Parallelism

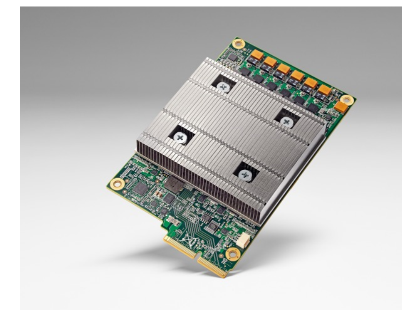
CS6787 Lecture 7 — Spring 2024

So far

- We’ve been talking about algorithms
- We’ve been talking about ways to optimize their parameters
- But we haven’t talked about the **underlying hardware**
  - How does the properties of the hardware affect our performance?
  - How should we implement our algorithms to best utilize our resources?

What does modern ML hardware look like?

- **Lots of different types**
  - CPUs
  - GPUs
  - FPGAs
  - Specialized accelerators

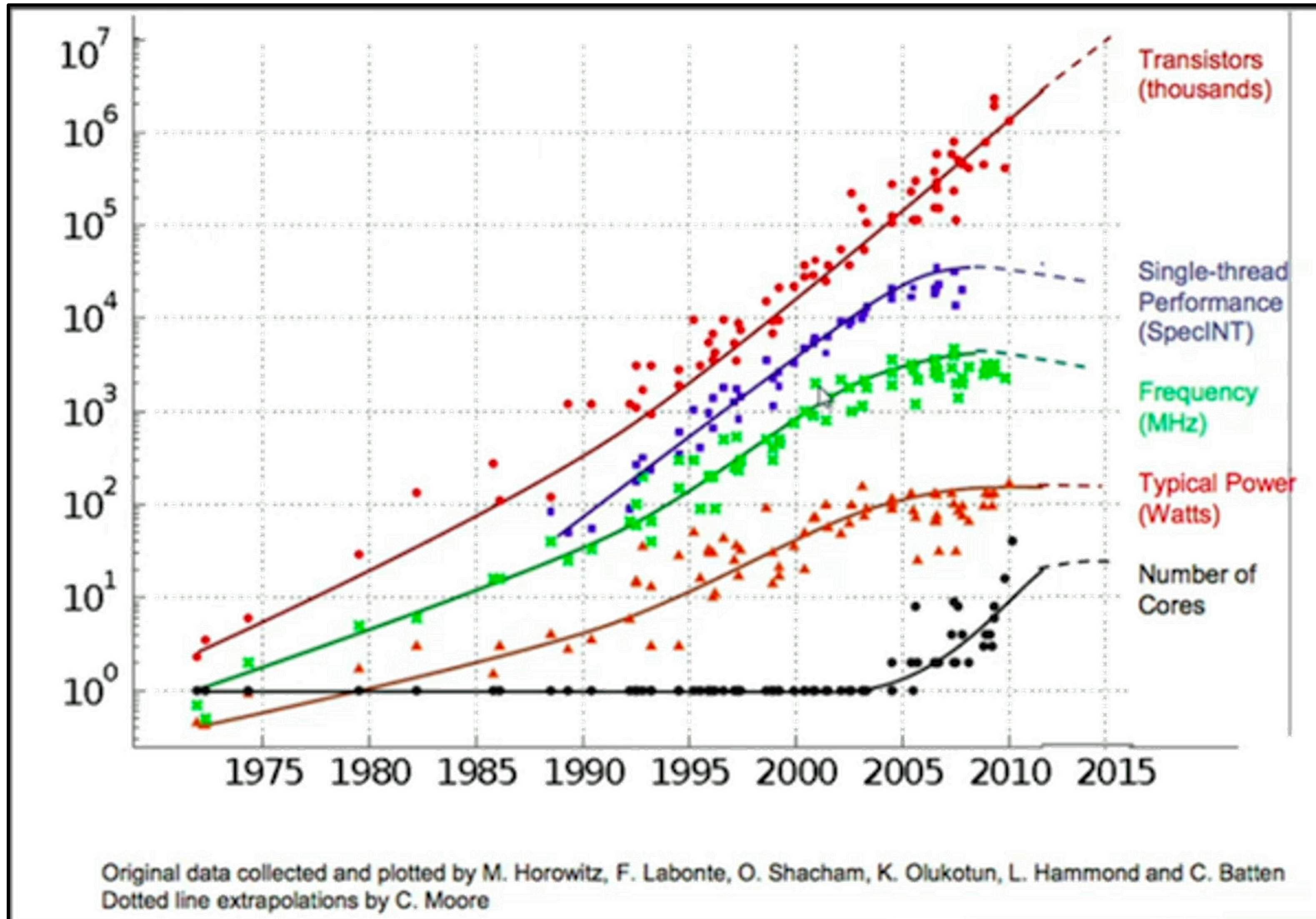


- Common thread: all of these architectures are **highly parallel**

Parallelism: A History

- The good old days: if I want my program to run faster, I can just **wait**
  - Moore's law —transistors on a chip doubles every 18 months
  - Dennard scaling — transistors shrink, power density stays constant
- This "free lunch" drove a **wave of innovation** in computing
  - Applications with bigger data were constantly becoming feasible
  - Drove a couple of AI boom-bust cycles
- But also drove a **lack of concern for systems efficiency**
  - Why work on making efficient systems when I can just wait instead?

Moore’s Law: A Graphic



The End of the Free Lunch

- In 2005, Herb Sutter declares — “**The Free Lunch Is Over**” and that there will be “A Fundamental Turn Toward Concurrency in Software”
  - He’s not the only one that was saying this.
- You can see this on the previous figure as trends start to flatten out.
- Why? **Power**
  - Dennard scaling started breaking down
  - Too much heat at high clock frequencies — chip will melt

The Solution: Parallelism

- I can re-write my program in parallel

- Moore's law is still in effect
  - Transistor density **still increasing exponentially**

- Use the transistors to **add parallel units** to the chip
  - Increases throughput, but not speed

The Effect of Parallelism

- **Pros:**
  - Can continue to get speedups from added transistors
  - Can even get speedups beyond a single chip or a single machine
- **Cons:**
  - Can’t just sit and wait for things to get faster
  - Need to work to get performance improvements
  - Need to develop new frameworks and methods to parallelize automatically

What benefits can we expect

- If we run in parallel on **N** copies of our compute unit, naively we would expect our program to run **N** times faster
- **Does this always happen in practice?**
- **No! Why?**

