Announcements

- For everyone's safety:
  - Please do not congregate after the class for Q/A -- ask questions during the lecture or make use of Piazza and OH
  - If you are sick, please watch the lectures remotely
  - Wear your mask properly **covering your nose and mouth entirely at all times during the lecture**
- For any private communication, use course staff email < ds-staff-f21-private@lists.andrew.cmu.edu>. Not individual instructor email addresses.

15-440/640
Distributed Systems

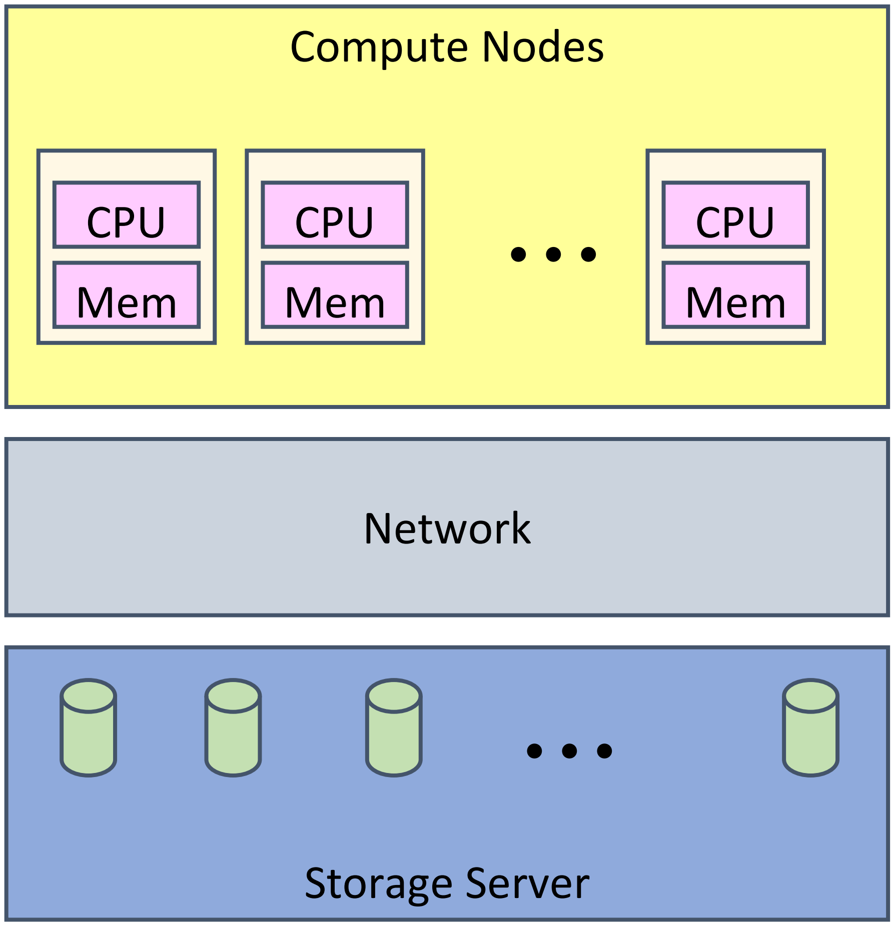
- Finish up distributed computation (MPI & MapReduce)
- In-memory cluster compute (Spark)
- Distributed ML

