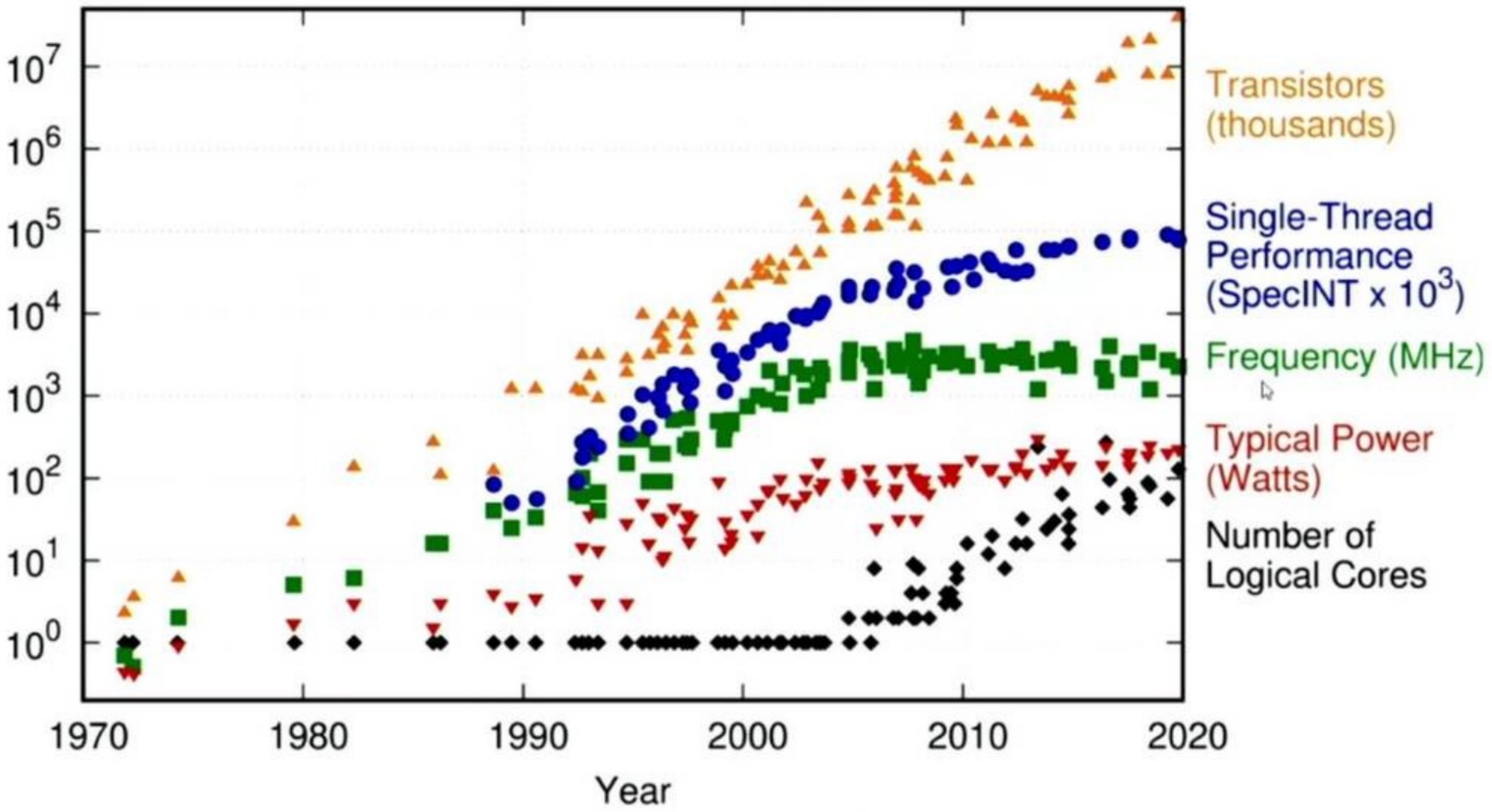
# Parallelism
# 6.S079 Lecture 16

Xinjing Zhou

4/16/2022

# Hardware Trend



# Parallelism Goal

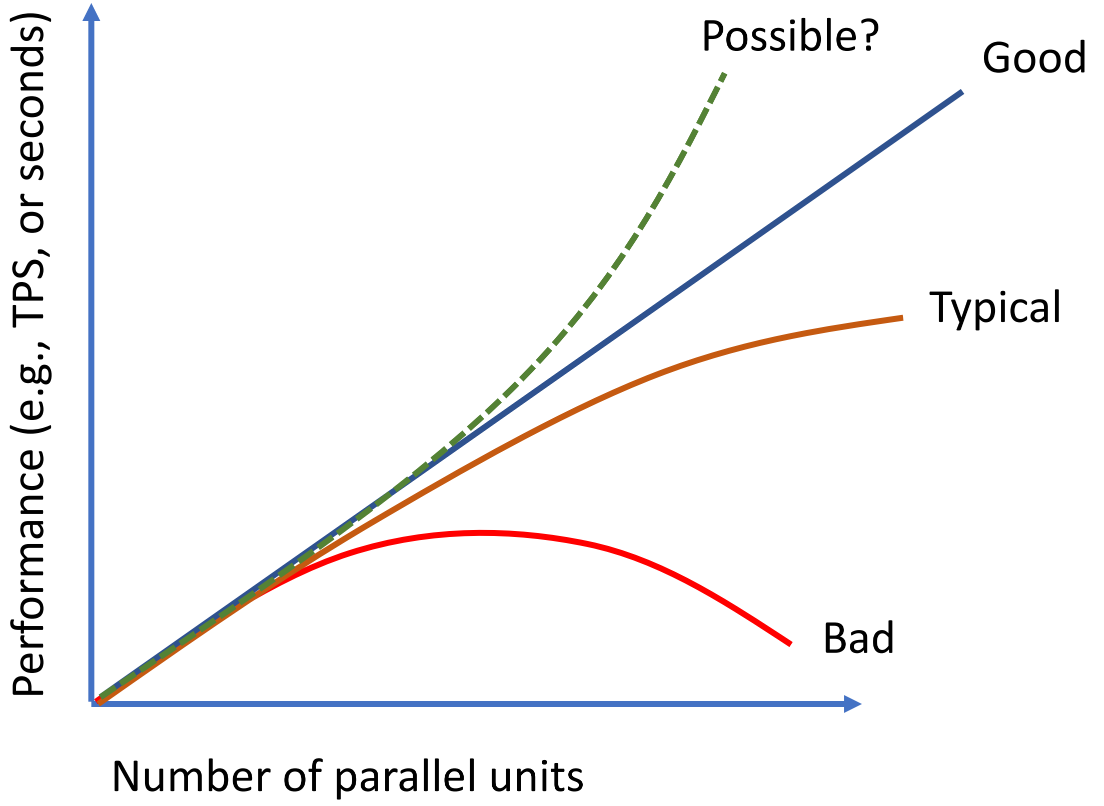
- Make a job faster by running on multiple processors

- What do we mean by faster?

$$speed\ up = \frac{old\ time}{new\ time}\ \text{on same problem, with N times more hardware}$$

# Speedup Goal

- Linear?