Amdahl's Law

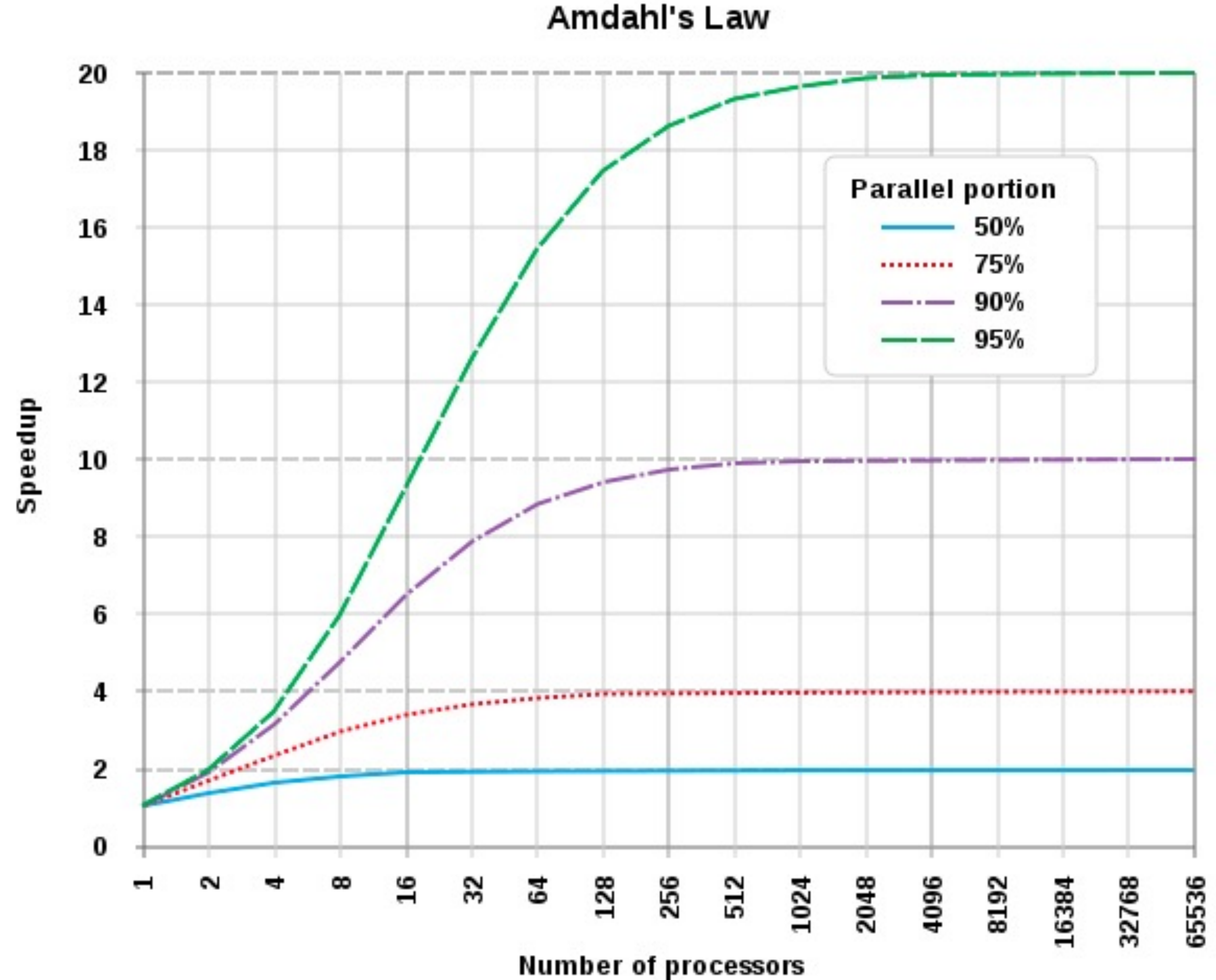
- Gives the **theoretical speedup** of a program when it's parallelized

$$S_{\text{latency}}(s) = \frac{1}{(1-p) + \frac{p}{s}}$$

  - $\mathbf{S_{latency}}$ is total speedup
  - **p** is the parallelizable portion of the algorithm
  - **s** is the number of parallel workers/amount of parallelism

From Wikipedia

Amdahl's Law (continued)



From Wikipedia

Consequences of Amdahl's Law

- **Diminishing marginal returns** as we increase the parallelism
- Can never actually achieve a linear or super-linear speedup as the amount of parallel workers increases
- **Is this always true in practice?**
- **No! Sometimes we do get super-linear speedup. When?**

What does modern parallel hardware look like?

- **CPUs**
  - Many parallel cores
  - Deep parallel cache hierarchies — taking up most of the area
  - Often many parallel CPU sockets in a machine

- **GPUs**
  - Can run way more numerical computations in parallel than a CPU
  - Loads of lightweight cores running together

- In general: can run many heterogeneous machines in parallel in a **cluster**

Sources of parallelism

From most fine-grained to most course-grained

On CPUs: Instruction-Level Parallelism

- **How many instructions in the instruction stream can be executed simultaneously?**

- For example:
  - C = A * B
  - Z = X * Y
  - S = C + Z

The first two instructions here can be executed in parallel

- Important for **pipelining**, and used fully in **superscalar processors**.

On CPUs: SIMD/Vector Parallelism

- **Single-Instruction Multiple-Data**
  - Perform the same operation on multiple data points in parallel
- Uses registers that store and process **vectors** of multiple data points
  - Latest standards use 512-bit registers, which can hold 16 floating point numbers
- A long series of instruction set extensions for this on CPUs
  - SSE, SSE2, SSE3, SSSE3, SSE4.1, SSE4.2, AVX, AVX2, AVX-512, …
- Critical for **dense linear algebra** operations common in ML

On CPUs: Multicore Parallelism

- Modern CPUs come with **multiple identical cores** on the same die
- Cores can work independently on **independent parallel tasks**
  - Unlike ILP and SIMD
- Cores communicate through **shared memory abstraction**
  - They can read and write the same memory space
  - This is done through a sophisticated cache hierarchy
- **Significant cost to synchronize** multiple CPUs working together

On CPUs: Hyperthreading

- Similar to multi-core

- Multiple threads running on the same core at the same time and sharing resources

- Slower than multi-core with the same thread count
  - Because resources are shared

- Usually don’t have to worry about it

On CPUs: Multi-socket parallelism

- Modern motherboards have multiple sockets for CPUs
- Cores on these CPUs still communicate through shared memory
- But latency/throughput to access memory that is “closer” to another CPU chip is **worse than accessing your own memory**
- This is called **non-uniform memory access** (NUMA)

DEMO

On GPUs: Stream Processing

- Given a stream of data, apply a series of operations to the data
  - Operations are called kernel functions
- This type of compute pattern is well-suited to GPU computation
  - Because compared with CPUs, GPUs have **much more of their area devoted to arithmetic** but much less devoted to memory and caches
- There's additional parallel structure within a GPU
  - For example, in CUDA threads running the same program are organized into **warps** and run at the same time

CUDA Parallelism Model

- "**Kernel**" a program element running on the GPU
- "**Warp**" a group of 32 threads that run together
  - Share an instruction stream — mostly
- "**Block**" a group of up to 1024 threads that can interact with each other via shared memory & synchronize
  - All threads in a block run on a single Streaming Multiprocessor (SM)
- "**Grid**" multiple blocks running on the GPU
  - Partitioned across all the SMs

