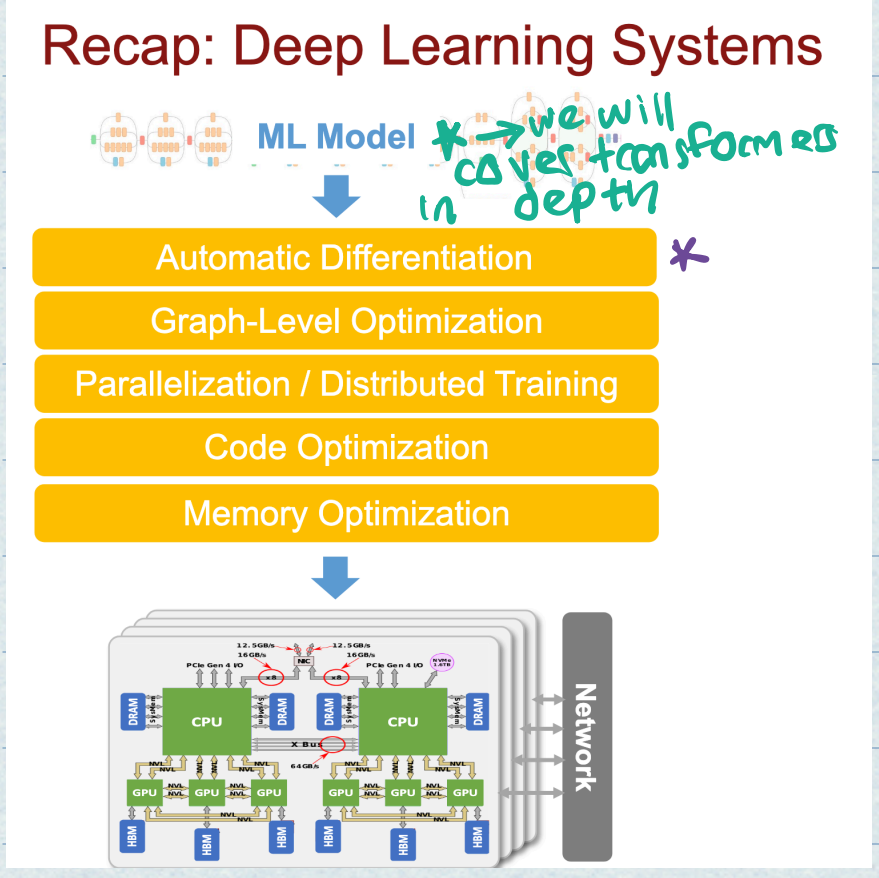
* Agenda:

- Overview of Topics we will cover in the class
- Data Parallelism
  - ↳ Parameter Server
  - ↳ All Reduce Algorithms

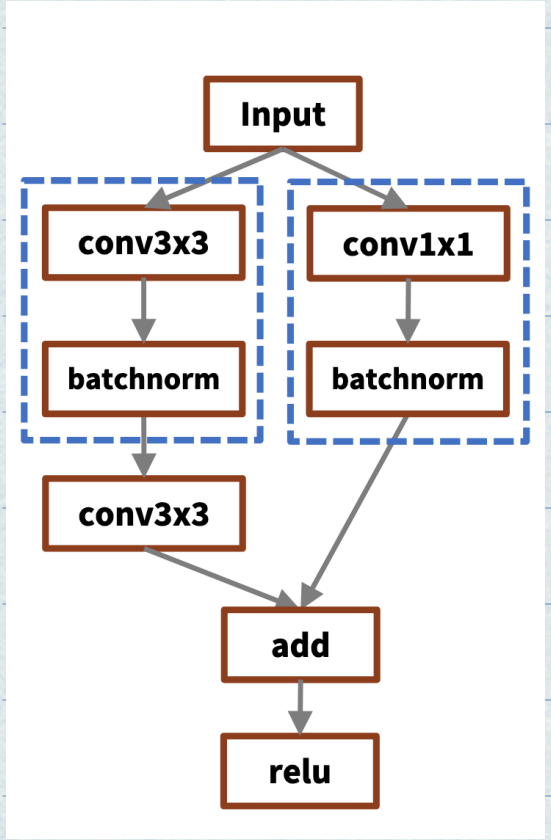
Recap: Deep Learning Systems



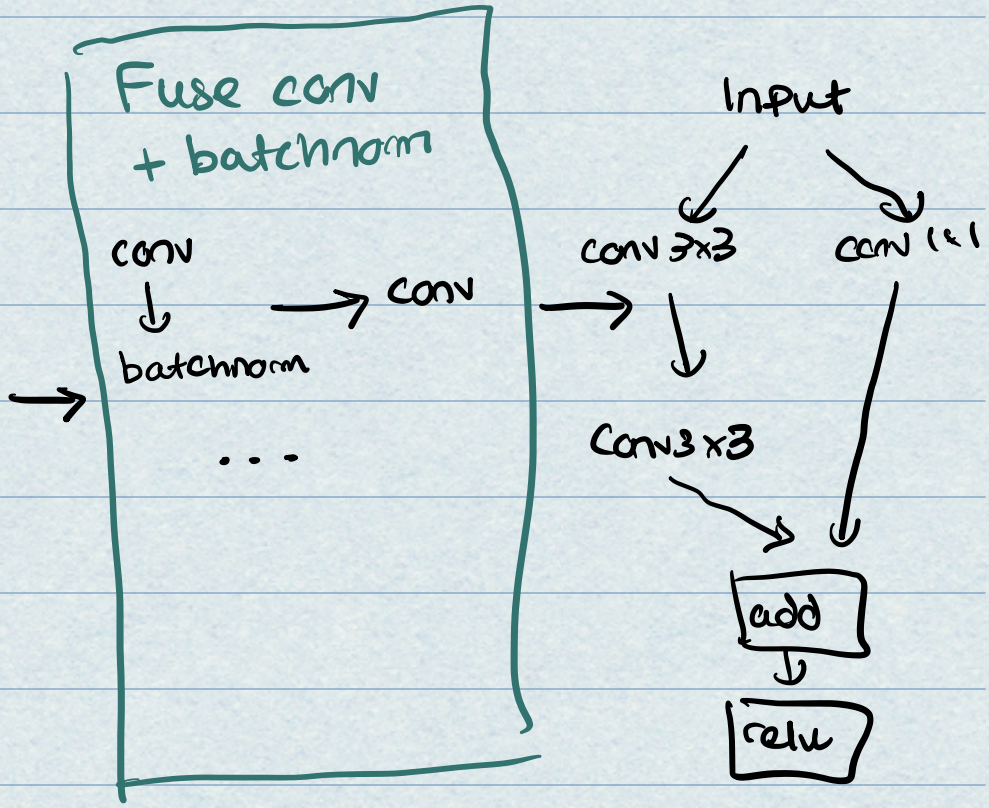
Auto Diff (from last time):