Cluster Computing

1. High-performance computing (HPC)
   - Message Passing Interface (MPI)

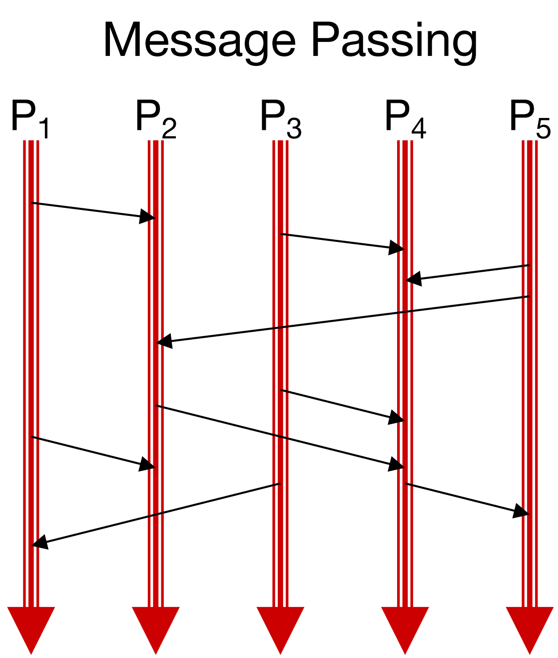
2. Cluster computing
   - MapReduce

Recall: Typical HPC Machine



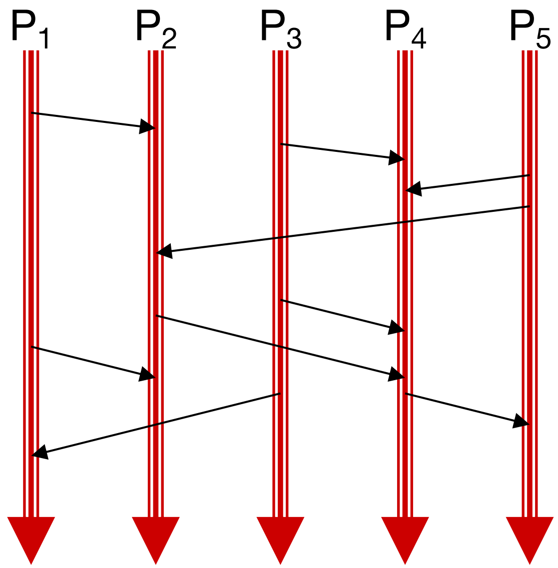
- Compute Nodes
  - Lots of high end processor(s)
  - Lots of RAM
- Network
  - Specialized
  - Very high performance
- Storage Server
  - RAID-based disk array

Recall: Typical HPC Operation



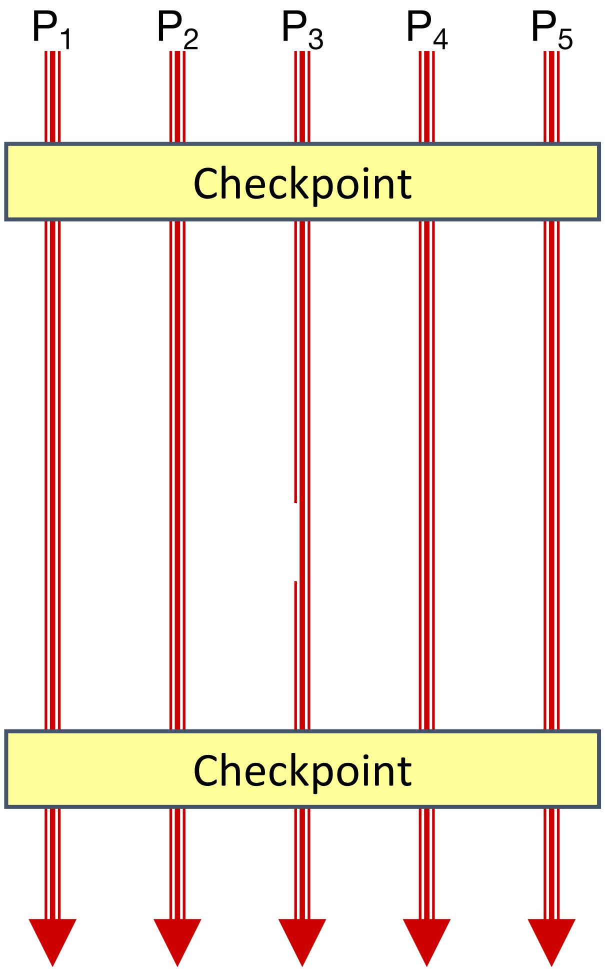
- Characteristics
  - Long-lived interdependent processes
  - Partitioning: exploit spatial locality
  - Hold all program data in memory (no disk access)
  - High bandwidth communication
- Strengths
  - High utilization of resources
  - Effective for many scientific applications
- Weaknesses
  - Requires careful tuning of application to resources
  - Intolerant of any variability