GPU Tensor Cores

- Special hardware that **computes a small fixed-size matrix-matrix multiply**
  - e.g. 16x16, 8x32, 32x8

https://developer.nvidia.com/blog/programming-tensor-cores-cuda-9/

$$D = \begin{pmatrix} A_{0,0} & A_{0,1} & A_{0,2} & A_{0,3} \\ A_{1,0} & A_{1,1} & A_{1,2} & A_{1,3} \\ A_{2,0} & A_{2,1} & A_{2,2} & A_{2,3} \\ A_{3,0} & A_{3,1} & A_{3,2} & A_{3,3} \end{pmatrix} \begin{pmatrix} B_{0,0} & B_{0,1} & B_{0,2} & B_{0,3} \\ B_{1,0} & B_{1,1} & B_{1,2} & B_{1,3} \\ B_{2,0} & B_{2,1} & B_{2,2} & B_{2,3} \\ B_{3,0} & B_{3,1} & B_{3,2} & B_{3,3} \end{pmatrix} + \begin{pmatrix} C_{0,0} & C_{0,1} & C_{0,2} & C_{0,3} \\ C_{1,0} & C_{1,1} & C_{1,2} & C_{1,3} \\ C_{2,0} & C_{2,1} & C_{2,2} & C_{2,3} \\ C_{3,0} & C_{3,1} & C_{3,2} & C_{3,3} \end{pmatrix}$$

FP16 or FP32 FP16 FP16 FP16 or FP32

- A single tensor core matmul is a collaborative operation of all the threads in a warp
  - Or else runs asynchronously from the CUDA cores

On specialized accelerators and ASICs

- **Whatever you want!**
- The parallelism opportunities are limited only by the available transistors
- We will see **many new accelerators for ML** with different parallel structures and resources
  - Some will look like FPGAs: e.g. CGRAs
  - Some will just speed up one particular operation, such as matrix-matrix multiply

The Distributed Setting

- Many workers communicate **over a network**
  - Possibly heterogeneous workers including CPUs, GPUs, and ASICs
- Usually **no shared memory abstraction**
  - Workers communicate explicitly through passing messages
- Latency **much higher than all other types of parallelism**
  - Often need fundamentally different algorithms to handle this

How to use parallelism in machine learning

From most fine-grained to most course-grained

Recall

- Stochastic gradient descent

$$x_{t+1} = x_t - \alpha \nabla f(x_t; y_{\tilde{i}_t})$$

- Can write this as an algorithm:

- For **t = 1 to T**
  - Choose a training example at random
  - Compute the gradient and update the model
  - Repeat.

How to run SGD in parallel?

- There are several places where we can extract parallelism from SGD.

- We can use any or all of these places
  - Often we use different ones to correspond to the different sources of parallelism we have in the hardware we are using.

Parallelism within the Gradient Computation

- Try to compute the **gradient samples themselves** in parallel

$$x_{t+1} = x_t - \boxed{\alpha \nabla f(x_t; y_{\tilde{i}_t})}$$

- Problems:
  - We run this so many times, we will need to **synchronize a lot**
- Typical place to use: **instruction level parallelism, SIMD parallelism**
  - And distributed parallelism when using **model/pipeline parallelism**

Parallelism with Minibatching

- Try to parallelize across the **minibatch sum**

$$x_{t+1} = x_t - \frac{\alpha}{B} \sum_{b=1}^{B} \nabla f\left(x_t; y_{\tilde{i}_b}\right)$$

- Problems:
  - Still run this so many times, we will need to **synchronize a lot**
  - Can have a **tradeoff with statistical efficiency**, since too much minibatching can harm convergence

- Typical place to use: **all types of parallelism**

Parallelism across iterations

- Try to compute **multiple iterations** of SGD in parallel
  - Parallelize the outer loop — usually a good idea

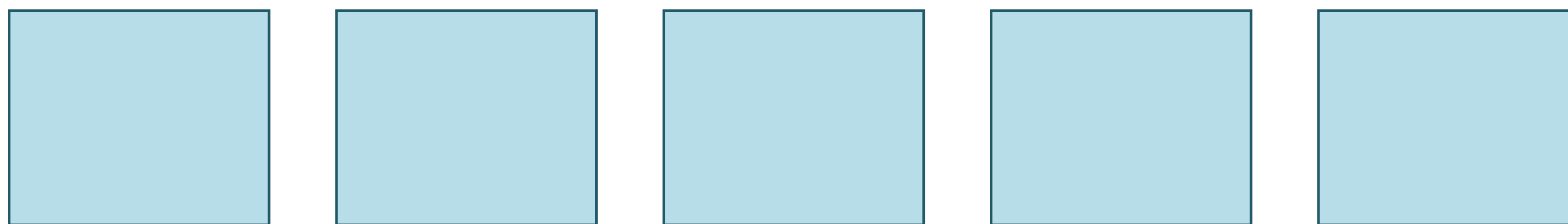
$$x_{t+1} = x_t - \alpha \nabla f(x_t; y_{\tilde{i}_t})$$
$$x_{t+1} = x_t - \alpha \nabla f(x_t; y_{\tilde{i}_t})$$
$$x_{t+1} = x_t - \alpha \nabla f(x_t; y_{\tilde{i}_t})$$
$$x_{t+1} = x_t - \alpha \nabla f(x_t; y_{\tilde{i}_t})$$

- Problems:
  - Naively, **the outer loop is sequential**, so we can't do this without fine-grained locking and frequent synchronization

- Typical place to use: **multi-core/multi-socket/cluster parallelism**

Parallelism for hyperparameter optimization

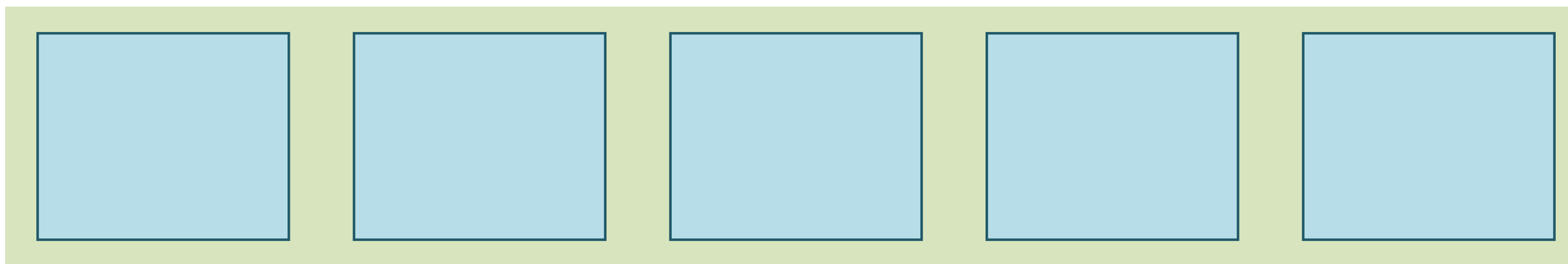
- Just run **multiple copies of the whole algorithm** independently, and use them to do **hyperparameter optimization**



- Problems:
  - Can't do this if you don't want to do hyperparameter optimization
  - **Isn't actually useful once you've already set your parameters**
- Typical place to use: **distributed computation**

Parallelism for ensembling

- Just like before, run **multiple copies of the whole algorithm** independently, and use them to produce **an ensemble classifier**



- Problems:
  - Can’t do this if you don’t want to train an ensemble classifier
  - Now the difficulty for learning

- Typical place to use: **distributed computation**

What about our other methods?