- automatically construct FW and BW computation GRAPH
- Implication: 1 unified FW/BW pass to discover weight updates (vs. separate for each weight)

* Graph-Level Optimizations

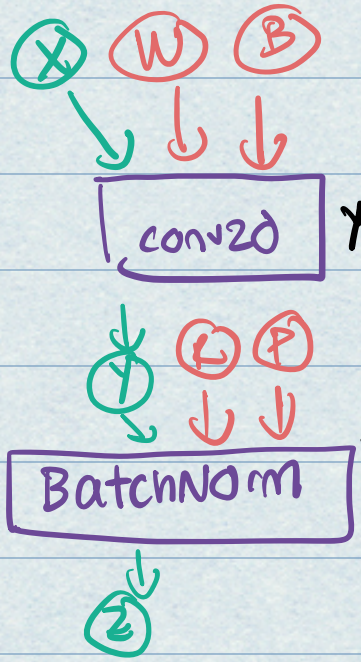


Input Computational Graph



Potential graph transformations

* Example: Fusing Conv and Batch Norm

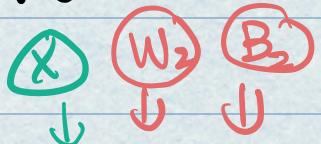


- W, B, R, P: constant pre-trained weights
- X, Y, Z involve learned params (d,u,v)

$$Y(n,c,h,w) = \sum_{d,u,v} \left(X(n,d,h+u,w+v) \cdot W(c,d,u,v) \right) + B(n,c,h,w)$$ -> not insum

$$Z(n,c,h,w) = Y(n,c,h,w) \cdot R(c) + P(c)$$

* We can Fuse conv and Batchnorm



$$W_2(n,c,h,w) = W(n,c,h,w) \cdot R(c)$$

conv2d

$B_2(n,c,h,w) = B(n,c,h,w) \cdot R(c) + P(c)$

(intuition): $((X \cdot W) + B) \cdot R + P$

$= (X \cdot W \cdot R) + (B \cdot R + P)$

$$Z(n,c,h,w) = \left(\sum_{d,u,v} X(n,d,h+u,w+v) \cdot W_2(c,d,u,v) \right) + B_2(n,c,h,w)$$

- Discussion Q: Why is fusion a good idea? where is it applied in other areas of systems?

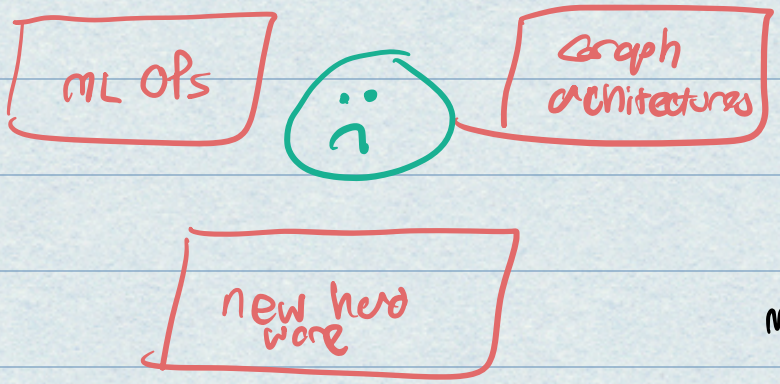
* How do these rules get applied?
  - tensorflow has ~200 rules (53,000 LOC!)
  - includes:
    - Fuse conv + relu
    - Fuse conv + batchnorm
    - Fuse ^multi conv

* What are limitations of rule based optimizers?

1) Robustness: heuristics may not always make things faster

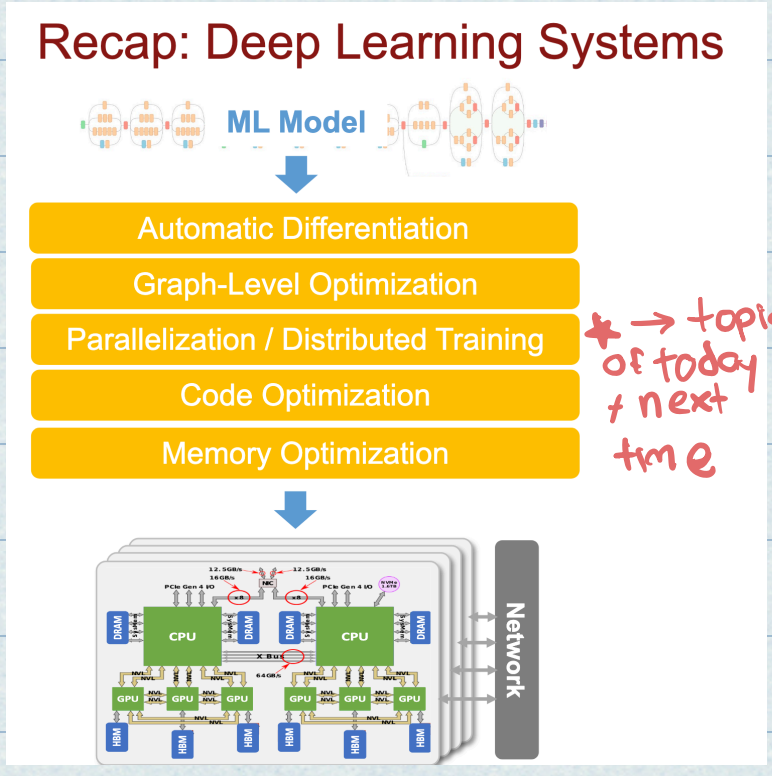
2) scalability of implementation:
   - tensorflow uses ~4K LOC to optimize convs

3) Still can miss some subtle optimizations



→ Week 7: infeasible to manually design all these optimizations

Recap: Deep Learning Systems



* → topic of today + next time

Code optimization:

- given a hardware, how do we map a specific operator onto it?

$$C = \sum_{k=0}^{K} A(k,y) \bullet B(k,x)$$

Simple option:

```
for y in range(1024):
  for x in range(1024):
    C[y][x] = 0
    for k in range(1024):
      C[y][x] += A[k][y] * B[k][x]
```

* how are X and Y layed out in memory on the device?

- How is this handled in systems today?
  - HW/SW engineers manually write

operator libraries

- cuDNN, cuSparse, cuRAND, cuBLAS

for GPUs

↳ "cudnnConvolutionFw"

"cublasSGemm"

- Problem: cannot provide support immediately for new ops
- Problem: what if HW changes?

- Week 6 and Project 3:
  - how do we program HW?
  - HW for ML
  - Frameworks to automatically generate HW code

- Overview of whats to come:
  - week 1-2: parallelism
  - week 2-5: Thinking about HW & HW-aware algorithms w/ a focus on attention, revisit parallelism
  - week 6: GPUs
  - Week 7-8: ML frameworks,

Quantization, sparsity

- After spring break: TBD Topics in research

- what are general THemes/tradeoffs?