HPC Fault Tolerance



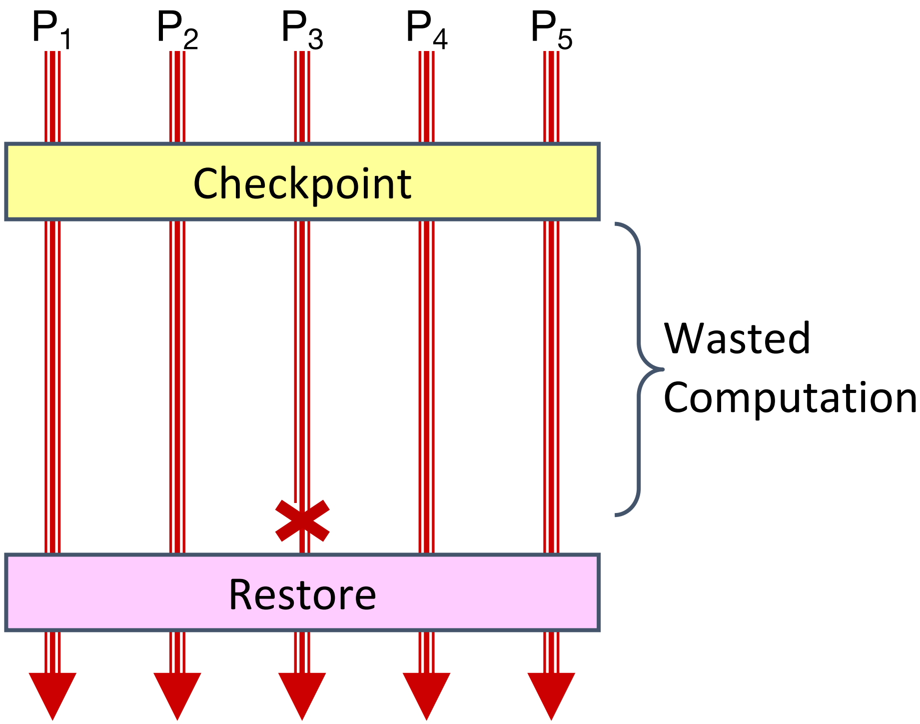
- Tightly coupled processes
  - Failure of one processes prevents all others from progressing
- How to ensure correct execution in presence of failures?

HPC Fault Tolerance



- Tightly coupled processes
  - Failure of one processes prevents all others from progressing
- How to ensure correct execution in presence of failures?
- Checkpointing
  - Periodically save system state of all processes
  - Stored in reliable storage that can withstand targeted failure
  - Roll back to error-free state in case of failure

HPC Fault Tolerance



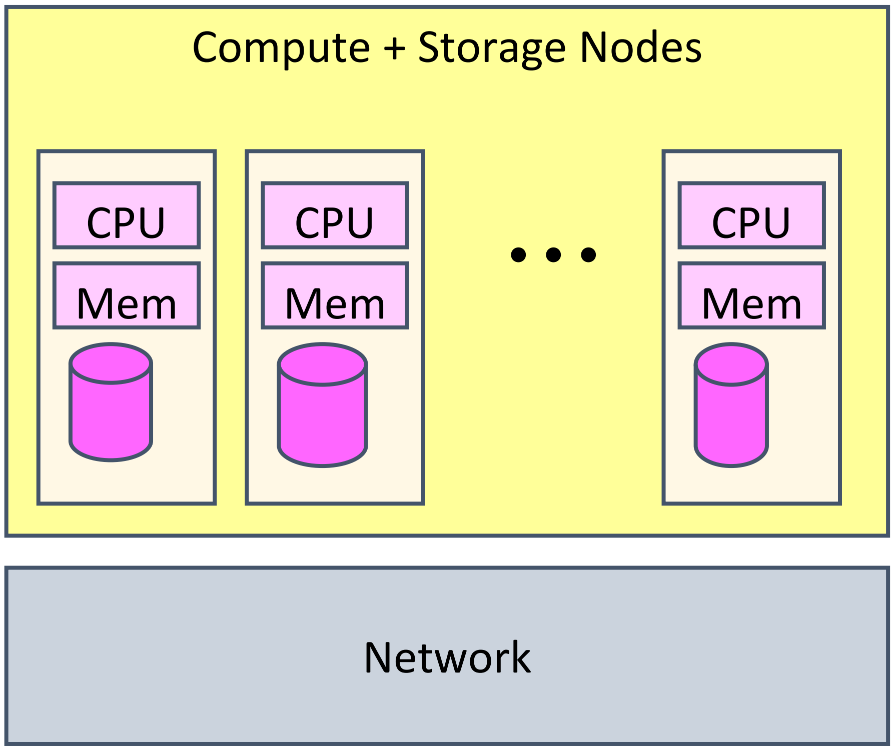
- Rollback upon failure
  - Restore state to that of last checkpoint
  - All intervening computation wasted
- Design decisions
  - Asynchronous or synchronous?
  - How often to checkpoint?
  - What data to checkpoint?
  - Who checkpoints: application or system?
- Significant I/O traffic
- Very sensitive to number of failing components