- We can **speed up all our methods with parallel computing**
  - Minibatching — has a close connection with parallelism
  - SVRG
  - Momentum

- And any **SGD-like algorithm** lets us use the same ways to extract parallelism from it
  - Things like gradient descent, stochastic coordinate descent, stochastic gradient Langevin dynamics, and many others.

Asynchronous Parallelism

Limits on parallel performance

- Synchronization
  - Have to synchronize to keep the workers aware of each other's updates to the model — otherwise can introduce errors

- **Synchronization can be very expensive**
  - Have to stop all the workers and wait for the slowest one
  - Have to wait for several round-trip times through a high-latency channel

- **Is there something we can do about this?**

Idea: Just Don’t Synchronize

- Not synchronizing adds **errors due to race conditions**
- But our methods were already noisy — **maybe these errors are fine**
- If we don’t synchronize, get **almost perfect parallel speedup**

Fast Parallel SGD: HOGWILD!

Multiple parallel workers

- Pick a training example y_{i_t} uniformly at random
- Update the model x_t using a gradient estimate

$$x_{t+1} = x_t - \alpha \nabla f(x_t; y_{i_t})$$

- Iterate

- Pick a training example y_{i_t} uniformly at random
- Update the model x_t using a gradient estimate

$$x_{t+1} = x_t - \alpha \nabla f(x_t; y_{i_t})$$

- Iterate

- Pick a training example y_{i_t} uniformly at random
- Update the model x_t using a gradient estimate

$$x_{t+1} = x_t - \alpha \nabla f(x_t; y_{i_t})$$

- Iterate

- Pick a training example y_{i_t} uniformly at random
- Update the model x_t using a gradient estimate

$$x_{t+1} = x_t - \alpha \nabla f(x_t; y_{i_t})$$

- Iterate

- Pick a training example y_{i_t} uniformly at random
- Update the model x_t using a gradient estimate

$$x_{t+1} = x_t - \alpha \nabla f(x_t; y_{i_t})$$

- Iterate

- Pick a training example y_{i_t} uniformly at random
- Update the model x_t using a gradient estimate

$$x_{t+1} = x_t - \alpha \nabla f(x_t; y_{i_t})$$

- Iterate

- Pick a training example y_{i_t} uniformly at random
- Update the model x_t using a gradient estimate

$$x_{t+1} = x_t - \alpha \nabla f(x_t; y_{i_t})$$

- Iterate

- Pick a training example y_{i_t} uniformly at random
- Update the model x_t using a gradient estimate

$$x_{t+1} = x_t - \alpha \nabla f(x_t; y_{i_t})$$

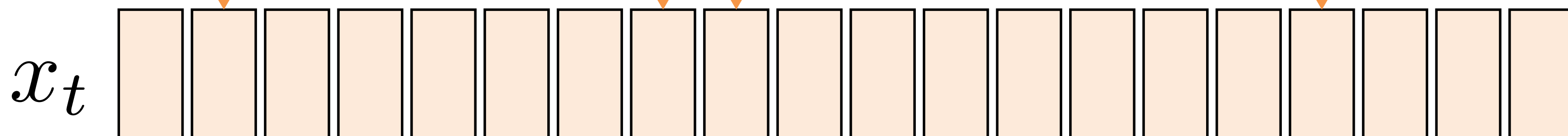
- Iterate

- Pick a training example y_{i_t} uniformly at random
- Update the model x_t using a gradient estimate

$$x_{t+1} = x_t - \alpha \nabla f(x_t; y_{i_t})$$

- Iterate

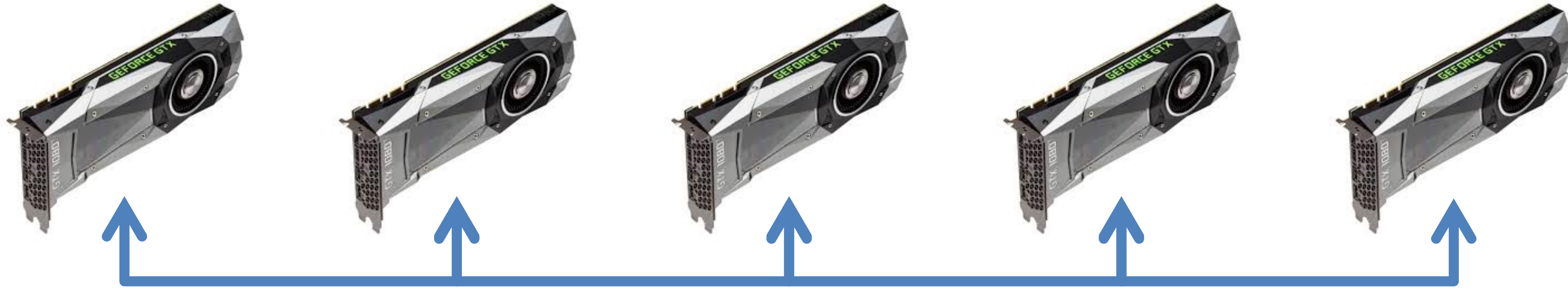
Asynchronous parallel updates (no locks) to a single shared model



Distributed Learning

CS6787 Lecture 7/8 — Fall 2024

Main idea: **use multiple machines to do learning**.



Why distribute?

- Train **more quickly**
- Train **models too large** to fit on one machine
- Train when the **data are inherently distributed**

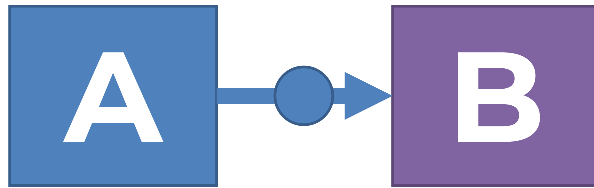
Distributed computing basics

- Distributed parallel computing **involves two or more machines collaborating on a single task by communicating over a network**.
    - Distributed computing requires explicit (i.e. written in software) communication among the workers.
    - **No shared memory abstraction!** (Unlike parallelism on 1 machine)

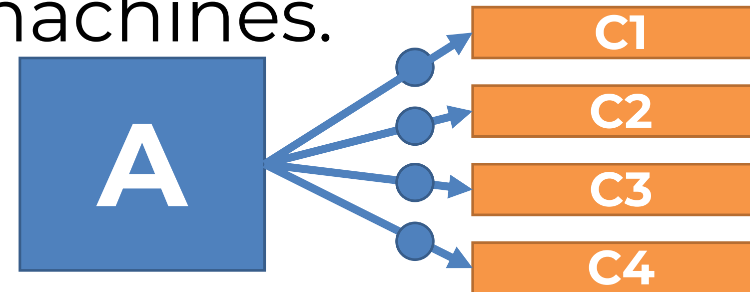
- There are a **few basic patterns of communication** that are used by distributed programs.