1) Memory vs. computation

- can redesign procedures to avoid recomputations, but comes at cost OF memory

2) Memory vs. communication

- can scale up. but this will add communication overhead

3) Accuracy vs. memory vs. computation:

- can quantize → how much perf overhead?

4) How do we restructure methods to perform better on HW?

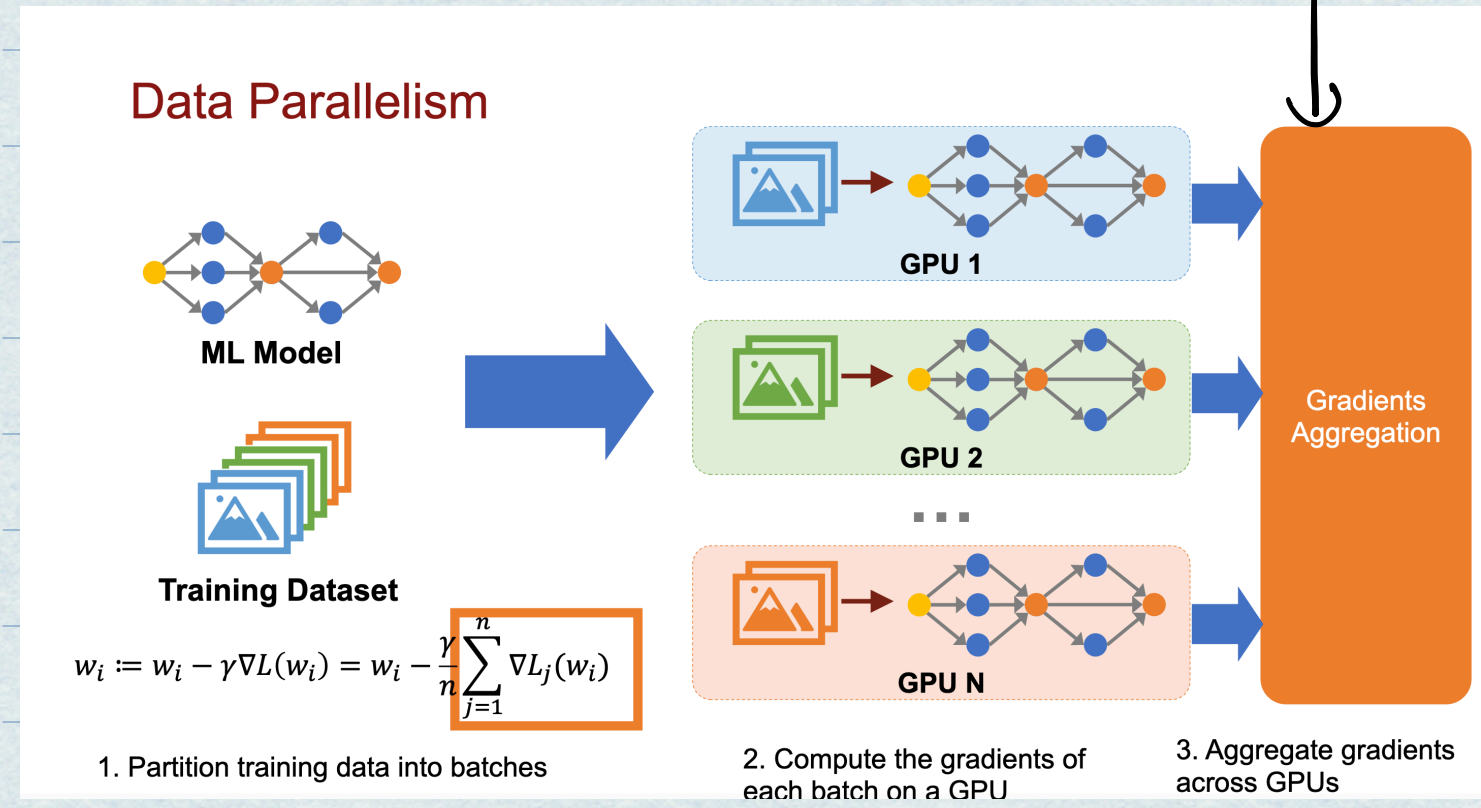
- Level oF alg it self:
  - efficient HW-aware attention
  - using OS paging to improve attention

- Level of computation graph/ code optimizations

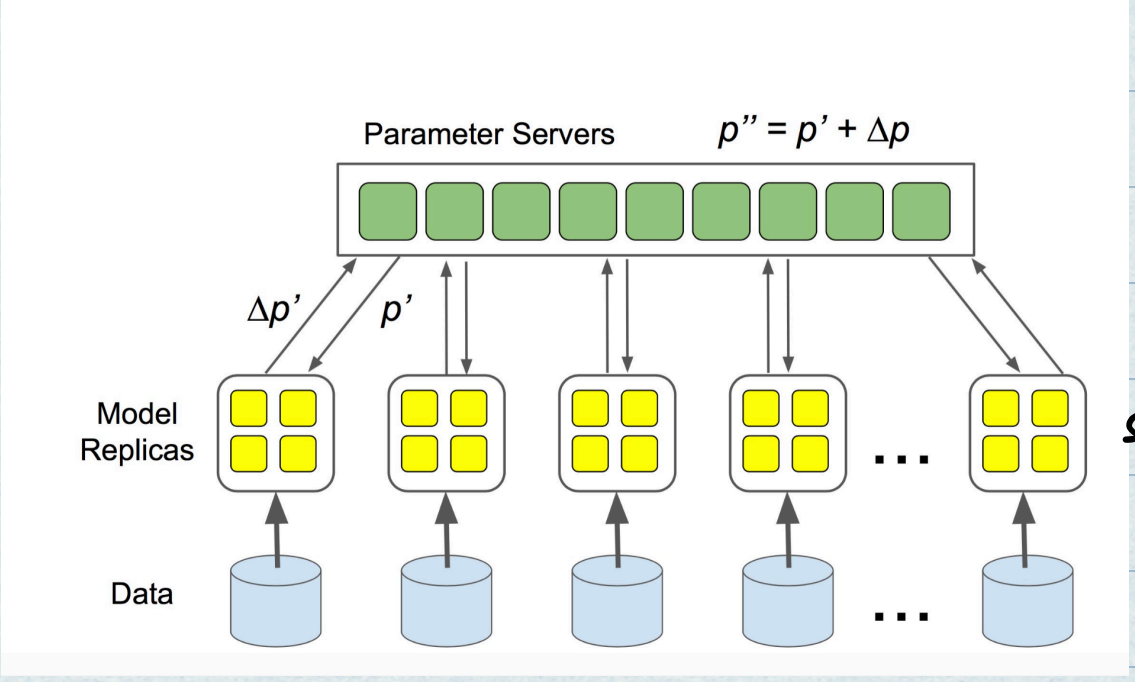
PARALLELISM

- Data Parallelism

what does orange box look like?