$$speed\ up(P,N) = \frac{1}{(1-P)+\frac{P}{N}}$$

Sequential

Parallelizable

# Barriers to Linear Scaling

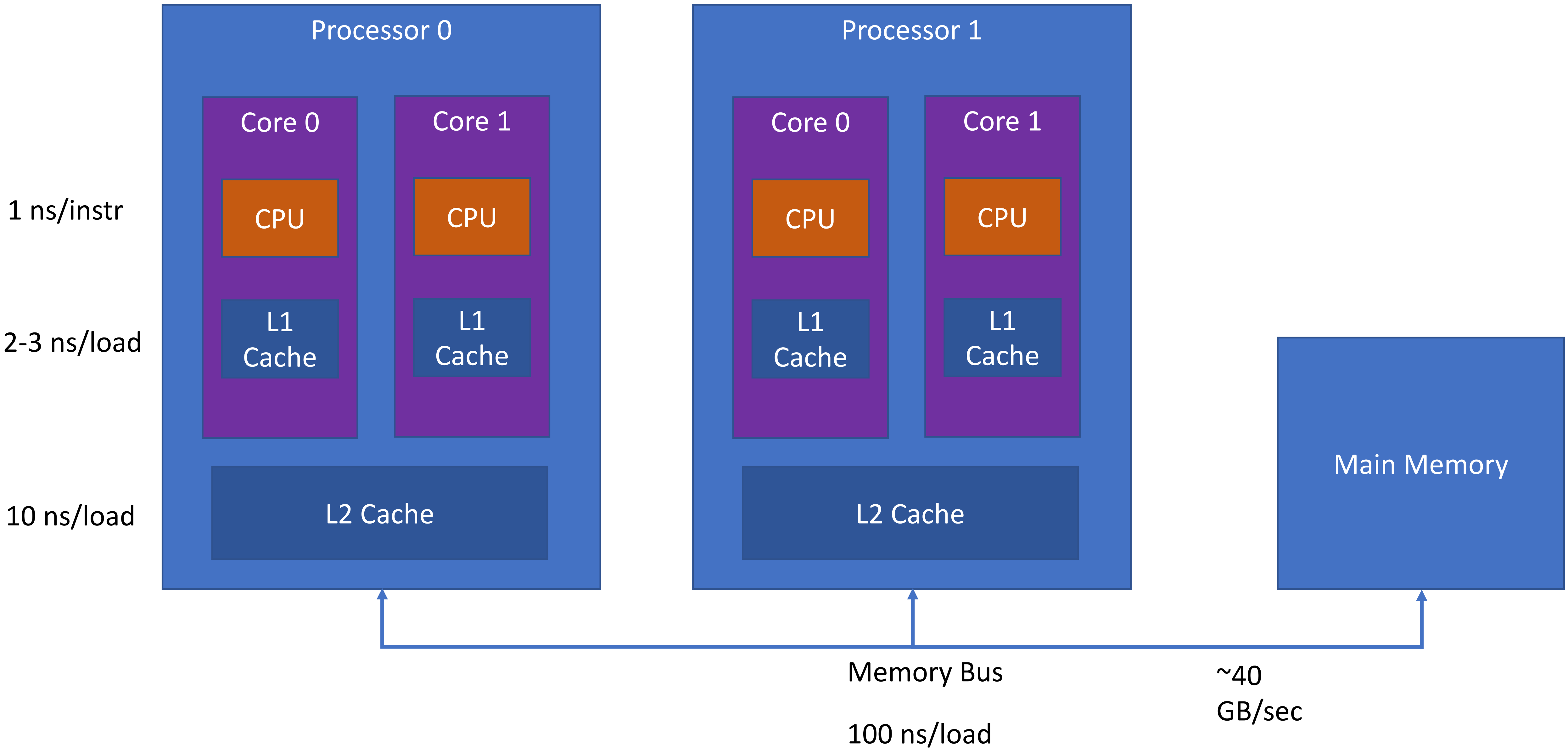
- Startup times
  - e.g., may take time to launch each parallel executor
- Interference
  - processors depend on some shared resource
  - E.g., input or output queue, or other data item
- Skew
  - Workload not of equal size on each processor
  - The name “Mike” is way more common than “Xinjing”

- *Almost all workloads will stop scaling at some point!*

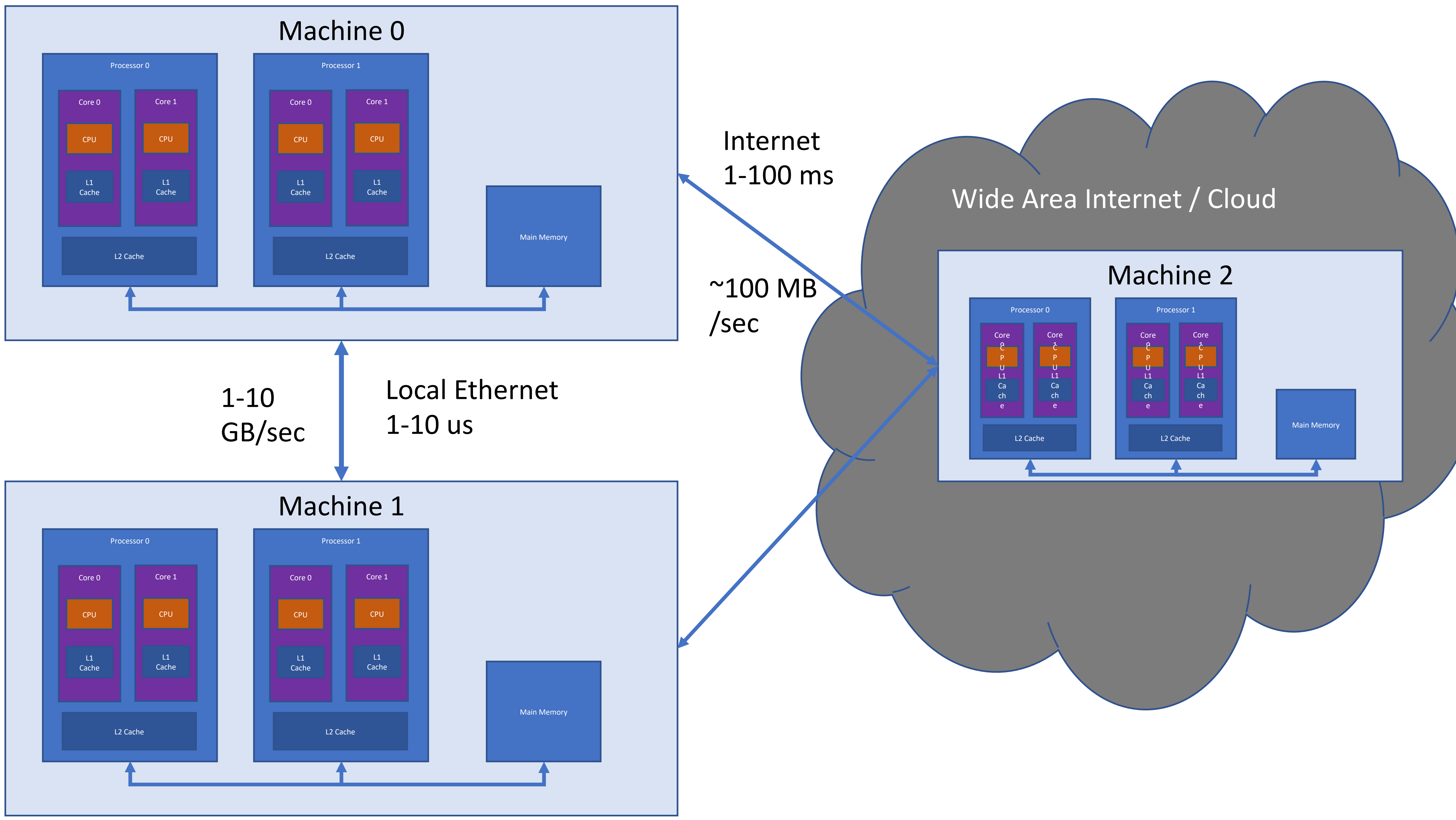
- What are some barriers in data science workloads?

# Properties of Parallelizable Workloads

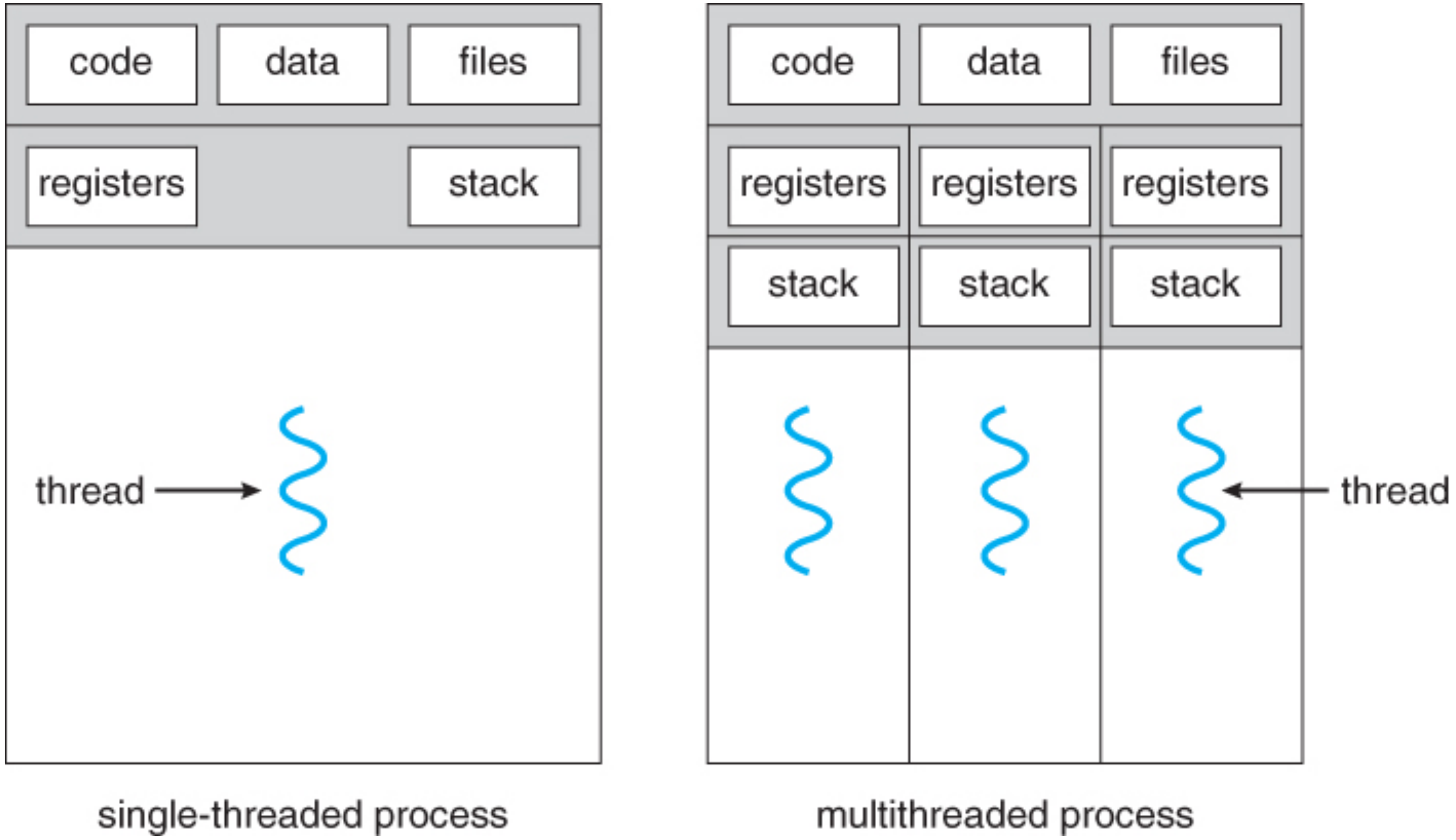
- Provide linear speedup
- Embarrassingly parallel workloads
  - Can be decomposed into small units that can be executed independently
  - E.g., Among 100 files, find records whose name field contain “Sam”
- Non-"embarrassingly parallel” workloads
  - Requires synchronization
  - E.g., Aggregation & Join