Basic patterns of distributed communication

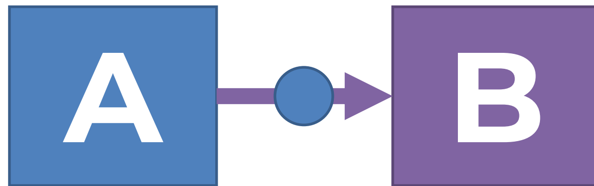
Push: Machine A sends some data to machine B.



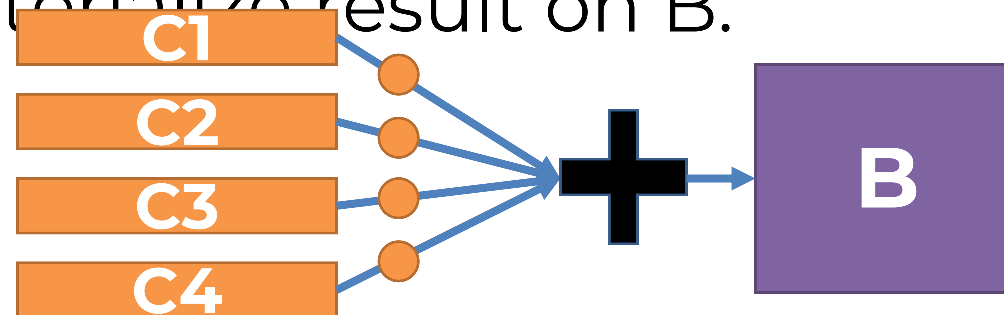
Broadcast: Machine A sends data to many machines.



Pull: Machine B requests some data from machine A.

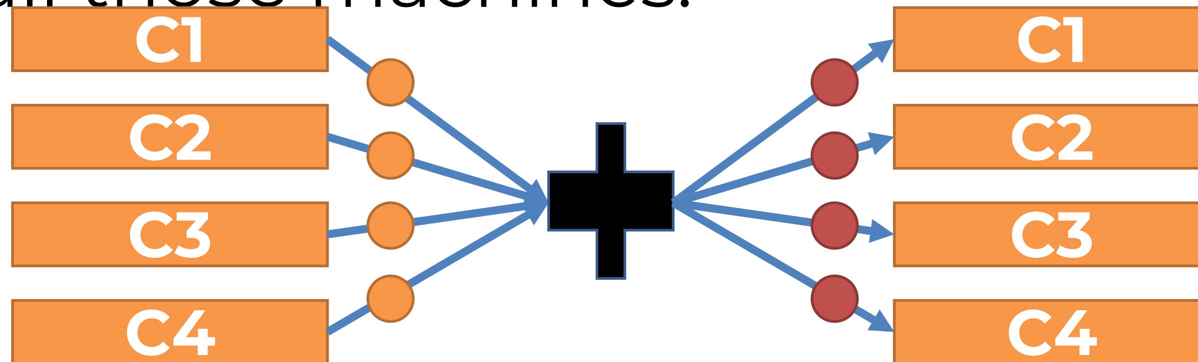


Reduce: Compute some reduction of data on multiple machines and materialize result on B.

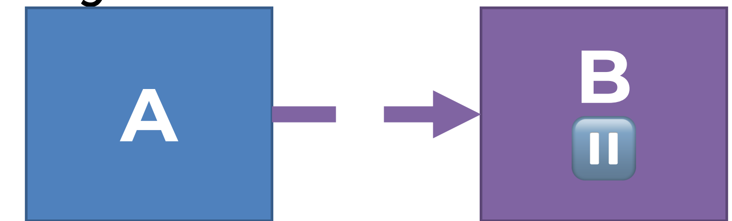


Basic patterns of distributed communication (cont'd)

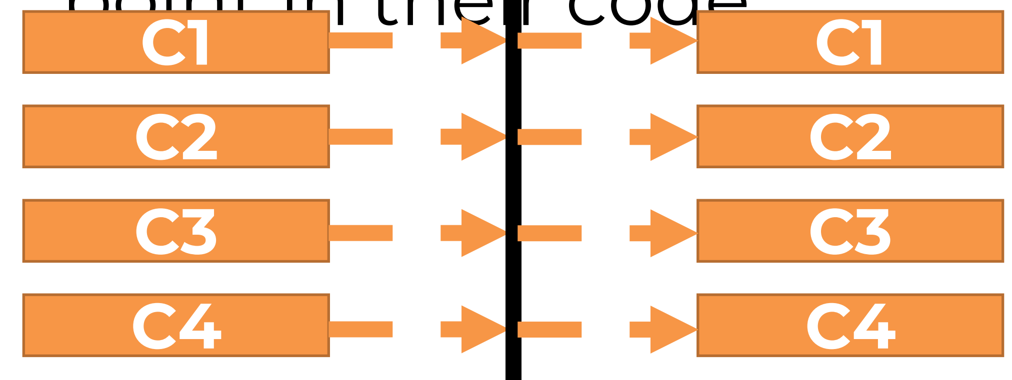
All-reduce: Compute some reduction of data on multiple machines and materialize result on all those machines.



Wait: Pause until another machine says to continue.



Barrier: Wait for all workers to reach some point in their code.



All these operations can be synchronous or asynchronous.

Overlapping computation and communication

- Communicating over the network can have high latency
  - we want to hide this latency

- An important principle of distributed computing is **overlapping computation and communication**

- For the best performance, we want our workers to **still be doing useful work while communication is going on**
  - rather than having to stop and wait for the communication to finish
  - sometimes called a **stall**
  - **asynchronous communication** can help a lot here

Running SGD with All-reduce

- All-reduce gives us a simple way of running learning algorithms such as SGD in a distributed fashion.
- Simply put, the idea is to just **parallelize the minibatch.** We start with an identical copy of the parameter on each worker.
- Recall that SGD update step looks like:

$$w_{t+1} = w_t - \alpha_t \cdot \frac{1}{B} \sum_{b=1}^{B} \nabla f_{i_{b,t}}(w_t),$$

Running SGD with All-reduce (continued)

- If there are M worker machines such that $B = M \cdot B'$, then

$$w_{t+1} = w_t - \alpha_t \cdot \frac{1}{M} \sum_{m=1}^{M} \frac{1}{B'} \sum_{b=1}^{B'} \nabla f_{i_{m,b,t}}(w_t).$$

- Now, we assign the computation of the sum when m = 1 to worker 1, the computation of the sum when m = 2 to worker 2, et cetera.

- After all the gradients are computed, we can perform the outer sum with an **all-reduce operation**.

