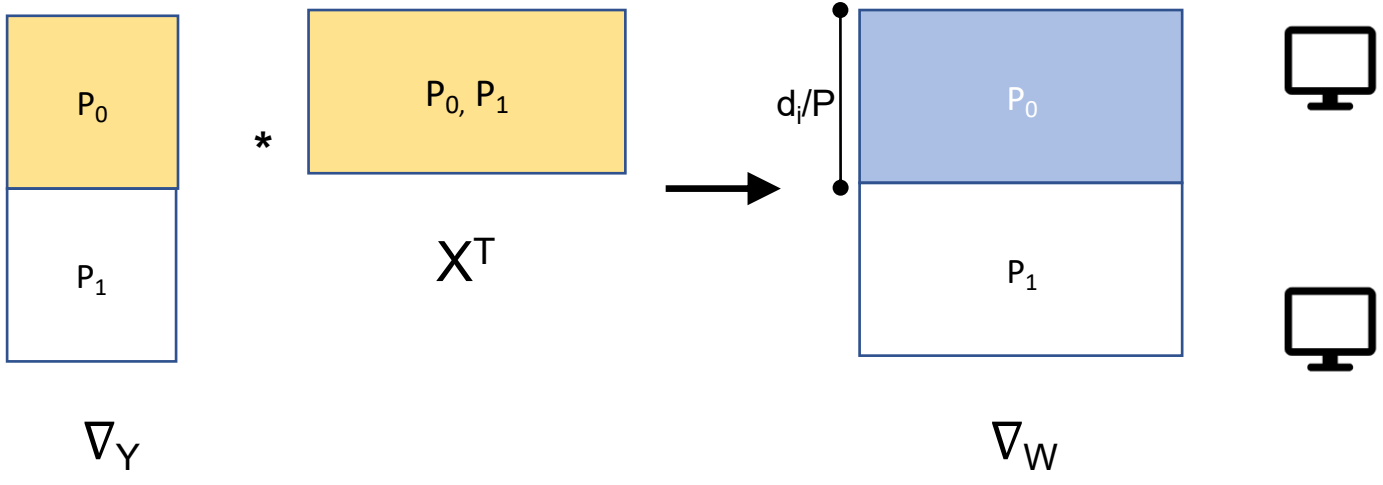


Forward Pass



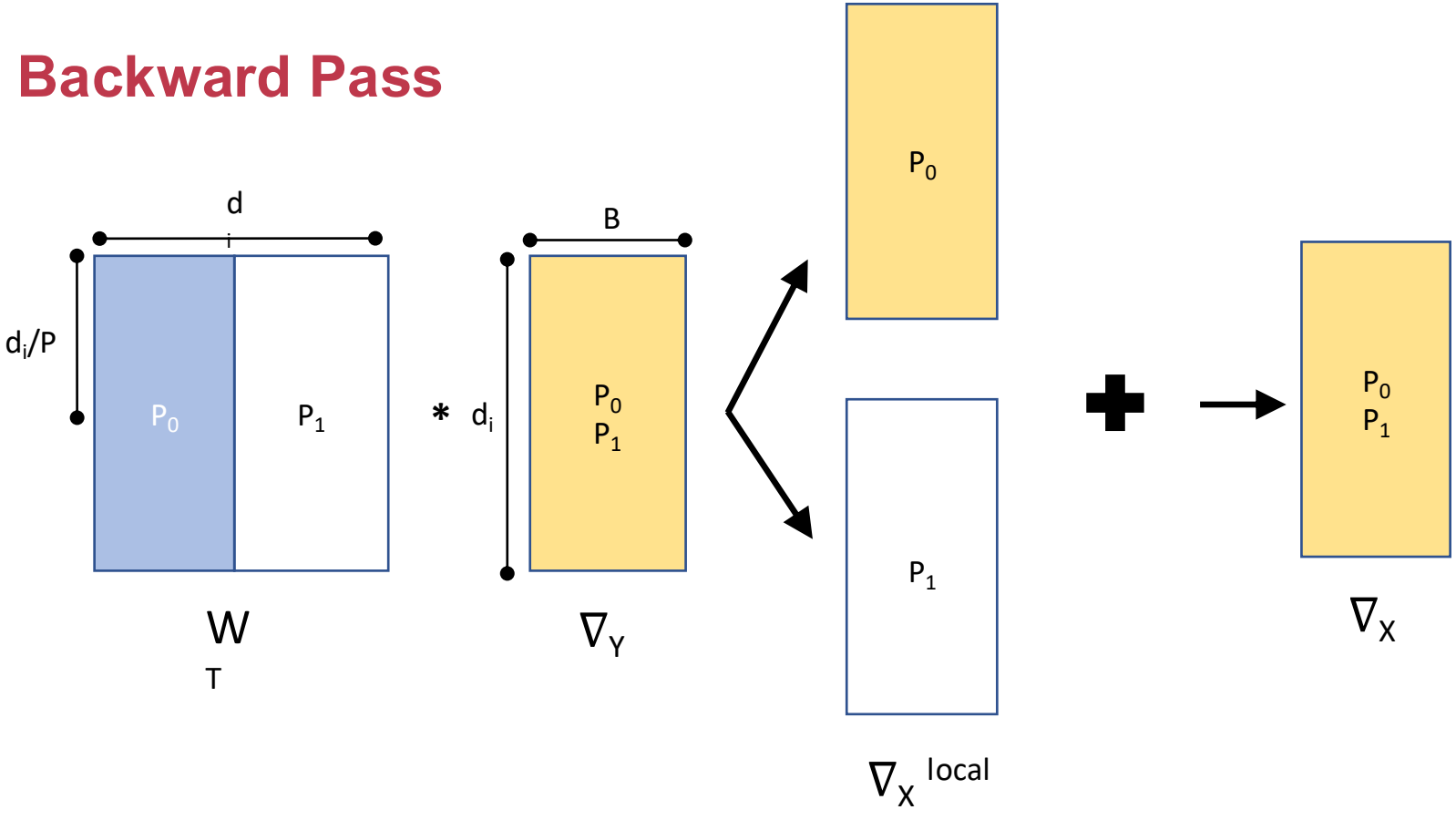
- Requires an all gather communication so that all processes get each others activation data (Y^{local})
- Suppose with a simple forward, each node needs to send $(\frac{di \times B}{P}) \times (P - 1)$ ~= $di \times B$