- option 1: parameter server



- workers push gradient updates to centralized server and pulls updates back

- each worker must have 1 wire to PS, PS must have N wires

Cons:

- centralized communicatia →

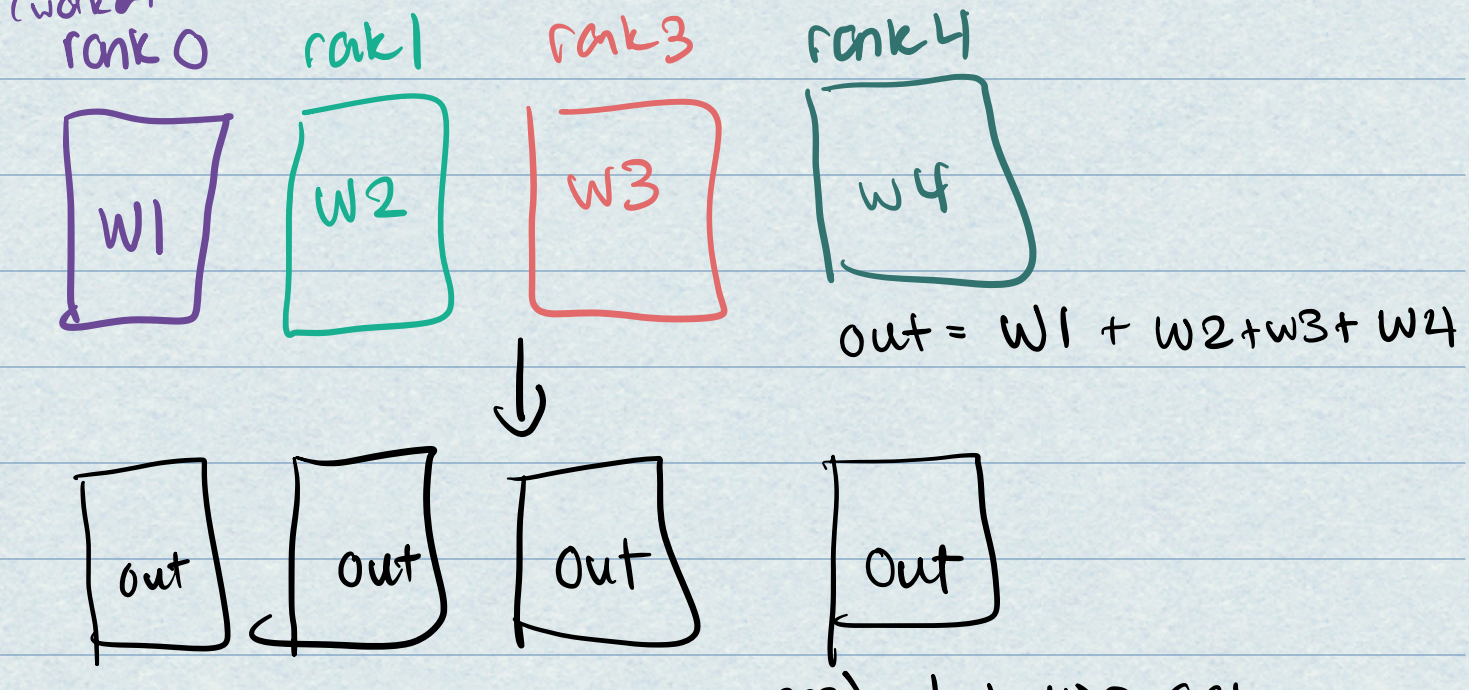
all workers communicate w/ param. server for update; cannot scale to large numbers of workers

Pros:

- can potentially scale # of param. servers by splitting which params each is responsible for

- Can we DECENTRALIZE communication?

  - solution: All-reduce

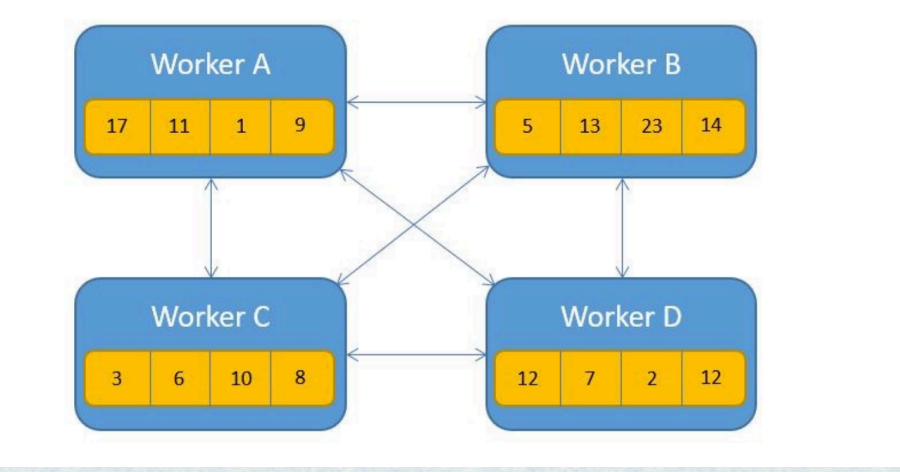


- can also do min, max prod, bitwise ops

- Q: How do we actually do all reduce?

1) Naive All-reduce

- each worker sends its local state to all other workers
- consider: N workers and M params



→ each worker must have N - 1 wires of bandwidth B

→ communication: $O(N \cdot (N-1) \cdot M)$

→ each worker communicates pt-to-pt w/ every other worker:

- imagine param. server w/o server
- same scalability issue

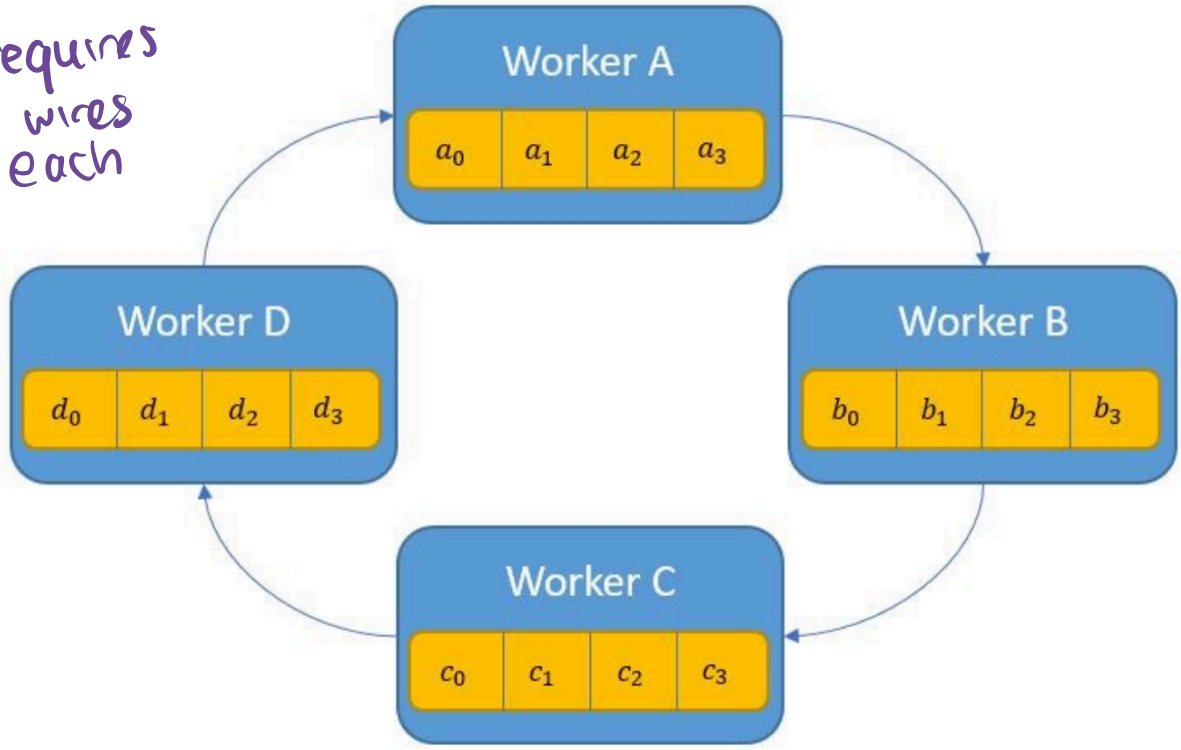
2) Ring all-reduce

- construct ring of N workers, divide M params into N slices

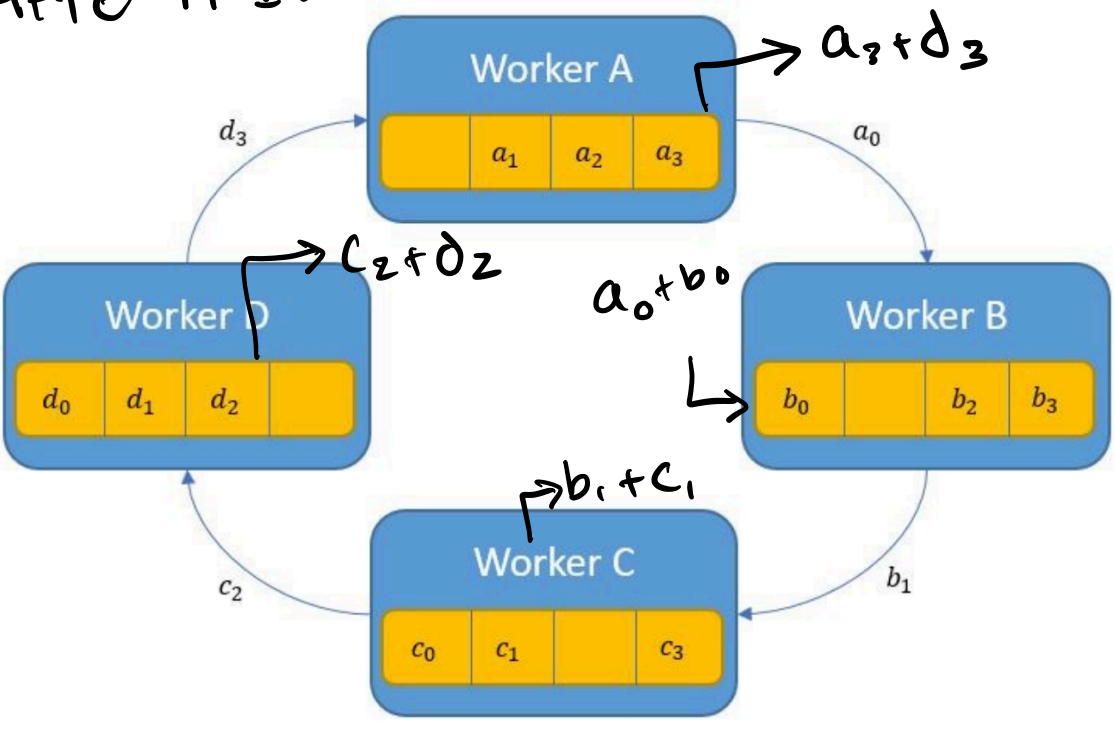
→ step 1: Aggregation: each worker sends a single slice: (M/N) params: to next worker on ring; repeat N - 1 times