Some machines may have 2 levels of cache per core


Machine 0
Processor 0
Core 0
CPU
L1 Cache
Core 1
CPU
L1 Cache
L2 Cache
Processor 1
Core 0
CPU
L1 Cache
Core 1
CPU
L1 Cache
L2 Cache
Main Memory
Internet
1-100 ms
~100 MB /sec
1-10 GB/sec
Local Ethernet
1-10 us
Machine 1
Processor 0
Core 0
CPU
L1 Cache
Core 1
CPU
L1 Cache
L2 Cache
Processor 1
Core 0
CPU
L1 Cache
Core 1
CPU
L1 Cache
L2 Cache
Main Memory
Wide Area Internet / Cloud
Machine 2
Processor 0
L2 Cache
Processor 1
L2 Cache
Main Memory

# Threads vs Processes




https://www.cs.uic.edu/~jbell/CourseNotes/OperatingSystems/4_Threads.html

# Python Threads API

```
import threading

t = threading.Thread(target=func_name, args=(a1,a2,…))
t.start()   #start thread running – main thread continues
t.join()   #wait for thread to finish

lock = threading.Lock()   #create a lock object
lock.acquire() #acquire the lock; block if another thread has it
/// Critical section for safely accessing shared resources
lock.release()  #release the lock
```

**Problem: Python Global Interpreter Lock (GIL)**
**Only one thread can be executing python code at once**

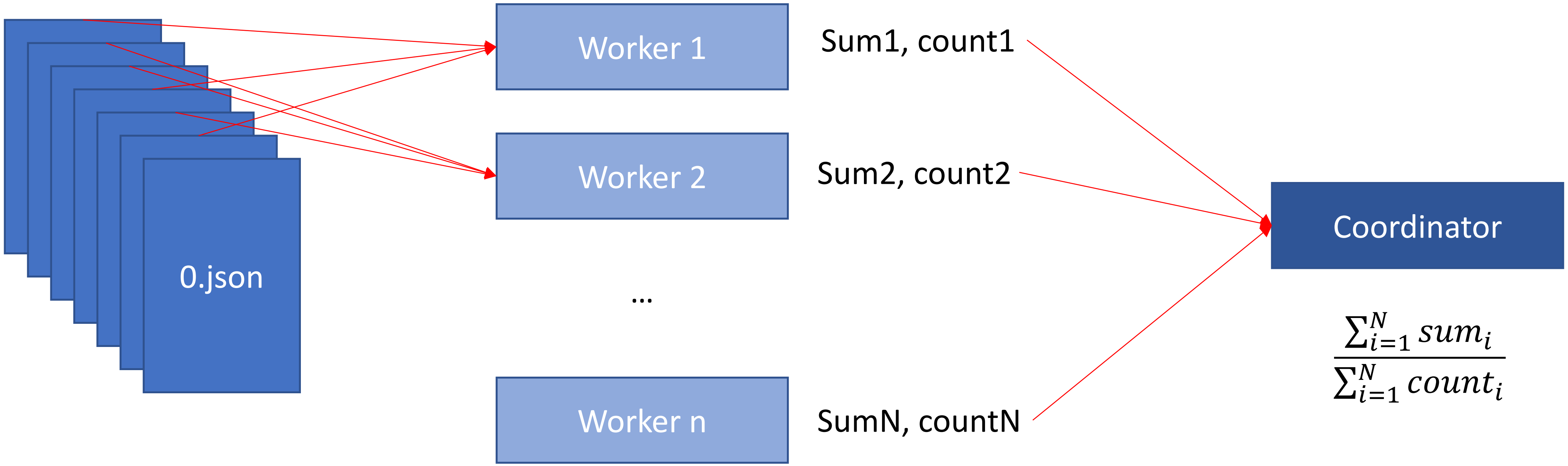
# Python Multiprocessing API

```
import multiprocessing

p = multiprocessing.Process(target=func_name, args=(a1,a2,…))
p.start()   #start process running - main process continues
p.join()   #wait for process to finish
```

# Parallel Aggregation

Task: compute average age across all people




$$\frac{\sum_{i=1}^{N} sum_i}{\sum_{i=1}^{N} count_i}$$

{"age": 30, "name": ["Michal", "Sharpe"], "occupation": "Archivist", "telephone": "285.290.9033", "address": {"address": "458 Girard Plantation", "city": "Wentzville"}, "credit-card": {"number": "5384 0033 6904 0042", "expiration-date": "06/23"}}

# Parallel Aggregation Implementation

- Use multiprocessing, not threading
- Main process creates a work queue

```
q = multiprocessing.Queue()
```

- Puts work on it, as pointers to files

```
q.put(file1); q.put(file2)
```

- Starts processes, passing them the work queue, as well as a result queue
- Processes pull from queue in a loop:

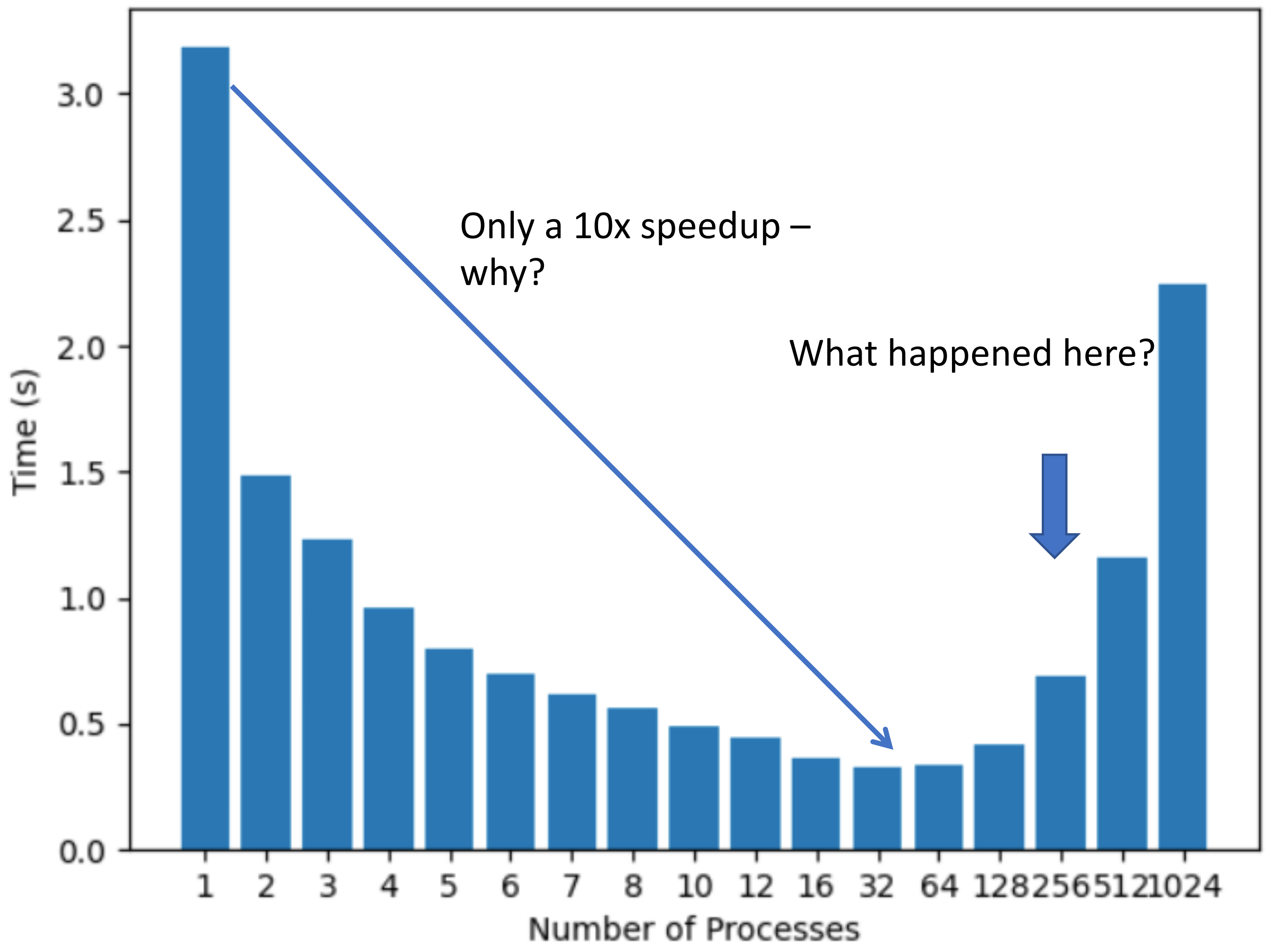
```
while True:
      f = q.get(block=False)
      process(f)
```

