

# Parallelism

CS6787 Lecture 9 — Fall 2019

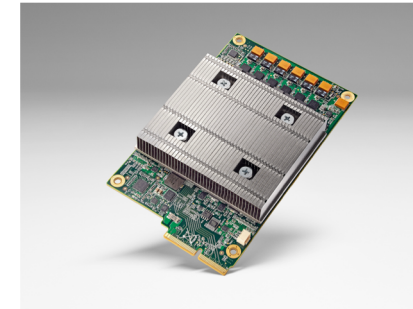
# So far

- 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 do 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

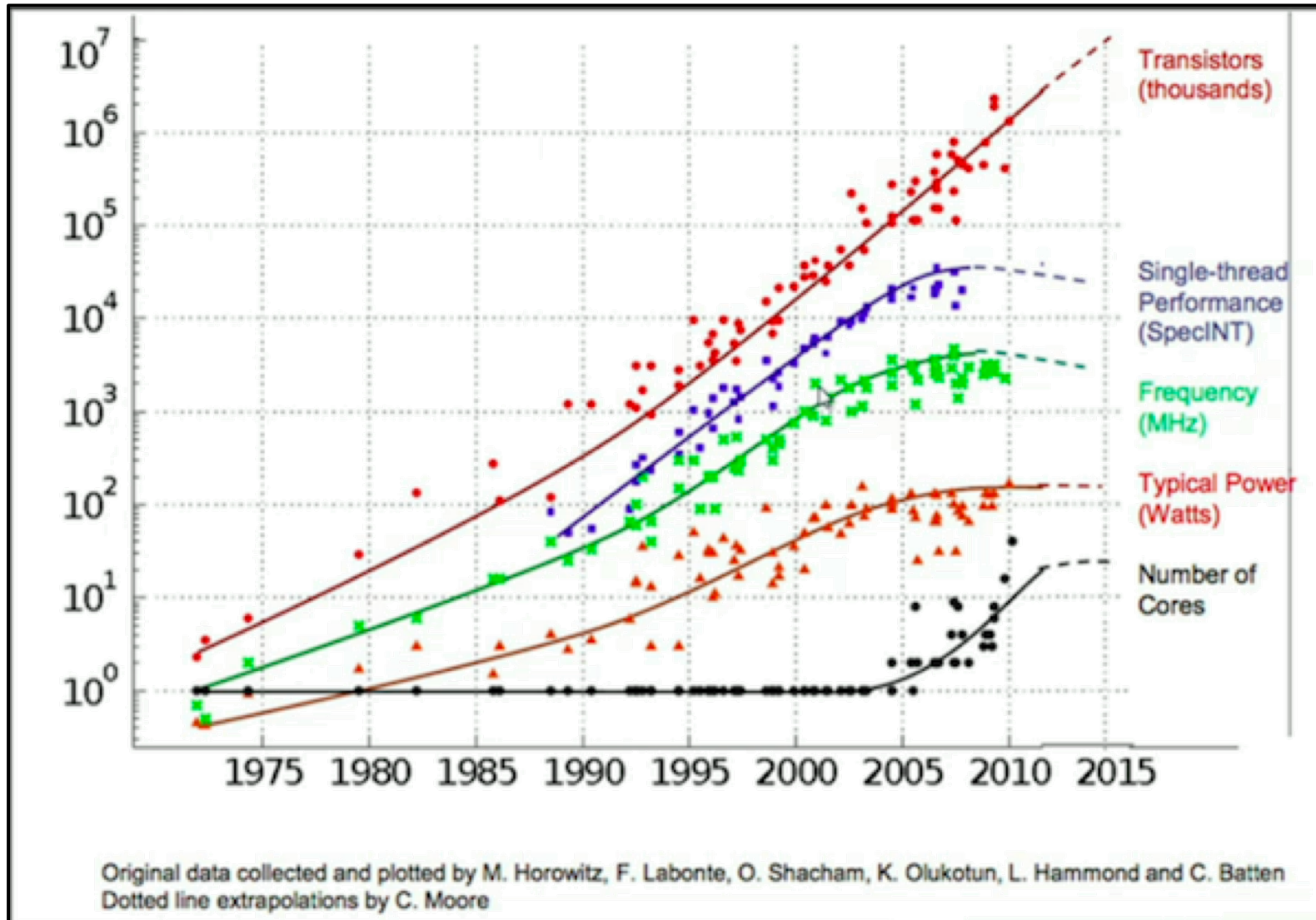


- Common thread: all of these architectures are **highly parallel**

# Parallelism: A History

- The good old days: if I want my program to run faster, I can just **wait**
  - Moore's law — number of transistors on a chip doubles every 18 months
  - Dennard scaling — as transistors get smaller, power density stays constant
- This “free lunch” drove a **wave of innovation** in computing
  - Because new 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 — no longer fixed power/area
  - Too much heat to dissipate 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 more 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?**