→ goal of step 1: end up w/ sum of each slice on 1 of the workers

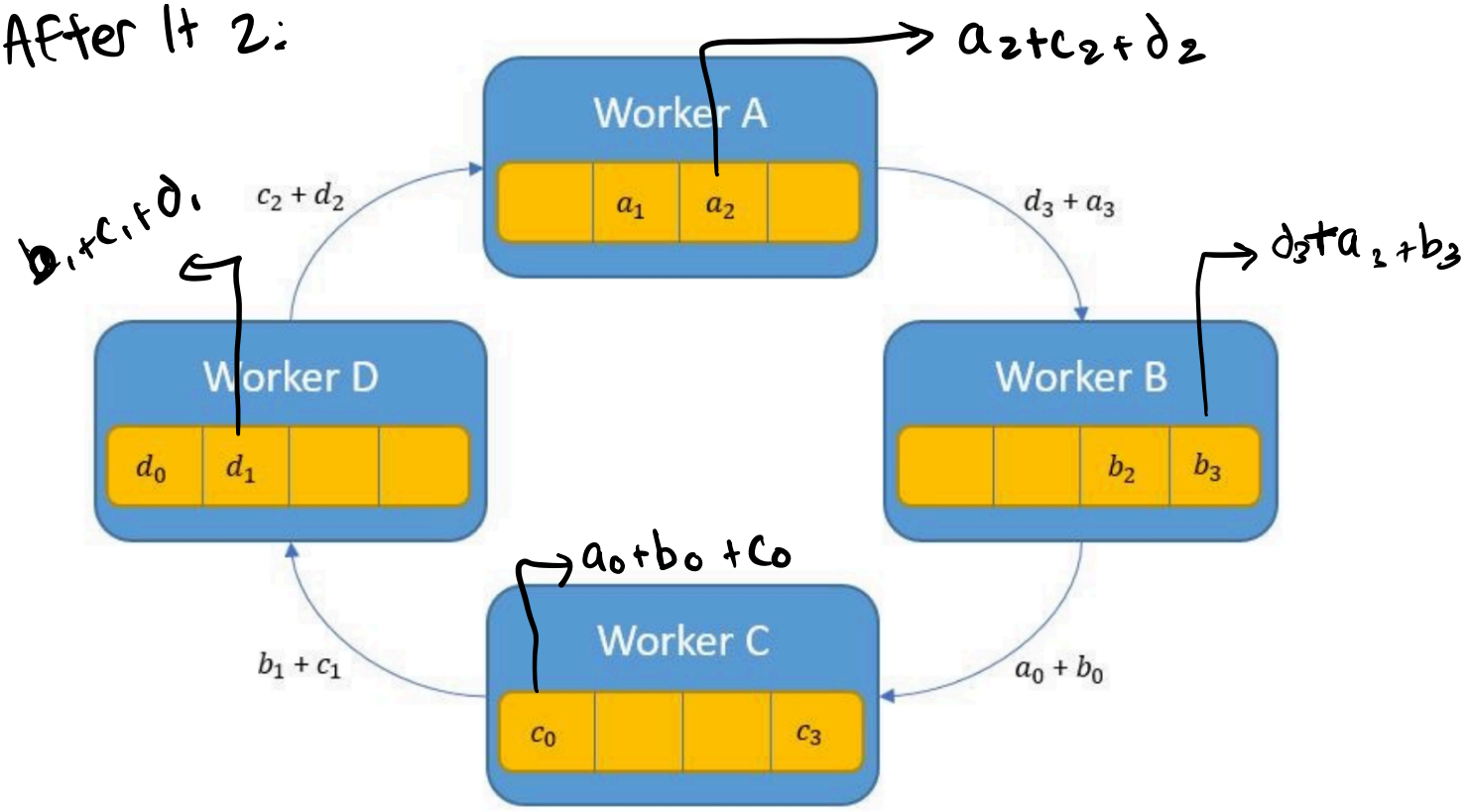
*requires 2 wires each



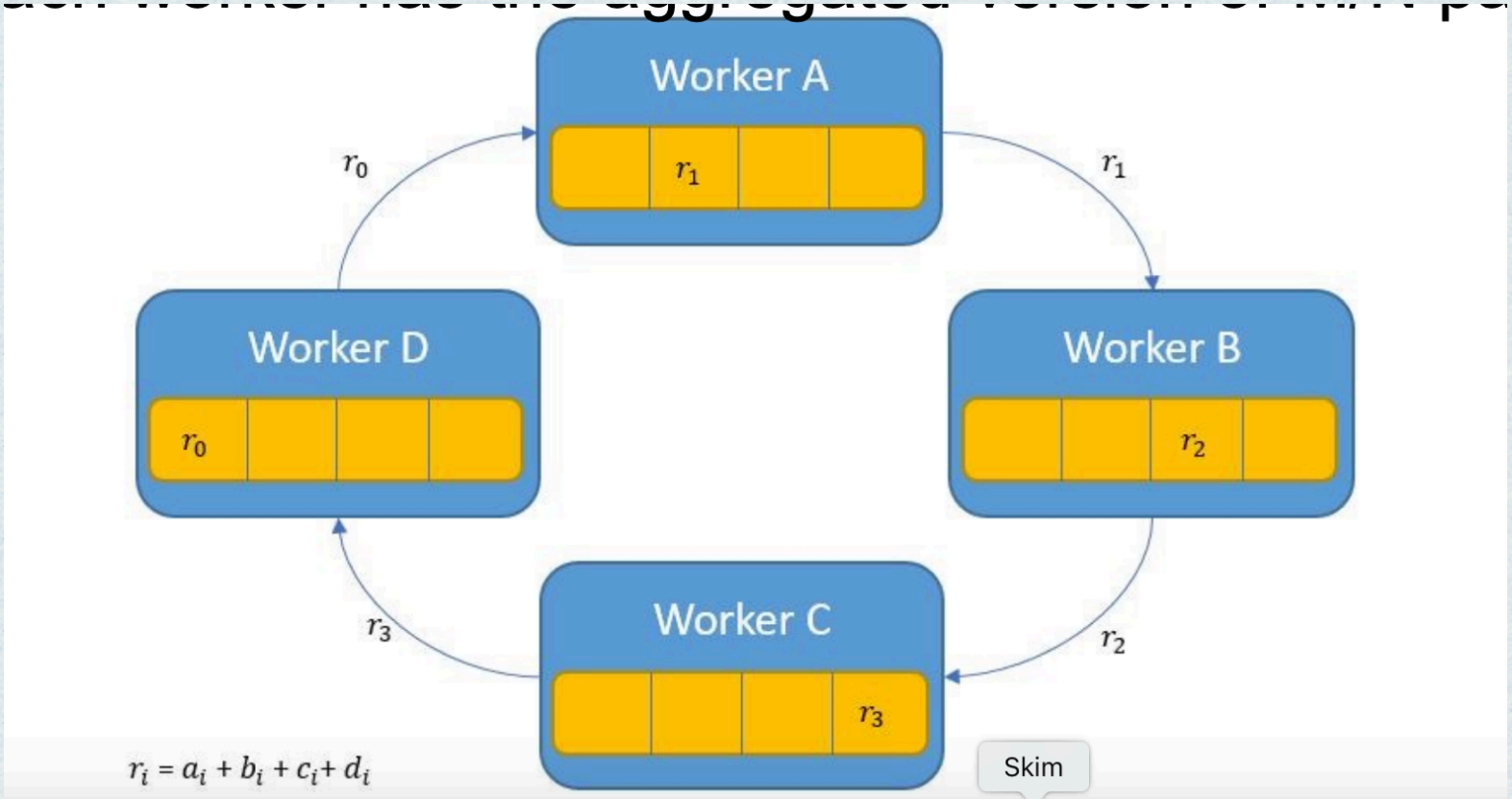
After It 1:



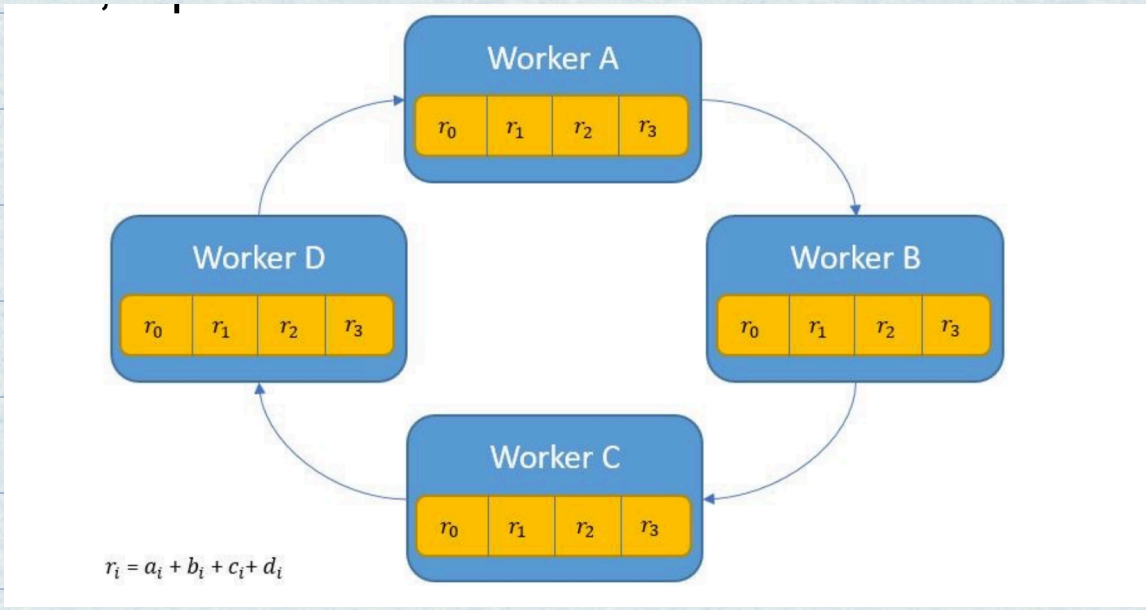
After It 2:



- After It 3 sum for each partition is on one of the workers



- Phase 2 of ring all reduce:
  - Broadcast → each worker sends one slice of aggregated params to next worker; repeat N-1 times



- what is total communication amount:
  - Discuss w/ neighbor

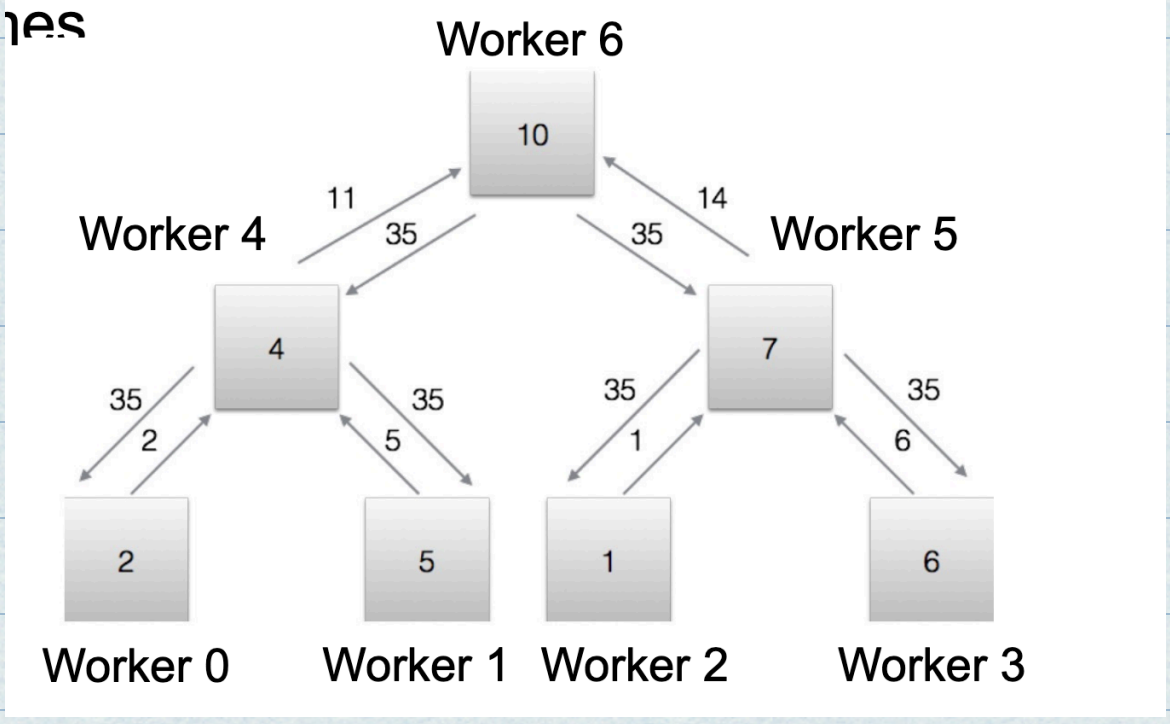
→ $O\left(2 \cdot (N-1) \cdot \frac{M}{N}, N\right)$

→ 2 times N-1 iterations

→ each it, a worker sends M/N data

✳ TREE Allreduce:

- construct tree of N workes
- step 1: aggregation:
  → each worker sends M params to its parent; repeat log (N) times
- step 2: Broadcast:
  → each worker sends M params to its children, repeat log (N) times