Running SGD with All-reduce (continued)

- After this all-reduce, the whole sum (which is essentially the minibatch gradient) will be present on all the machines
  - so each machine can now update its copy of the parameters
- Since sum is same on all machines, the parameters will update in lockstep
- **Statistically equivalent to sequential SGD!**

Algorithm 1 Distributed SGD with All-Reduce

input: loss function examples $f_1, f_2, \ldots$, number of machines M, per-machine minibatch size B'
input: learning rate schedule α_t, initial parameters w_0, number of iterations T
for $m = 1$ **to** M **run in parallel on machine** m
  load w_0 from algorithm inputs
  for $t = 1$ **to** T **do**
    select a minibatch $i_{m,1,t}, i_{m,2,t}, \ldots, i_{m,B',t}$ of size B'
    compute $g_{m,t} \leftarrow \frac{1}{B'} \sum_{b=1}^{B'} \nabla f_{i_{m,b,t}}(w_{t-1})$
    all-reduce across all workers to compute $G_t = \sum_{m=1}^{M} g_{m,t}$
    update model $w_t \leftarrow w_{t-1} - \frac{\alpha_t}{M} \cdot G_t$
  end for
end parallel for
return w_T (from any machine)

Same approach can be used for momentum, Adam, etc.

Benefits of distributed SGD with All-reduce

- The algorithm is easy to reason about, since it's **statistically equivalent to minibatch SGD**.
  - And we can use the same hyperparameters for the most part.

- The algorithm is easy to implement
  - since all the worker machines have the same role and it runs on top of standard distributed computing primitives.

Drawbacks of distributed SGD with all-reduce

- We're **not overlapping computation and communication**.
  - While the communication for the all-reduce is happening, the workers are idle.
- The **effective minibatch size is growing with the number of machines**
  - If we *don't* want to run with a large minibatch size for statistical reasons, this can prevent us from scaling to large numbers of machines using this method.
- Potentially requires **lots of network bandwidth** to communicate to all workers.

Where do we get the training examples from?

- There are two general options for distributed learning.

- **Training data servers**
  - Have one or more non-worker servers dedicated to storing the training examples (e.g. a distributed in-memory filesystem)
  - The worker machines load training examples from those servers.

- **Partitioned dataset**
  - Partition the training examples among the workers themselves and store them locally in memory on the workers.

The Parameter Server Model

The Basic Idea

- Recall from the early lectures in this course that a lot of our theory talked about the convergence of optimization algorithms.
  - This convergence was measured by some function over the parameters at time t (e.g. the objective function or the norm of its gradient) that is decreasing with t, which shows that the algorithm is making progress.
- For this to even make sense, though, we need to be able to talk about the value of the parameters at time t as the algorithm runs.
  - E.g. in SGD, we had $w_{t+1} = w_t - \alpha_t \nabla f_{i_t}(w_t)$

Parameter Server Basics Continued

- For a program runni
parameters at time t
memory hierarchy (b

- But in a distributed s ... d
communication mu
  - Each machine will us
parameters live at ar
updates less recently ...
something more complicated ... reduce.

- This raises the question: **when reasoning about a distributed algorithm, what we should consider to be the value of the parameters a given time**?

For SGD with all-reduce, we can answer this question easily, since the value of the parameters is the same on all workers (it's guaranteed to be the same by the all-reduce operation). We just appoint this identical shared value to be the value of the parameters at any given time.

The Parameter Server Model

- The parameter server model answers this question differently by appointing a single machine, the **parameter server**, the explicit responsibility of maintaining the current value of the parameters.
  - The most up-to-date gold-standard parameters are the ones stored in memory on the parameter server.

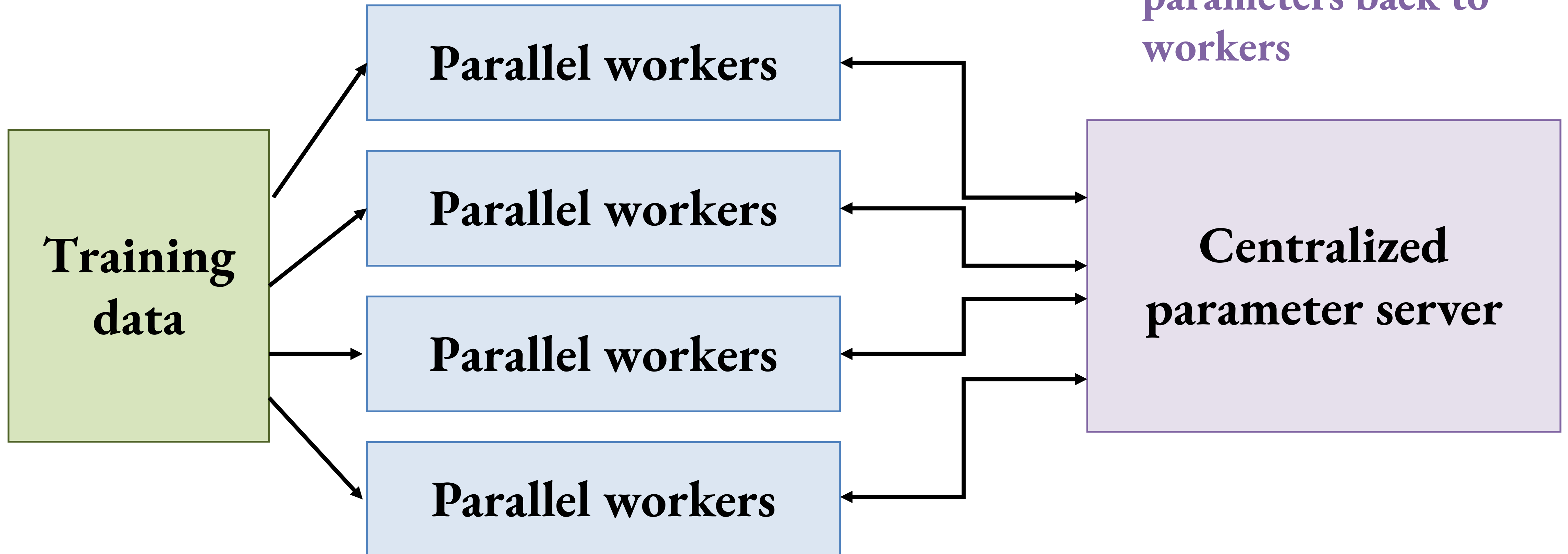
- The parameter server updates its parameters by using gradients that are computed by the other machines, known as **workers**, and pushed to the parameter server.

- Periodically, the parameter server **broadcasts its updated parameters** to all the other worker machines, so that they can use the updated parameters to compute gradients.

Parameter server model: visually

- A common model for distributed ML

- workers send gradients to parameter server
- parameter server sends parameters back to workers



Learning with the parameter server

- Two options when learning with a parameter server

- **Synchronous distributed training**
  - Similar to all-reduce, but with gradients summed on a central parameter server
  - Still **equivalent to sequential minibatch SGD**

- **Asynchronous distributed training**
  - Compute and send gradients and add them to the model as soon as possible
  - Broadcast updates whenever they are available

Parameter server summary

- The parameter server **holds the central copy of the weights**
- Each worker **computes gradients** on minibatches the data
  - Then sends those gradients back to the parameter server
- Periodically, the worker pulls an updated copy of the weights from the parameter server.
- All this can be done **asynchronously**.

Multiple parameter servers

- If the parameters are too numerous for a single parameter server to handle, we can use **multiple parameter server machines**.

- We partition the parameters among the multiple parameter servers
  - Each server is only responsible for maintaining the parameters in its partition.
  - When a worker wants to send a gradient, it will partition that gradient vector and send each chunk to the corresponding parameter server; later, it will receive the corresponding chunk of the updated model from that parameter server machine.