- Processes compute running sum and average
- Once complete, process put their running sum and average on the result queue:

```
out_q.put((age_sum, age_cnt))
```

- Main process blocks on result queue to read a result from each worker:

```
for p in procs:
      (p_sum,p_count) = out_q.get()
```

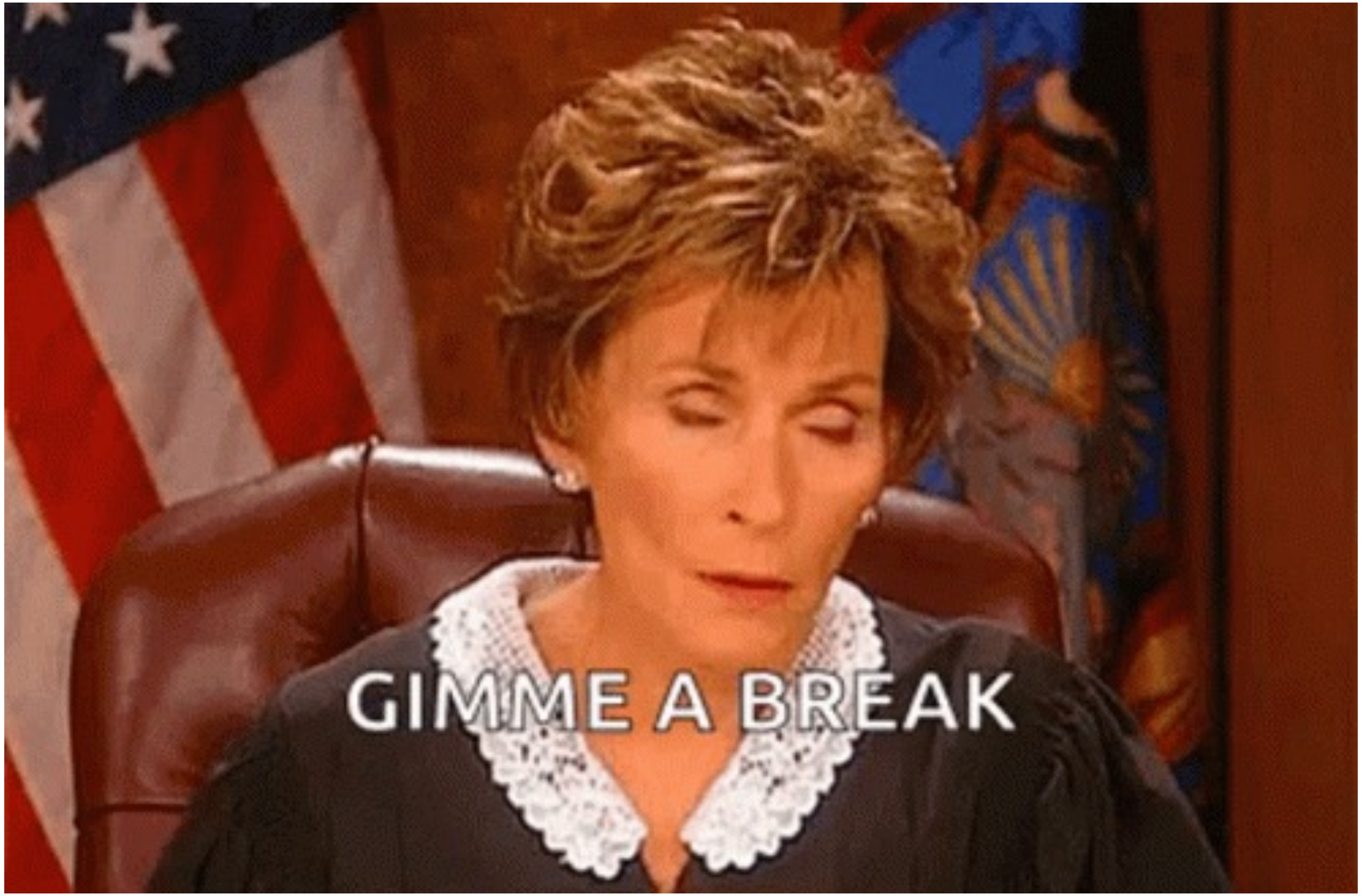

Only a 10x speedup – why?
What happened here?
Time (s)
3.0
2.5
2.0
1.5
1.0
0.5
0.0
1 2 3 4 5 6 7 8 10 12 16 32 64 128 256 512 1024
Number of Processes

# Question

Why didn’t this program speed up beyond 64 processes?
Choose all that apply

a) Not enough memory
b) Not enough processors
c) Startup overheads of launching processes
d) Too much coordination between processes

# Break



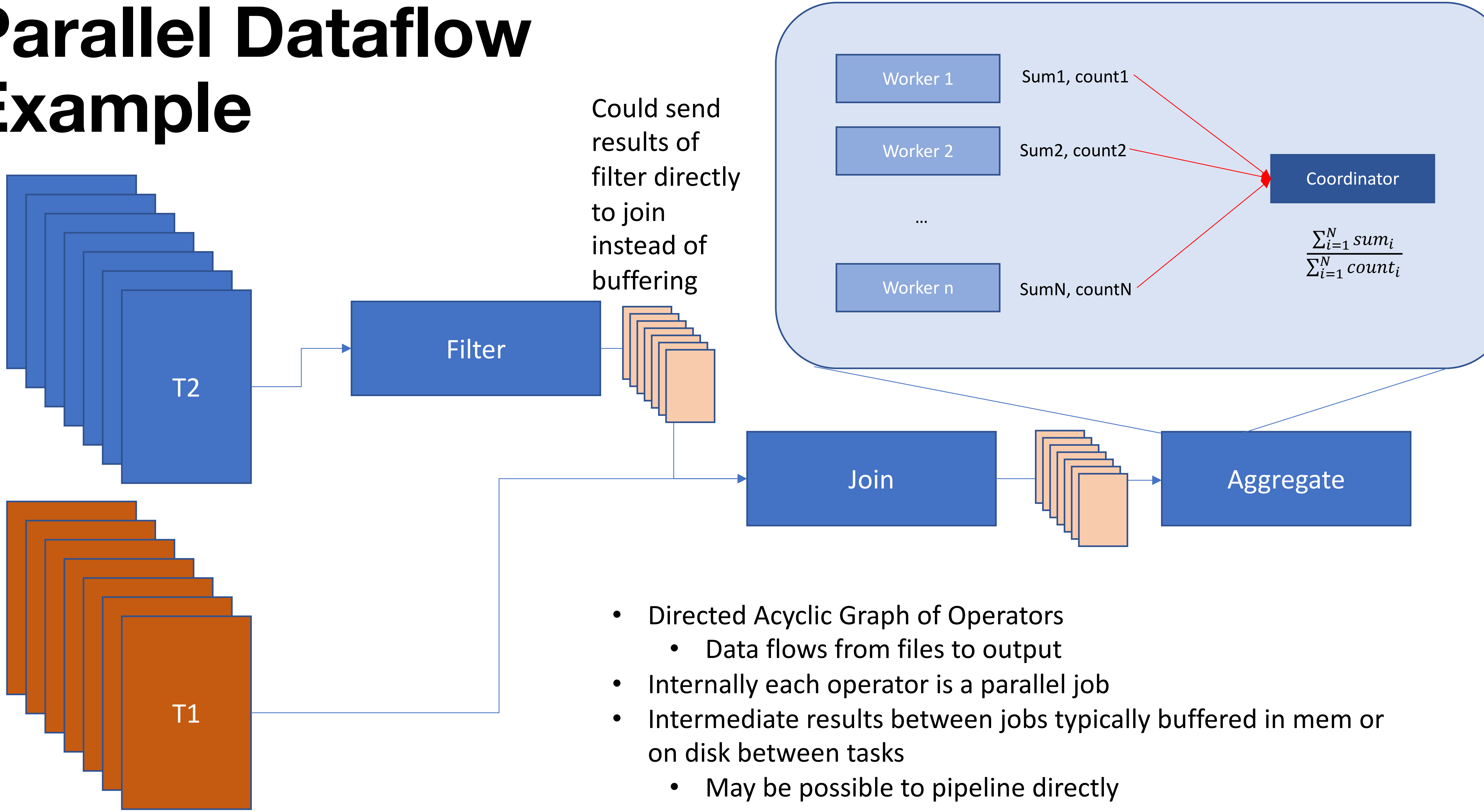
# Parallelism Approach

Split a data set into N partitions

Use M processors to process this data in parallel

We will need to come up with parallel implementations of common operators

# Parallel Dataflow Example




- Directed Acyclic Graph of Operators
  - Data flows from files to output
- Internally each operator is a parallel job
- Intermediate results between jobs typically buffered in mem or on disk between tasks
  - May be possible to pipeline directly