Cluster Computing

1. High-performance computing (HPC)
   - Message Passing Interface (MPI)

2. Cluster computing
   - MapReduce

Typical Cluster Computing



- Off-the-shelf servers
  - Collocation of compute and storage
  - Medium-performance processors
  - Modest memory
  - A few disks
- Network
  - Conventional Ethernet switches
  - 10s Gb/s

Oceans of Data, Skinny Pipes

- 10 Terabytes
  - Easy to store
  - Hard to move

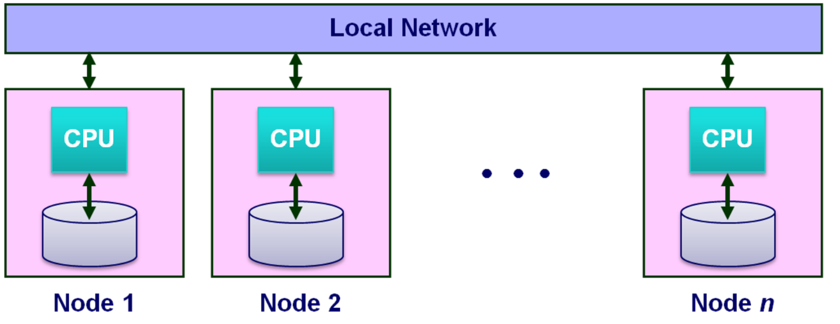
Disks	MB / s	Time
Seagate HDDs	~100s	> Few hours
Networks	**MB / s**	**Time**
Gigabit Ethernet	< 125	> 23 hours
10GE	< 1,200	> 2.4 hours
100GE	< 12,000	15 minutes

Data-Intensive System Challenge

How to process 10 TB in a few minutes?

- Distribute data over 100+ disks
  - Assuming uniform data partitioning

Key idea: partition compute tasks and run where data is stored



- Compute using 100+ processors
  - Without having to move data

- System Requirements
  - Lots of processors with co-located disks
  - Nodes located in close proximity
    - Within reach of fast, local-area network

How To Program A Cluster?

Example:

Many text files (e.g. logfiles, crawled webpages,..)

Stored in DFS on thousands of machines (GFS)