- This lets us **scale up to very large models!**

Other Ways To Distribute

The methods we discussed so far distributed across the minibatch (for all-reduce SGD) and across iterations of SGD (for asynchronous parameter-server SGD).

But there are other ways to distribute that are used in practice too.

Decentralized learning

"Can Decentralized Algorithms Outperform Centralized Algorithms? A Case Study for Decentralized Parallel Stochastic Gradient Descent." NeurIPS 2017

- Idea: learn **without any central coordination**
  - No parameter server; each worker has its own copy of the model
- Workers update by doing the following:
  - Run an SGD update step using an example stored on that worker,
  - Average the worker's current model with the models of some other workers, usually its neighbors in some sparse graph
    - This limits total communication
- This is sometimes called a **gossip algorithm**

Optimal Complexity in Decentralized Training. Yucheng L
Christopher De Sa. In *ICML: the Thirty-eighth International Conferen*
on Machine Learning, July 2021.

Decentralization cont'd

- Roughly three senses in which an algorithm can be **"decentralized"**
- Application layer: **Decentralized data**
  - Distributions of data different on different workers
- Protocol layer: **Gossip protocol**
- Network layer: Communication through **sparsely connected graph** topology

e.g. "Local SGD Converges Fast and Communicates Little." ICLR 2019

Local SGD

- Many parallel workers update their own copy of the model by running SGD steps using their own local data
- Periodically the workers all average by taking an **all-reduce**
  - Like all-reduce SGD, but the all-reduce happens less frequently than at every SGD iteration
- Can **generalize better** than large-batch SGD
  - "Don't use large mini-batches, use local SGD." ICLR 2020

So far: Data Parallelism

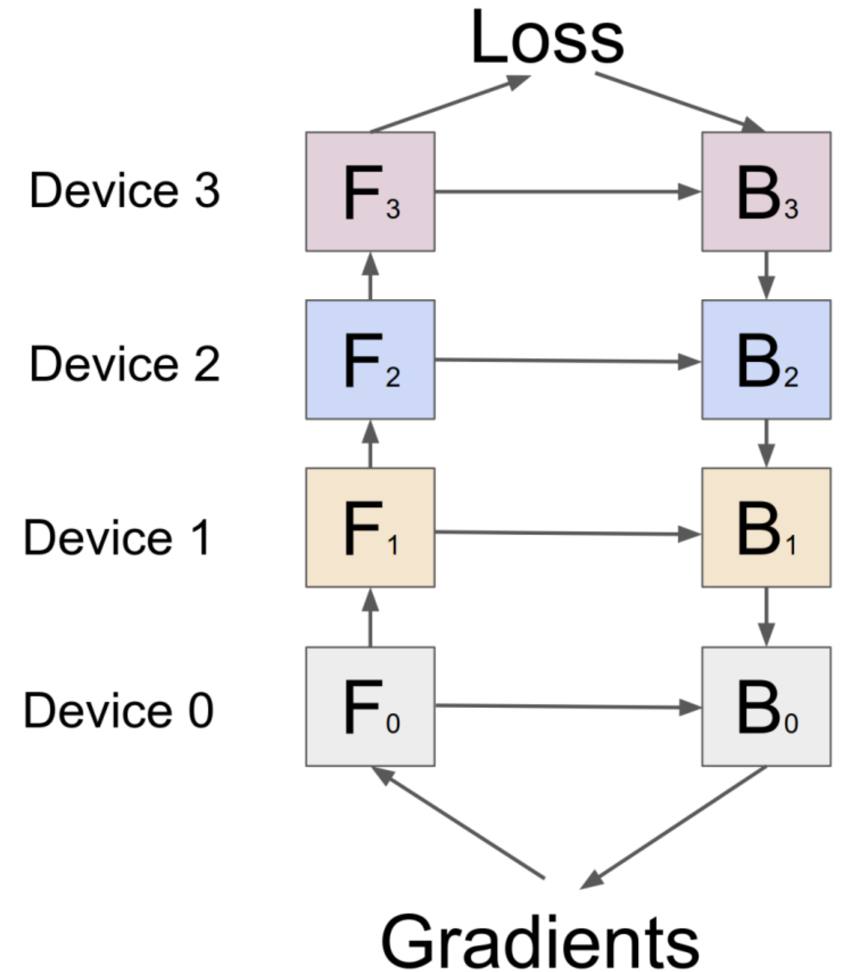
- The methods we've discussed are parallelizing over examples
  - Each worker is running the same computation to compute gradients, just on different examples.
- This is an instance of **data parallelism**
- But **data parallelism is not the only option...**

Model Parallelism

- Main idea: **partition the layers** of a neural network among different worker machines.
- This makes each worker responsible for a subset of the parameters.
- Forward and backward signals running through the neural network during backpropagation now also run across the computer network between the different parallel machines.
  - Particularly useful if the parameters won't fit in memory on a single machine.
  - This is very important when we move to specialized machine learning accelerator hardware, where we're running on chips that typically have limited memory and communication bandwidth.