# Parallel Dataflow Operations

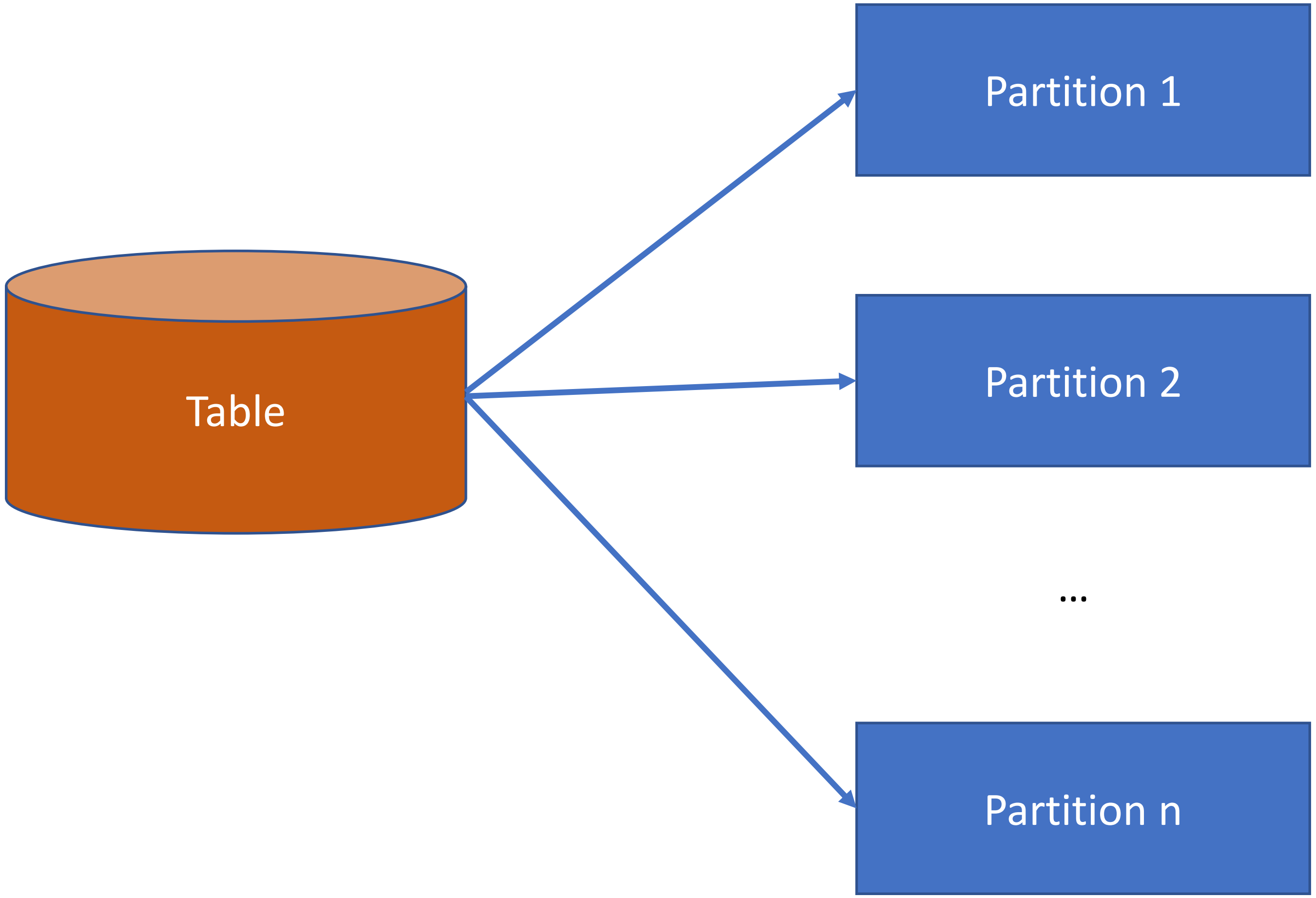
- Filter
- Project
- Element-wise or row-wise transform
- Join
  - Repartition vs broadcast
- Aggregate
- Sort

*Which of these are easy to parallelize?*

# Partitioning Strategies

- Random / Round Robin
  - Evenly distributes data (no skew)
  - Requires us to repartition for joins
- Range partitioning
  - Allows us to perform joins/merges without repartitioning, when tables are partitioned on join attributes
  - Subject to skew
- Hash partitioning
  - Allows us to perform joins/merges without repartitioning, when tables are partitioned on join attributes
  - Only subject to skew when there are many duplicate values

# Round Robin Partitioning



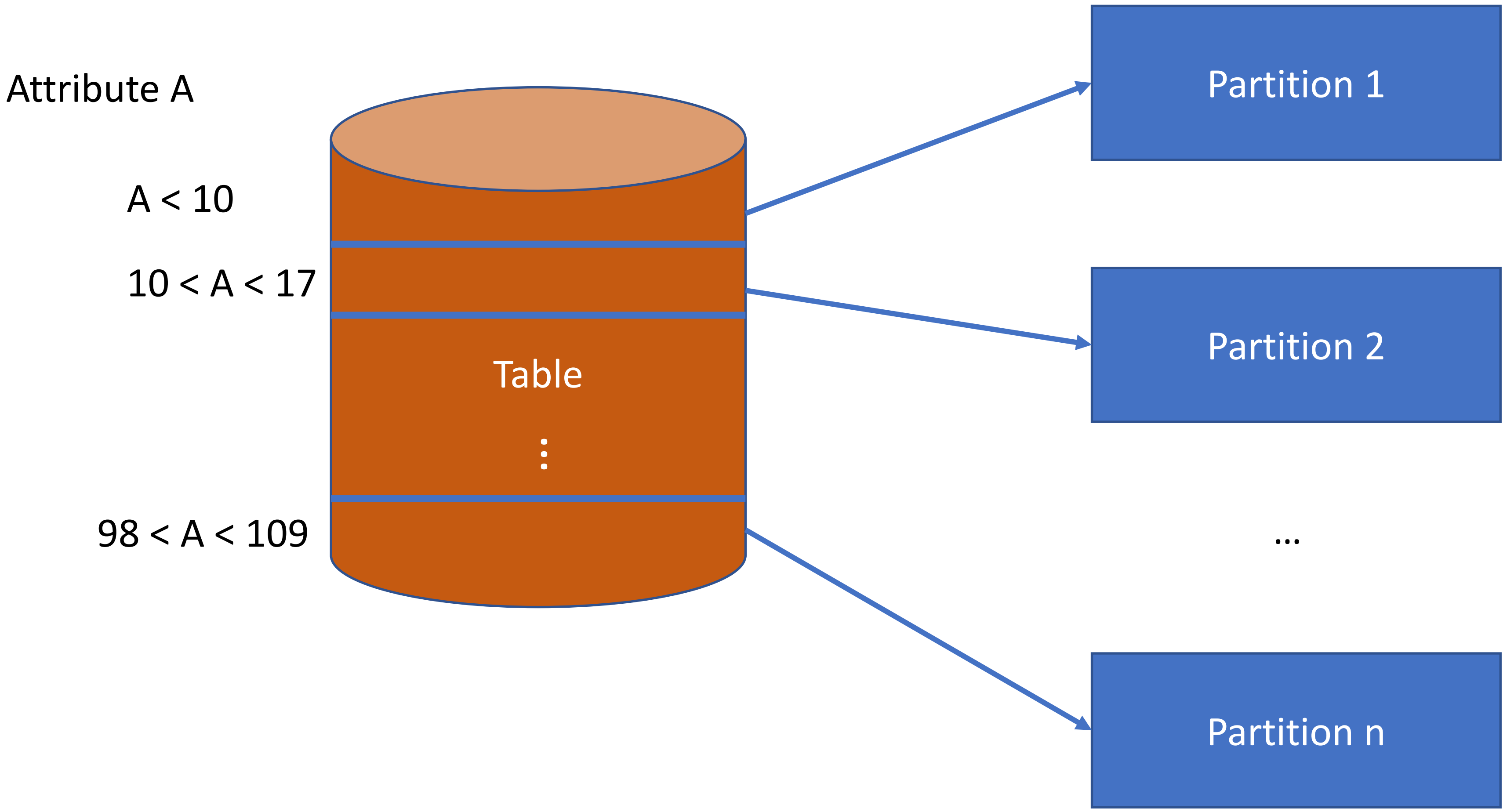

Advantages:

Each partition has the same number of records

Disadvantage:

No ability to push down predicates to filter out some partitions

# Range Partitioning




Advantages:

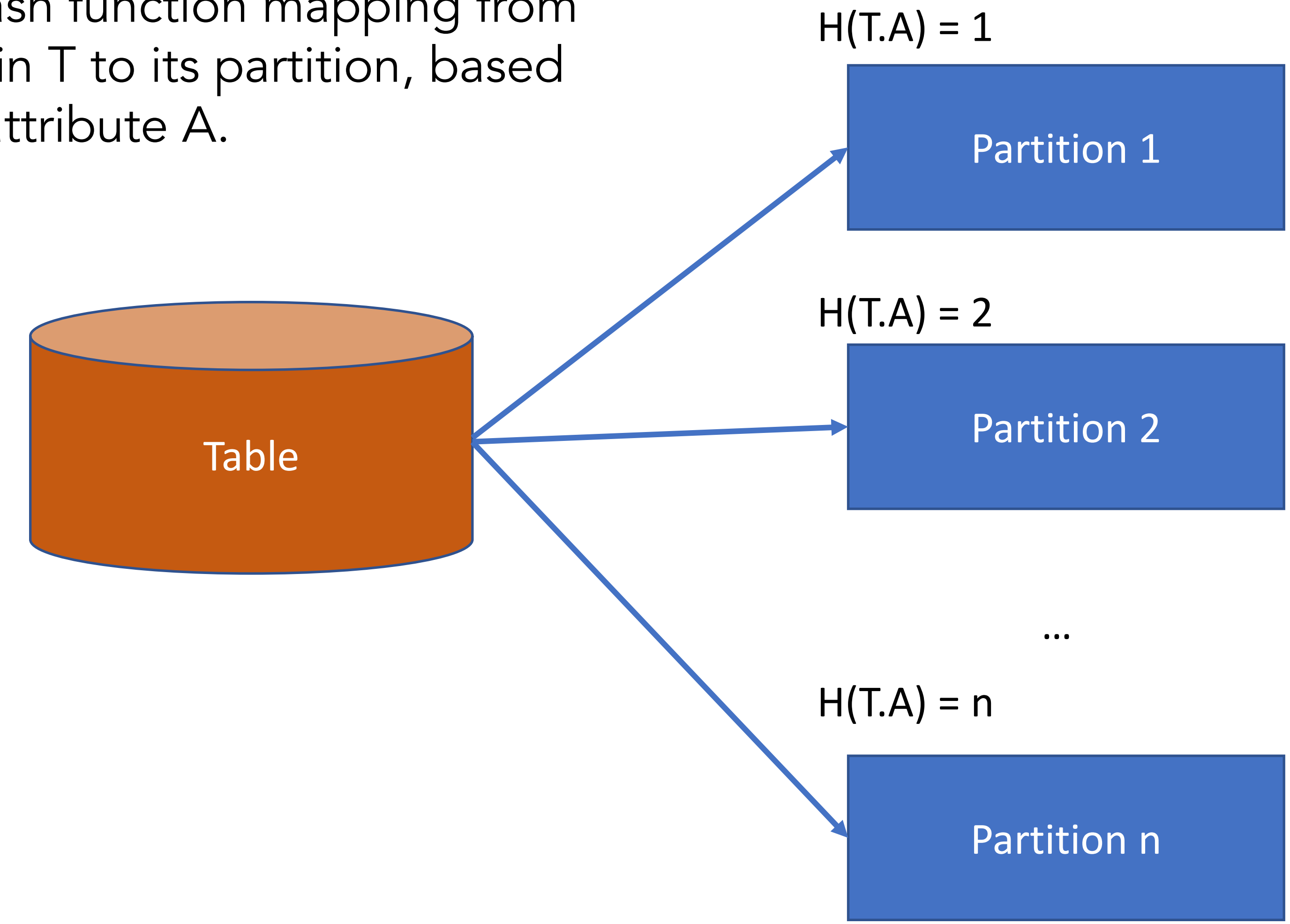
Easy to push down predicates (on partitioning attribute)

Disadvantage:

Difficult to ensure equal sized partitions, particularly in the face of inserts and skewed data

# Hash Partitioning

H(T.A) is a hash function mapping from each record in T to its partition, based on value of attribute A.




Advantages:

Each partition has about the same number of records, unless one value is very frequent

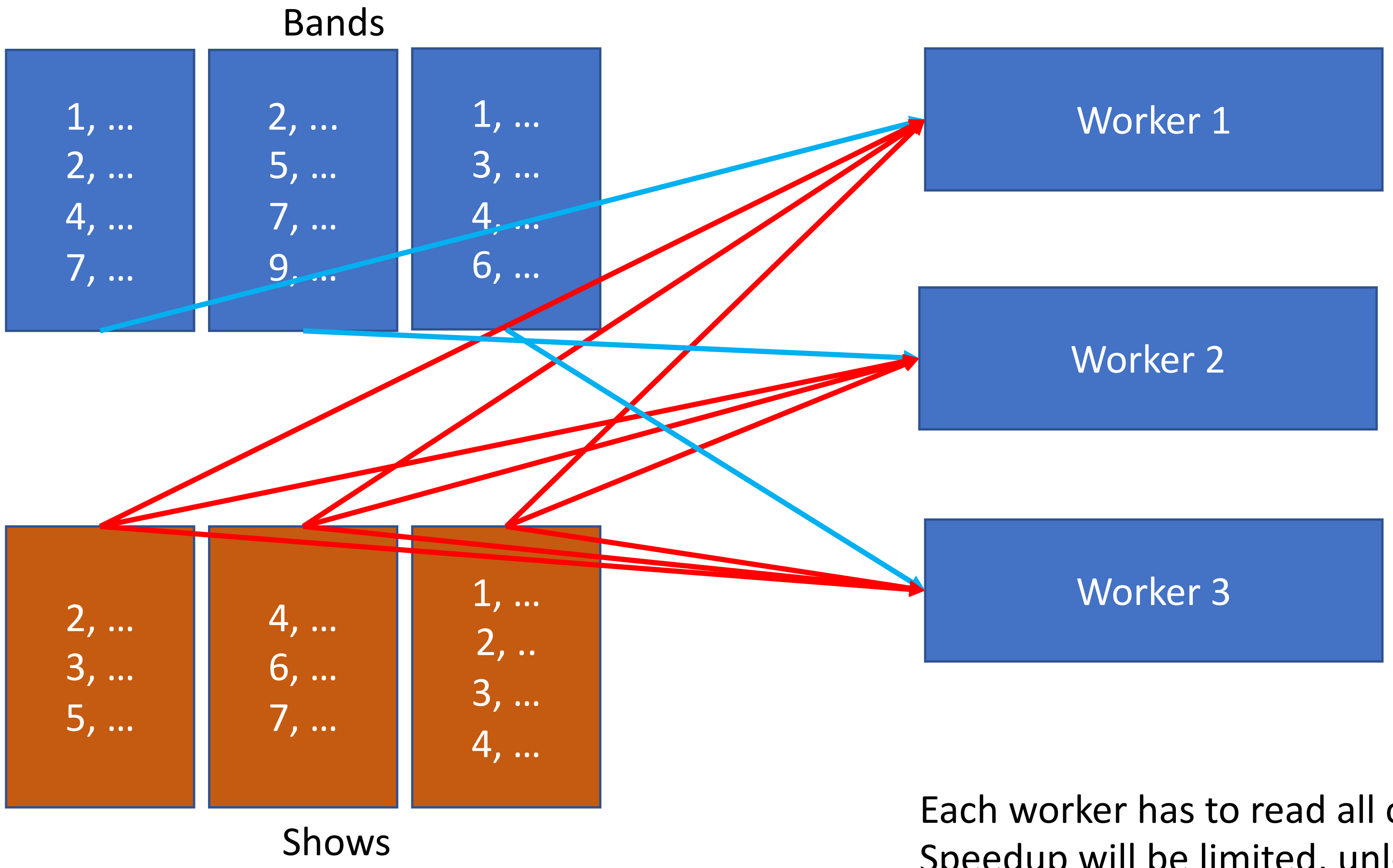
Possible to push down equality predicates on partitioning attribute

Disadvantages:

Can't push down range predicates

# Parallel Join – Random Partitioning Naïve Algo

## (1, …) indicates value of join attribute




(2, …) ⋈ (2, …)