Assume you have access to all those machines

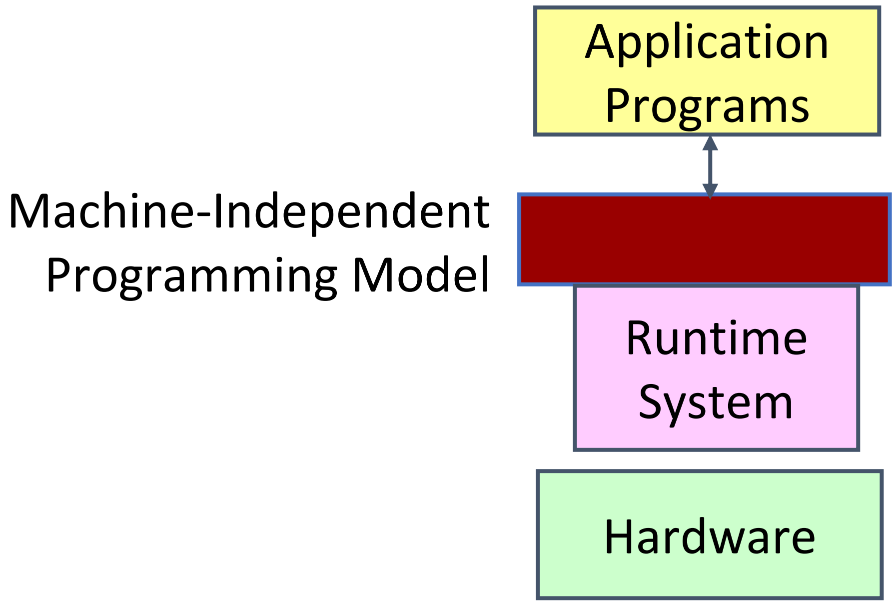
How do you find the frequency of words, such as , “440”, “error”, “p4” ?

What do you do if tasks run for > 1 week?

e.g., machines fail, get rebooted

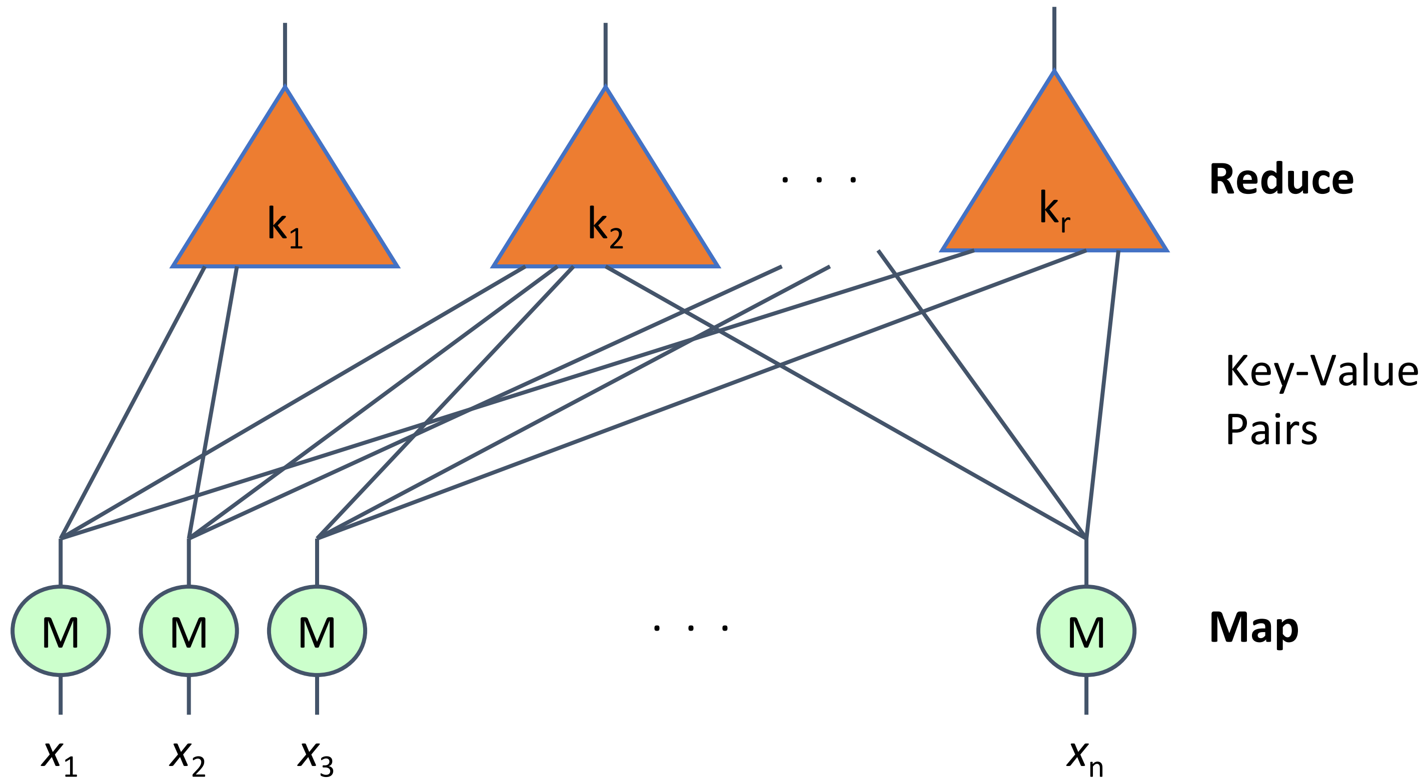
What do you do if a variant of this task comes up?