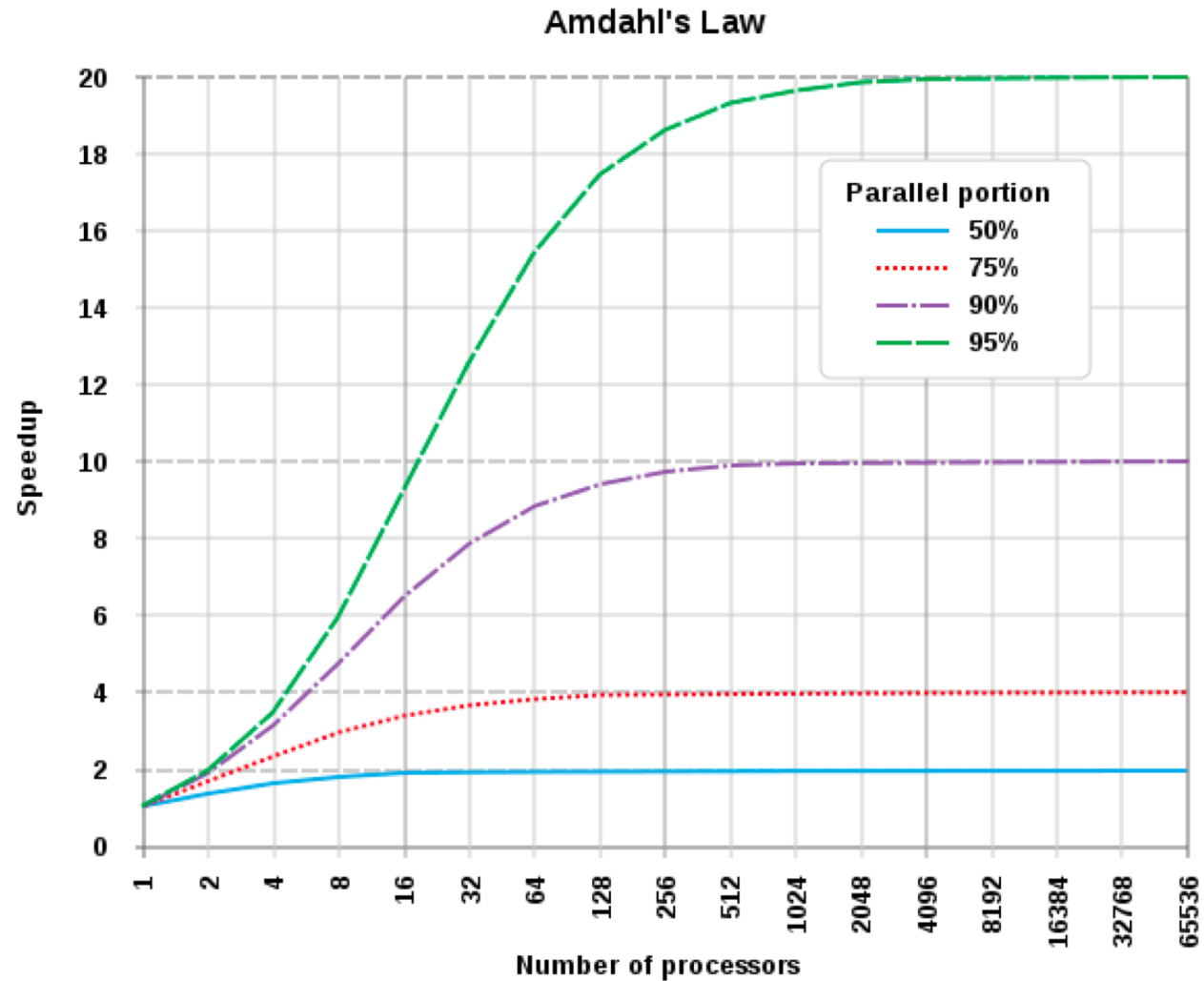
# Amdahl's Law

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

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

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

# Amdahl's Law (continued)



# 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: 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)

# 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

# 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

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

- Problems:
  - There usually is **not much work here**, so not much work to split up
  - We run this so many times, we will need to **synchronize a lot**
- Typical place to use: **instruction level parallelism, SIMD parallelism**