Backward Pass



- No communication needed as every processor only needs the gradient of its own parameters

Backward Pass



- Aggregating input gradient requires an **allreduce** operation
- Cost (using ring): $2 \times (\frac{di \times B}{P}) \times (P - 1) \cong 2 \times (di \times B)$

Communication Analysis

Suppose W is di x di**, and X is** di x B**, we have** P **partitions**

Setup: partition the graph into P **partitions w/ model parallelism**

- What is the expected communication needed for a link?

Forward pass of tensor parallelism:

- (di * B / P) * (P – 1) // must communicates with all the partitions

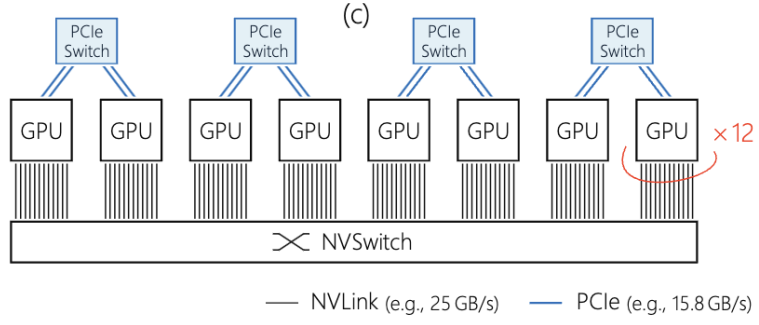
Backward pass of tensor parallelism:

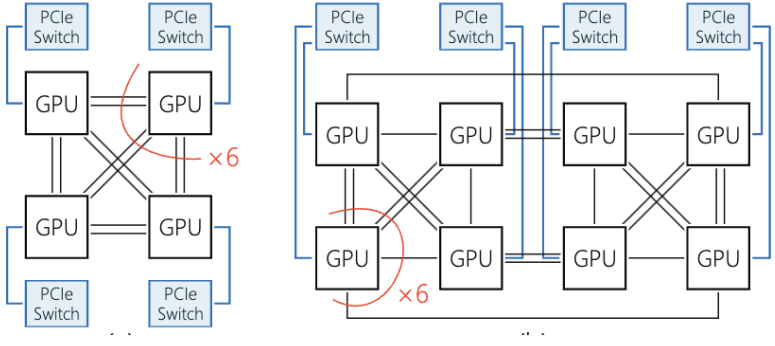
- 2 * (di * B / P) * (P – 1) // must communicates with all the partitions

Hardware Architecture of Modern Servers

Hierarchical and hybrid networking

- PCIe, NVLink, and NVSwtich
- Hard-wired and switch-based
- Ethernet, RDMA, CXL, ..





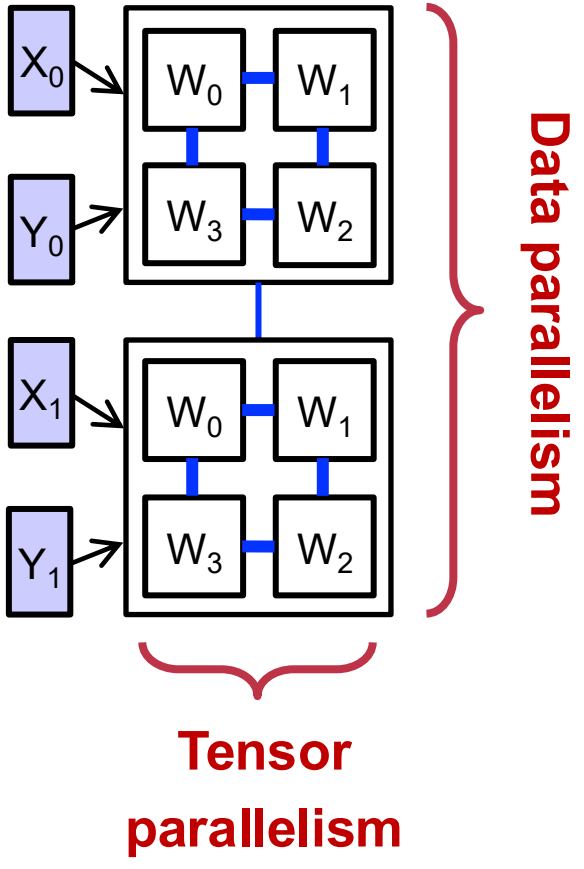
Hardware Architecture of Modern Servers

Hierarchical and hybrid networking

- PCIe, NVLink, and NVSwtich
- Hard-wired and switch-based
- Ethernet, RDMA, CXL, ..

Hybrid parallelism over hybrid networking

- **Inter-node**: data parallelism
- **Intra-node**: tensor parallelism



Summary: Tensor Parallelism

Tensor Parallelism: Distribute tensors of model to multiple PEs

Pros	Cons
Scalability (Large-scale Model)	Complex Implementation
Memory Efficiency	Communication Overhead

Drawback: higher communication cost

Target: Larger models with Fast interconnects

Next Lecture

Topic: Accelerating Distributed AI Infrastructure Systems (II)