(4, …) ⋈ (4, …)
(7, …) ⋈ (7, …)

(1, …) ⋈ (1, …)
(2, …) ⋈ (2, …)
(4, …) ⋈ (4, …)

*Must join each partition with every other partition*

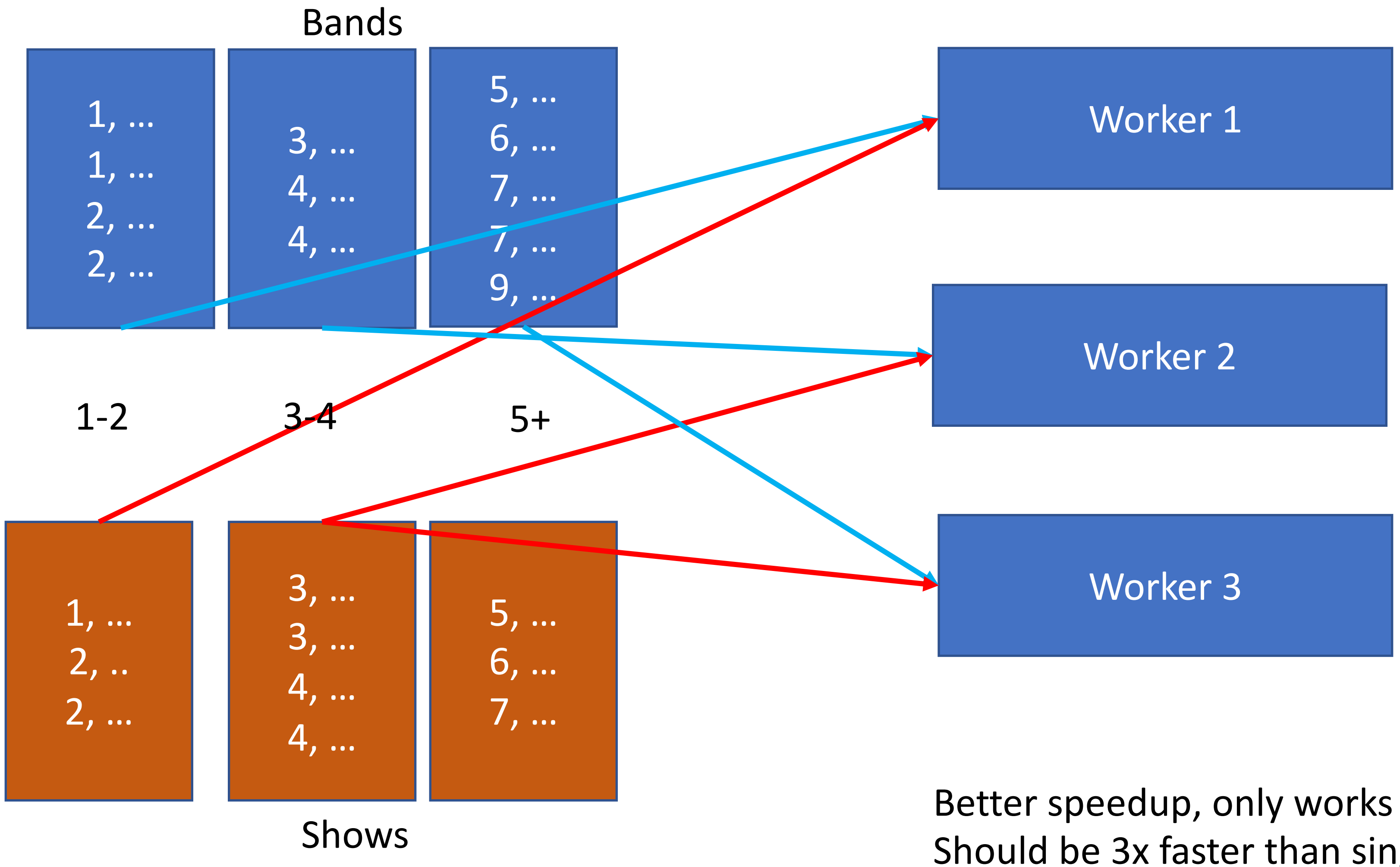
Each worker has to read all of Shows
Speedup will be limited, unless Shows is much smaller than Bands

**SELECT …**
**FROM Bands JOIN Shows on Bands.id=Shows.bandid**
**…**

# Parallel Join – Prepartitioned

## (1, …) indicates value of join attribute

*Only need to join partitions that match*



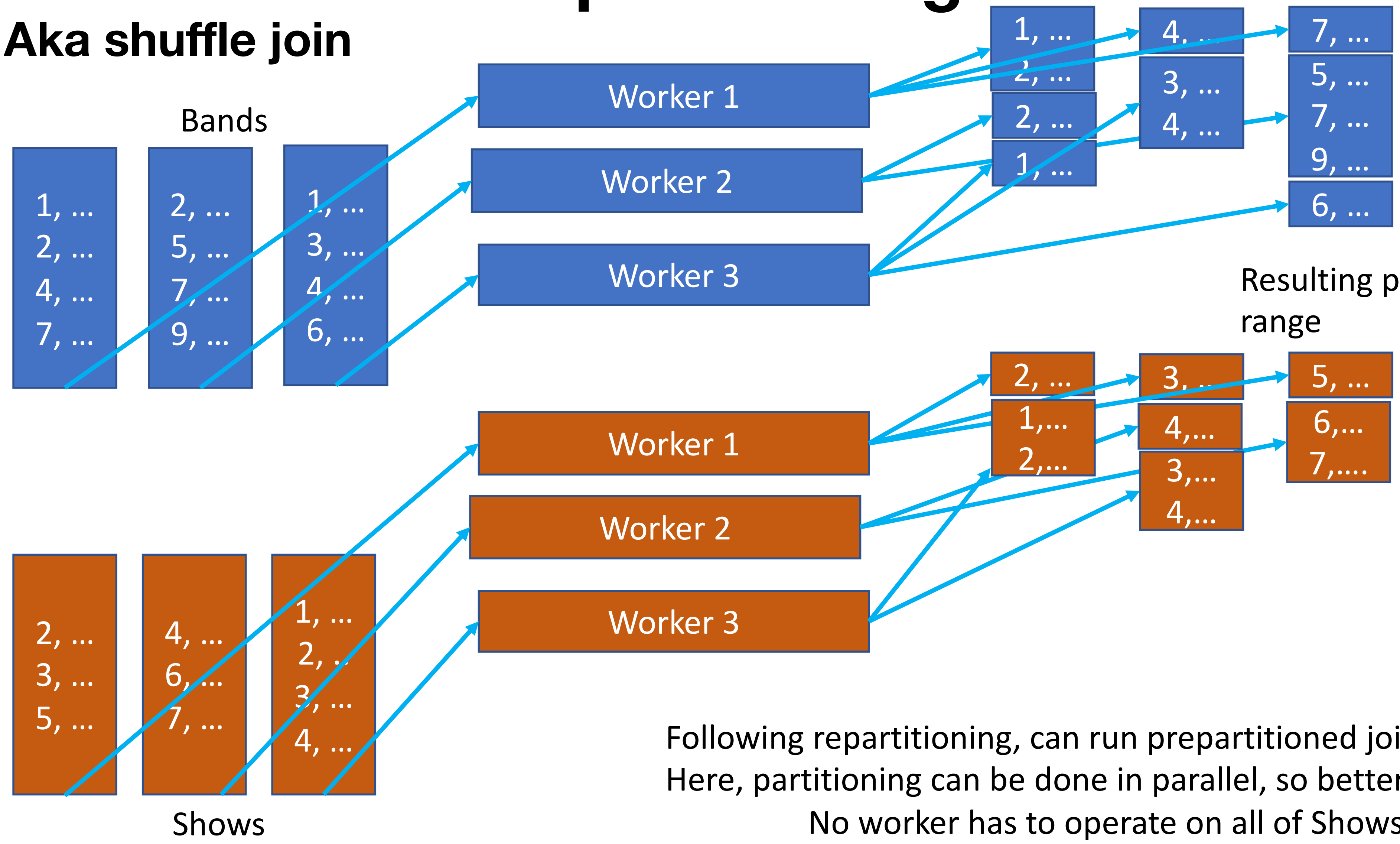

(1, …) ⋈ (1, …)
(1, …) ⋈ (1, …)
(2, …) ⋈ (2, …)
(2, …) ⋈ (2, …)
(2, …) ⋈ (2, …)
(2, …) ⋈ (2, …)
(2, …) ⋈ (2, …)

*This is what our Postgres example showed*

Better speedup, only works if data is properly pre-partitioned
Should be 3x faster than single node join
Skew problem (hashing may help)