Cluster Programming Model



- Application programs written in terms of high-level data operations
- Runtime system controls scheduling, load balancing, fault-tolerance
- This is idealized: no perfect cluster programming model, in practice
- One popular model: MapReduce

MapReduce Cluster Model

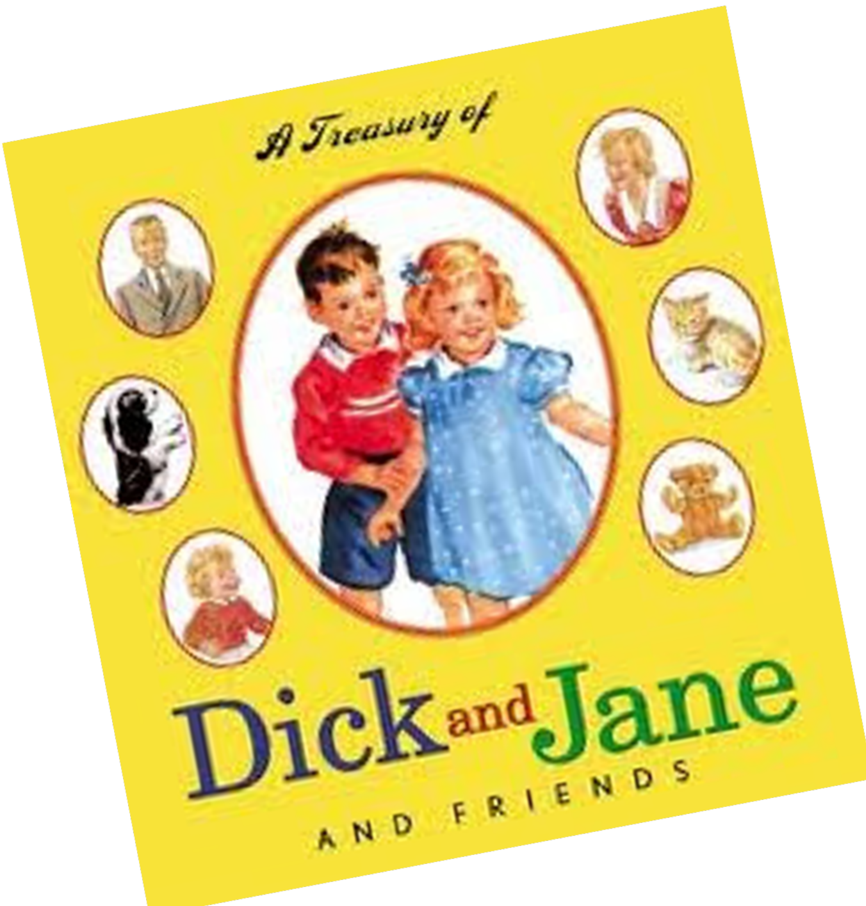


- Map: Map computation across many objects
  - Runtime schedules "mappers" so as to minimize data movement
- Reduce: Aggregation of results

Dean & Ghemawat: "MapReduce: Simplified Data Processing on Large Clusters", OSDI 2004

Example MapReduce

- Calculate word frequency of a set of documents
- Example: children book in basic English