# Parallelism with Minibatching

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

$$x_{t+1} = x_t - \frac{\alpha}{B} \sum_{b=1}^B \nabla f(x_t; y_{i_b})$$

- 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

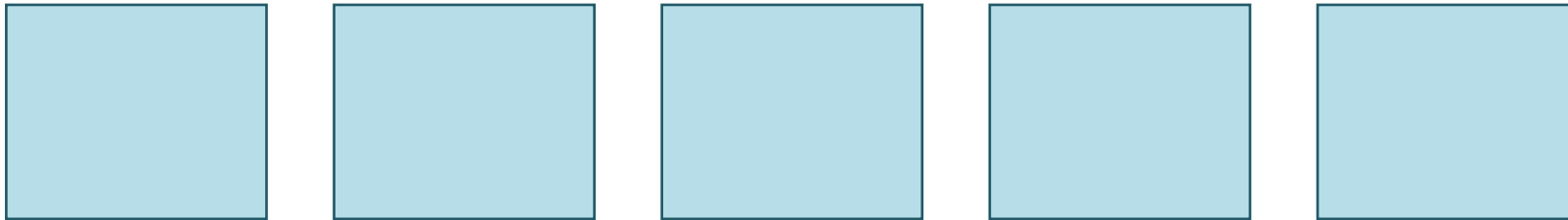
- Try to compute **multiple iterations** of SGD in parallel
  - Parallelize the outer loop — usually a good idea

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

- 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 particularly 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?**