# Parallel Join – Repartitioning

## Aka shuffle join




Resulting partitions are divided by range

Following repartitioning, can run prepartitioned join

Here, partitioning can be done in parallel, so better than naïve

No worker has to operate on all of Shows

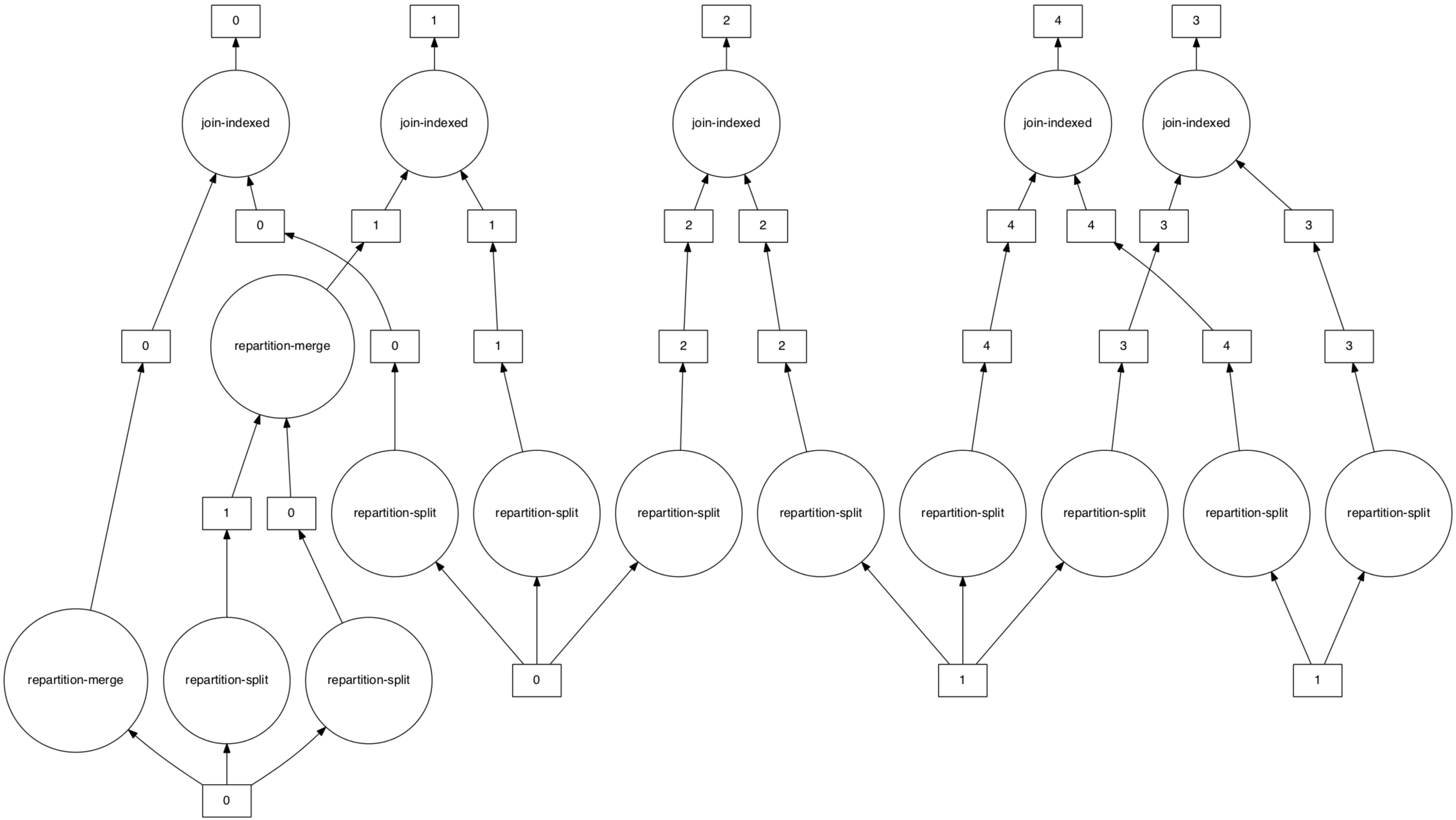
# Dask

https://dask.org

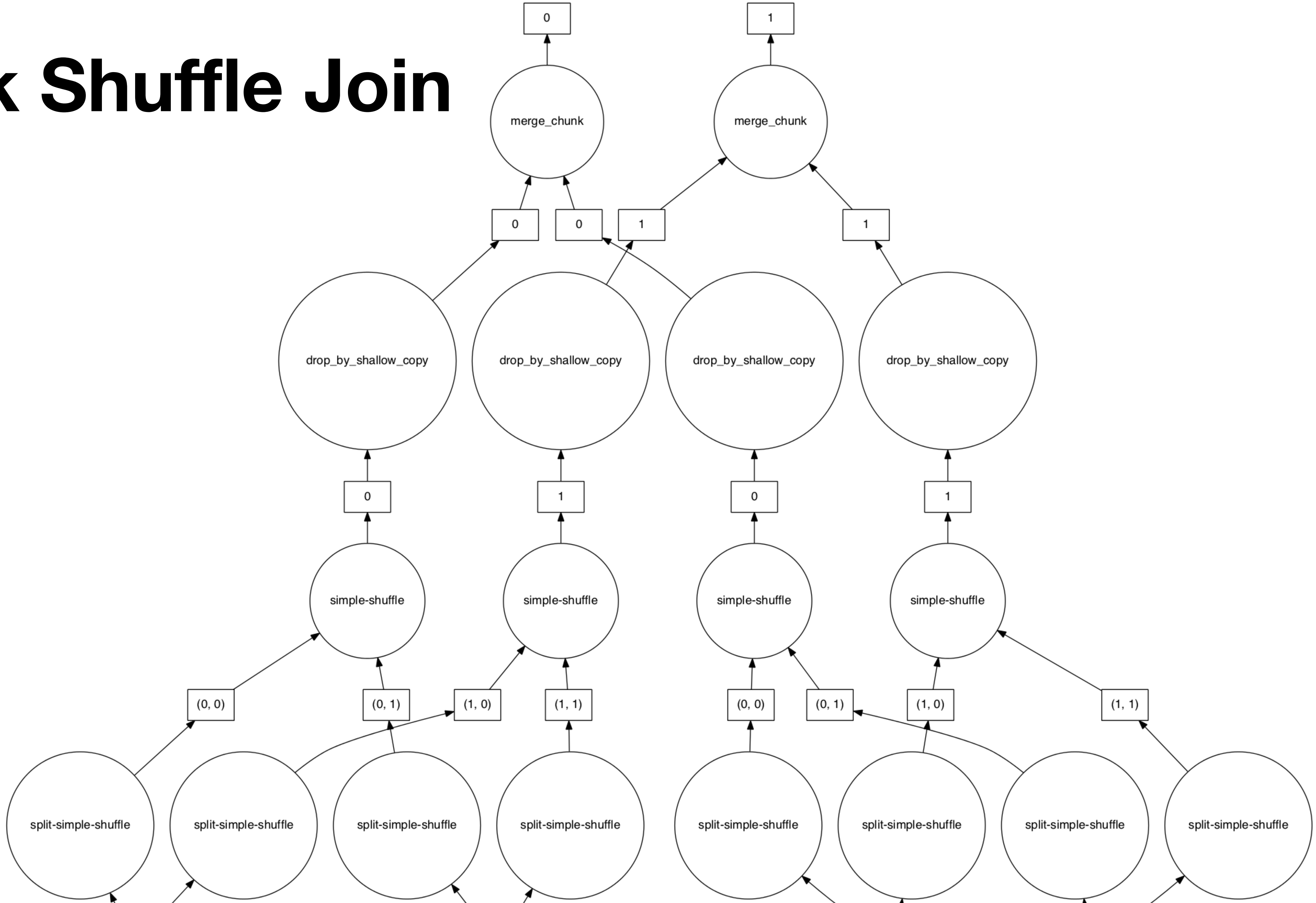
- General purpose python parallel / distributed computation framework
- Includes parallel implementation of Pandas dataframes
- Usually straightforward to translate a pandas program into a parallel implementation
  - Just use dask.dataframe instead of pandas.dataframe
  - Have to specify a parallel configuration to run on, via Client() object
    - Can be a local machine or distributed cluster
- Also has support for other types of parallelism, e.g., dask.bag class that allows parallel operation on collections of python objects

# Demo

# Dask Partitioned Join



# Dask Shuffle Join



# Many alternatives

- MapReduce / Hadoop
    - Rewrite you program as collection of parallel map() and reduce() jobs
    - Hard to do, slow()

# Spark

- In-memory DAG-based distributed computing framework
- Two sets of APIs
    - RDD: Resilient Distributed Dataset, with map-reduce like low-level operations, able to handle unstructured data.
    - Dataframe-like tabular APIs – similar to Dask
- Includes parallel implementations of ML and other operations

# Summary

- Parallelism is a good way to improve performance
- Ideal: linear speedup
  - Difficult to achieve in practice
- Some operations can be trivially parallelized with partitioned parallelism, e.g., filters and maps
- Other operations – like joins – are more difficult
- Dask is a popular open-source parallel programming library for Python
- Next lecture: Ray