→ total comm. amount

$2 \cdot M \cdot N$ → each worker sends N cnt up of data, once once down

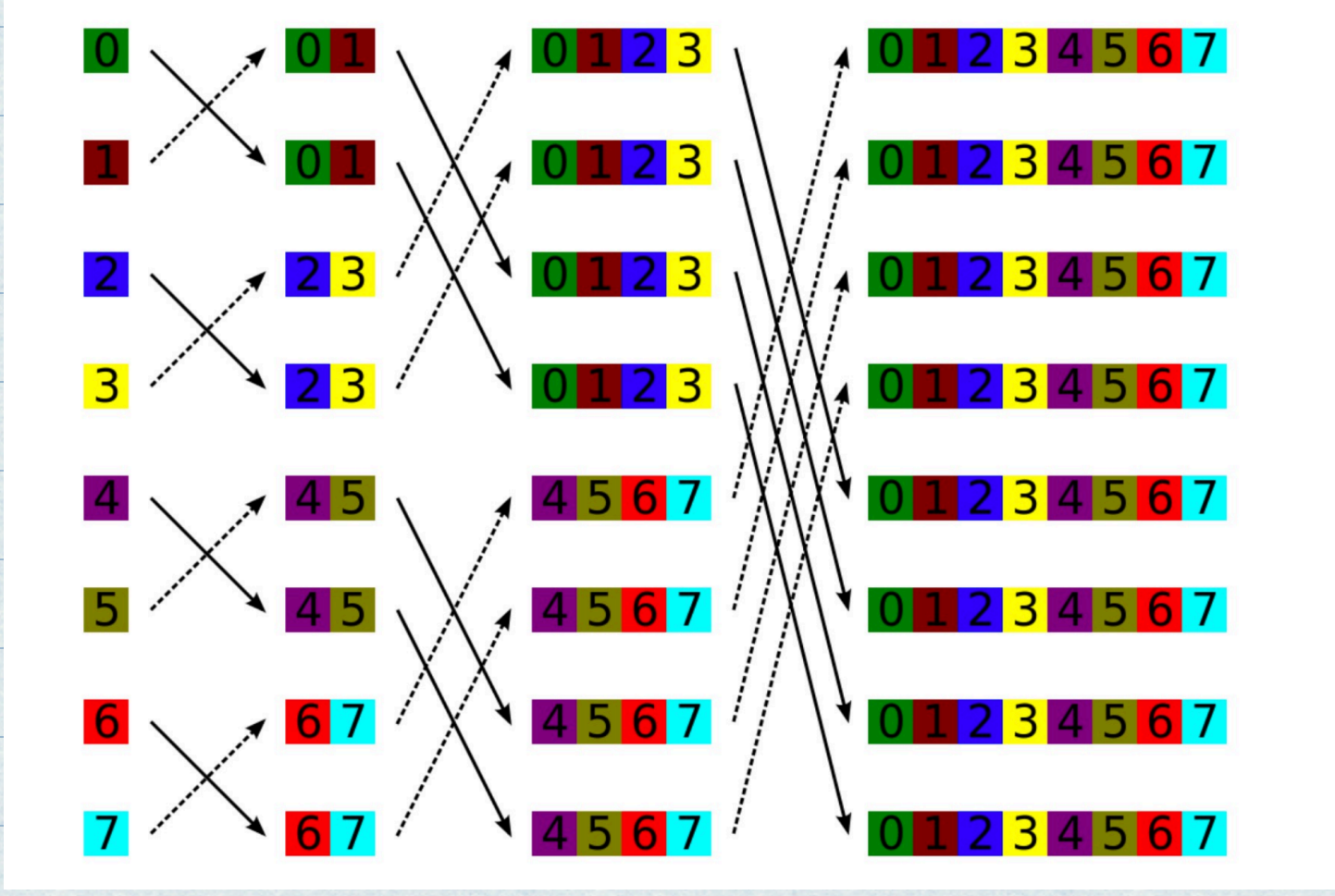
→ Agg: $M \cdot N$

→ Broadcast: $M \cdot N$

→ each worker requires 3 wires

* Butterfly All-reduce (I promise its the last one):

Construct butterfly network



→ Repeat log(N) times:
- each worker sends M param to its target node in butterfly network
- each worker aggregates gradients locally

→ comm. amount: $O(M \cdot N \cdot \log(N))$

→ each worker requires ~ log N wires

Method / Metric	Param server	Ring	Tree	Butterfly
# wires of bandwidth B	- each worker has 1 wires to PS - PS has N wires to each worker of B	- each worker has 2 wires of BW B	- each worker has 3 wires of BW B	- each worker has $\log N$ wires of BW B
# rounds/steps	2 (to PS and back)	$2 \cdot (N-1)$	$2 \cdot \log N$	$\log N$
amt of data each worker sends in a step	M (each worker sends	M/N	→ not all workers sending at all steps	M
Time per step	$\frac{M}{B}$ (for broadcast, assume PS can broadcast in parallel	$\frac{M}{N \cdot B}$	$\frac{M}{B}$ (can broadcast or aggs. in parallel)	$\frac{M}{B}$
Agg data sent: multiply row 3 by # of workers	$2 \cdot M \cdot N$	$2(N-1) \cdot \frac{M}{N} \cdot N$ $= 2(N-1) \cdot M$	$2 \cdot M \cdot N$ (up and down tree, each worker sends N data	$(MN \log N)$
Total Time =	$\frac{2 \cdot M}{B}$	$\frac{2(N-1) \cdot M}{N \cdot B}$	$\frac{2 \log N \cdot M}{B}$	$\frac{\log N (M)}{B}$

of "rounds" times time per round

Q: does tree or ring scale better? (both send ~ same amt of ags. data):

- ring appears to scale w/ $(N-1)/N$
- while tree w/ $\log N$
- inreality: B is really a function of message size (due to fixed overheads):

$B = f_B(\text{messagesize})$

- for large N,:

$$F_B\left(\frac{M}{N}\right)^{\text{ring}} \ll F_B(M)^{\text{tree}}$$

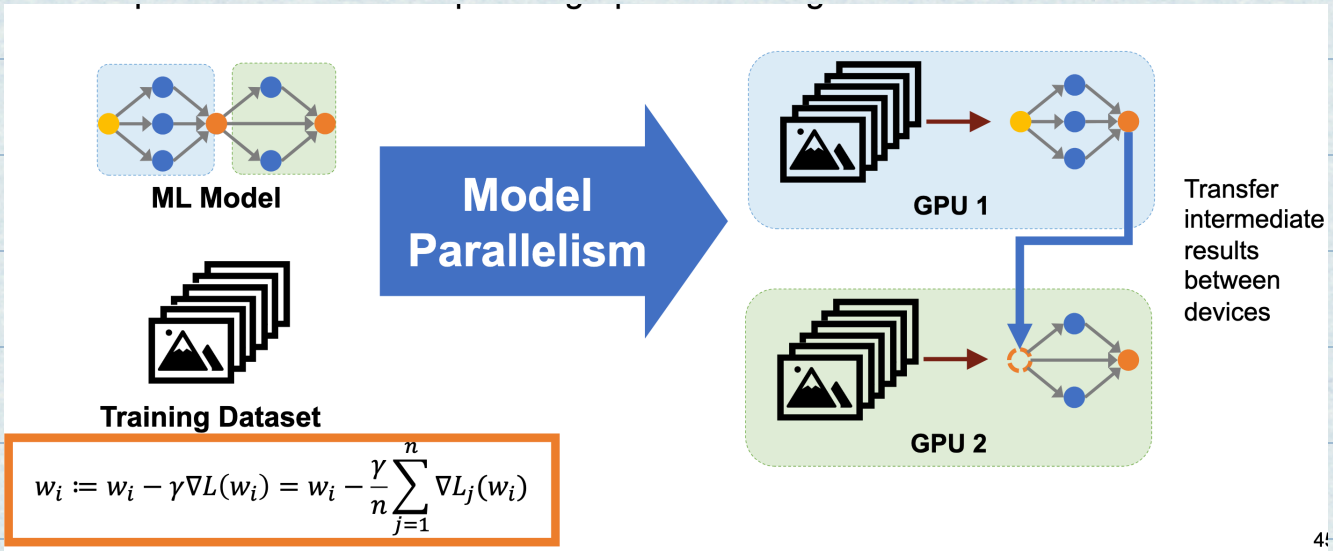
so time per-step: $\frac{M}{N \cdot f_B(M/N)} \gg \frac{M}{F_B(M)}$

- so for large N:

→ total time (time per step · # steps): $\frac{2(N-1)\cdot M}{N \cdot F_B(M/N)} \gg \frac{2 \cdot \log N \cdot M}{F_B(M)}$

↳ hence, tree is preferred

model Parallelism



- You will implement layer-by-layer model parallelism (partition the model by layers)