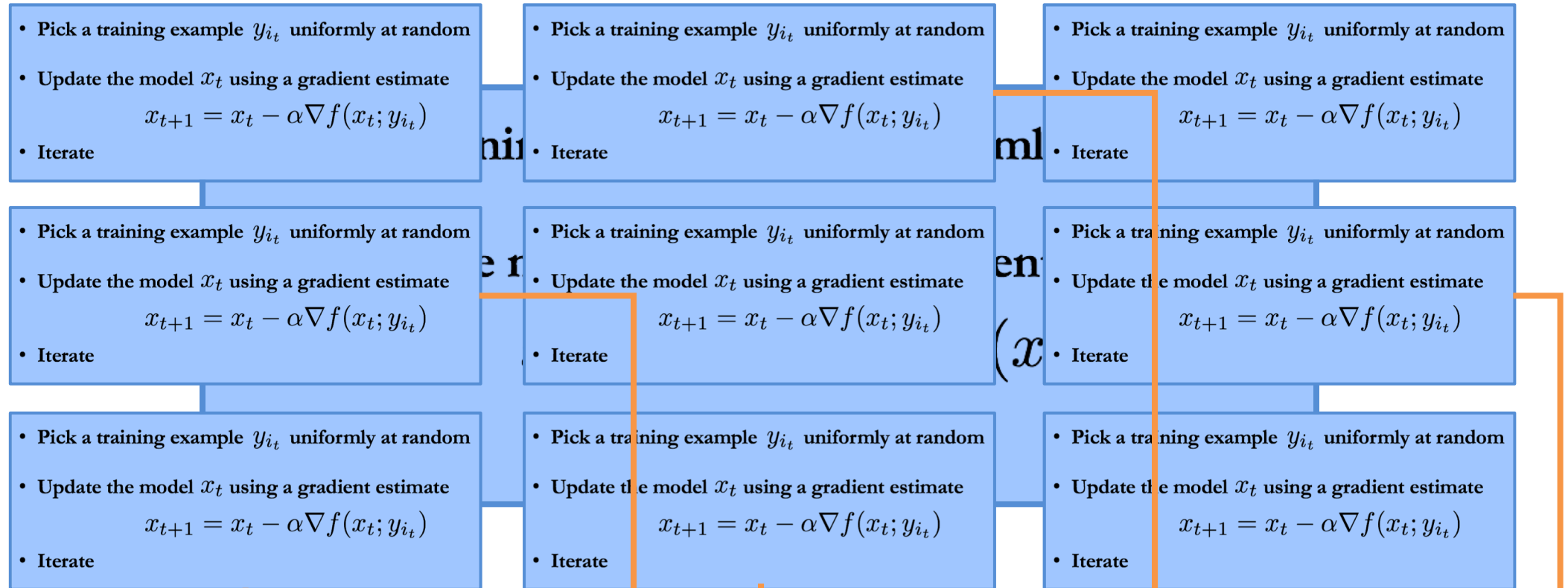


# Idea: Just Don't Synchronize

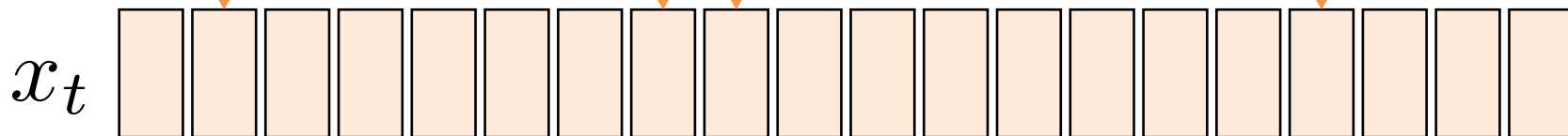
- Not synchronizing adds **errors due to race conditions**
- But our methods were already noisy — **maybe these errors are fine**
- If we don't synchronize, get **almost perfect parallel speedup**

# Fast Parallel SGD: HOGWILD!

## Multiple parallel workers



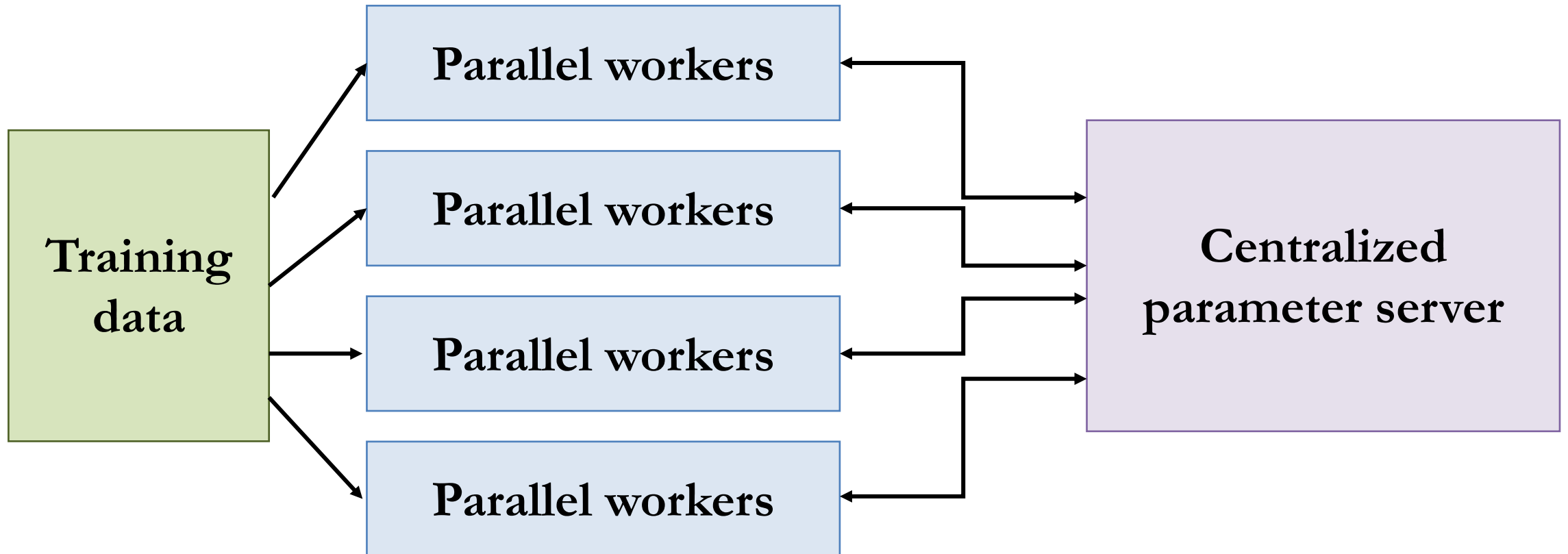
Asynchronous parallel updates (no locks) to a single shared model



# The Parameter Server Model

# Parameter server model

- A common model for distributed ML



# Parameter server (continued)

- 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**.

# Questions?

- Upcoming things
  - Paper Review #6a or #6b — **due today**
  - Programming Assignment 2 — **due today**
  - Paper Presentation #7a and #7b **on Wednesday**