Pipeline Parallelism

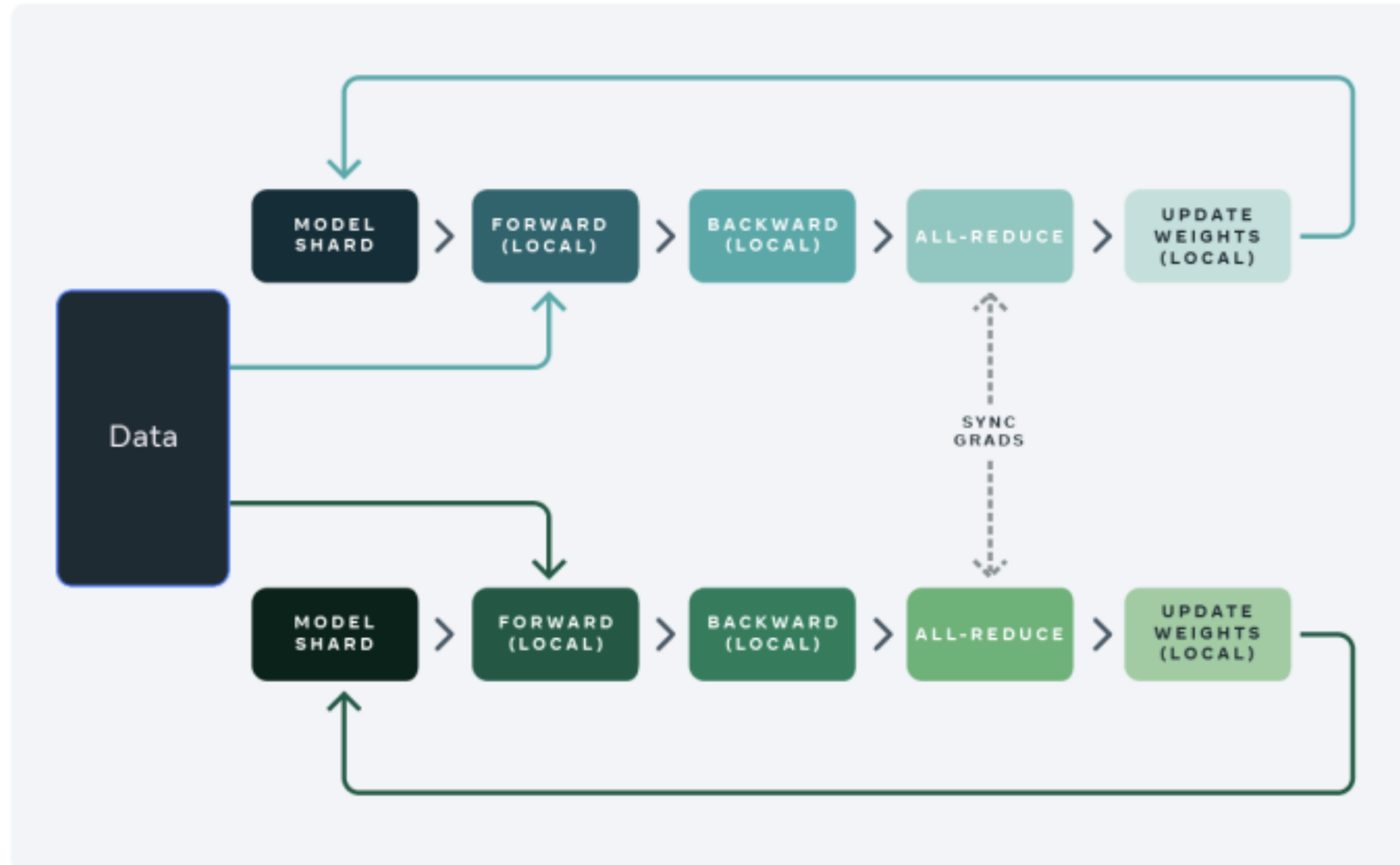
- Distribute a DNN over multiple workers by **assigning each layer to its own worker**.
  - Each worker manages and updates the parameters for its own layer.
  - Use **microbatching** to avoid stalls
- Advantage: **workers no longer need to store the entire model**
  - Can often keep parameters in memory



From “GPipe: Easy Scaling with Micro-Batch Pipeline Parallelism”

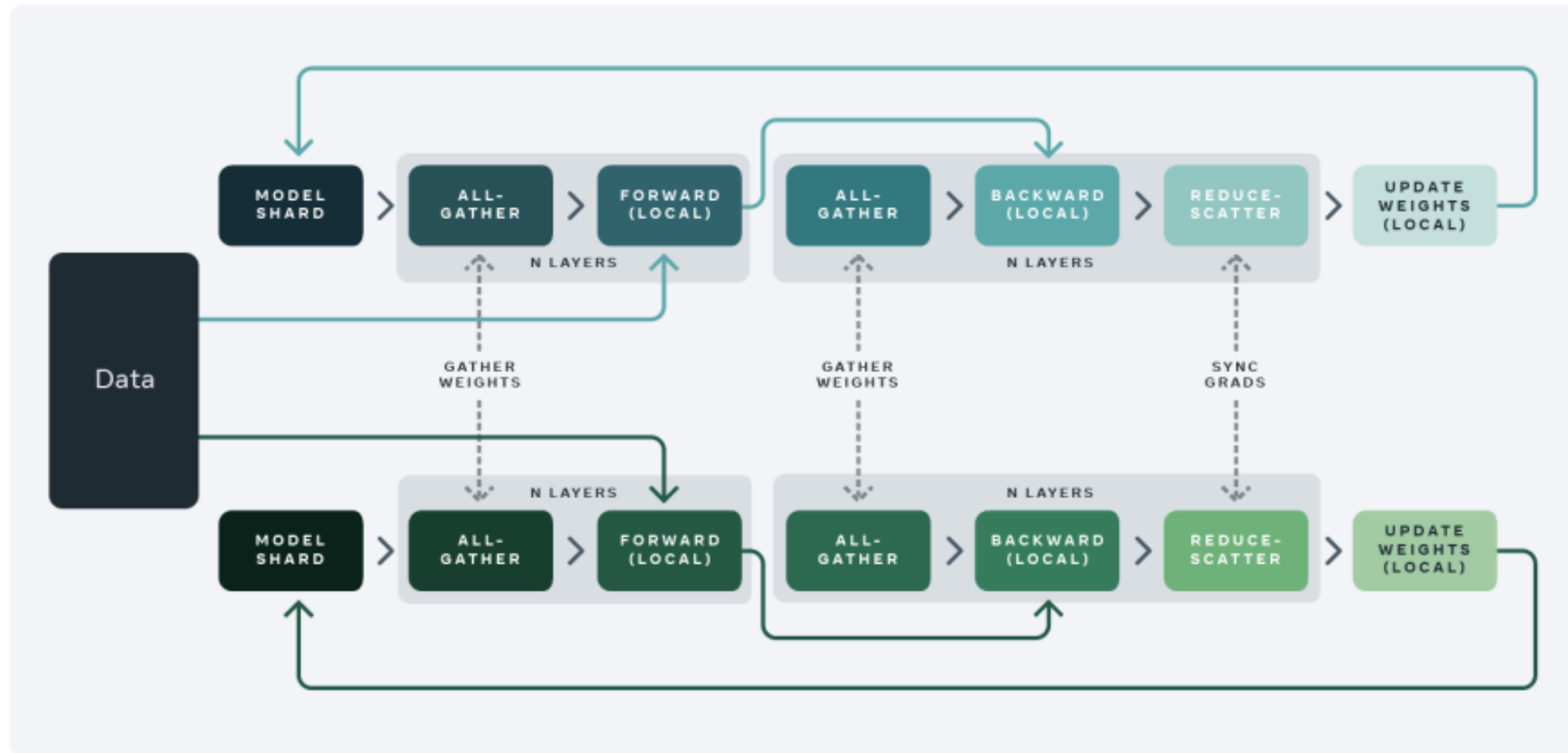
Fully Sharded Data Parallel

- Distribute a DNN over workers by **assigning a portion of each layer to each worker**.
  - Each worker manages and updates the parameters for its own “shard”
  - Use **all-gather** to manifest whole weight matrix on all workers when it is time to run forward/backward
  - Still **parallelize over data**!
- Advantage: **workers no longer need to store the entire model**



https://engineering.fb.com/2021/07/15/open-source/fsdp/

Fully sharded data parallel training



https://engineering.fb.com/2021/07/15/open-source/fsdp/

Federated learning

- Sometimes, **your data is inherently distributed**
  - For example, data gathered on people's mobile phones
  - For example, data measured by internet-of-things devices
- Rather than centralizing the data, may want to learn on the distributed devices themselves
  - E.g. to preserve the privacy of users
- This is called **federated learning**
  - **Lots of interest from industry right now**

Distributed computing for hyperparameter optimization

- This is something we've already talked about.

- Many commonly used hyperparameter optimization algorithms, such as **grid search and random search**, are very simple to distribute.
  - They can easily be run on many parallel workers to get results faster.