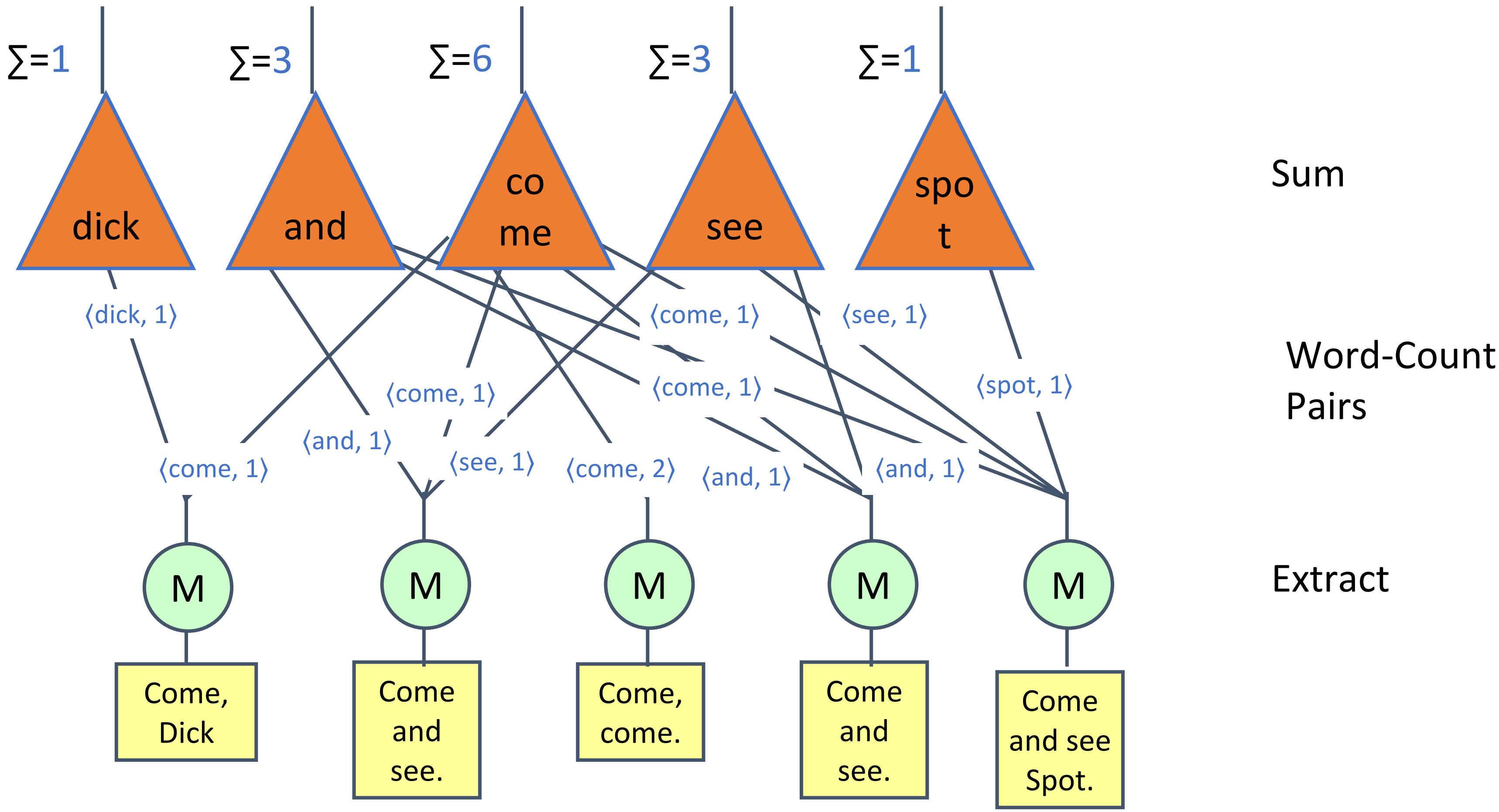


Example MapReduce

Come, Dick	Come and see.	Come, come.	Come and see.	Come and see Spot.

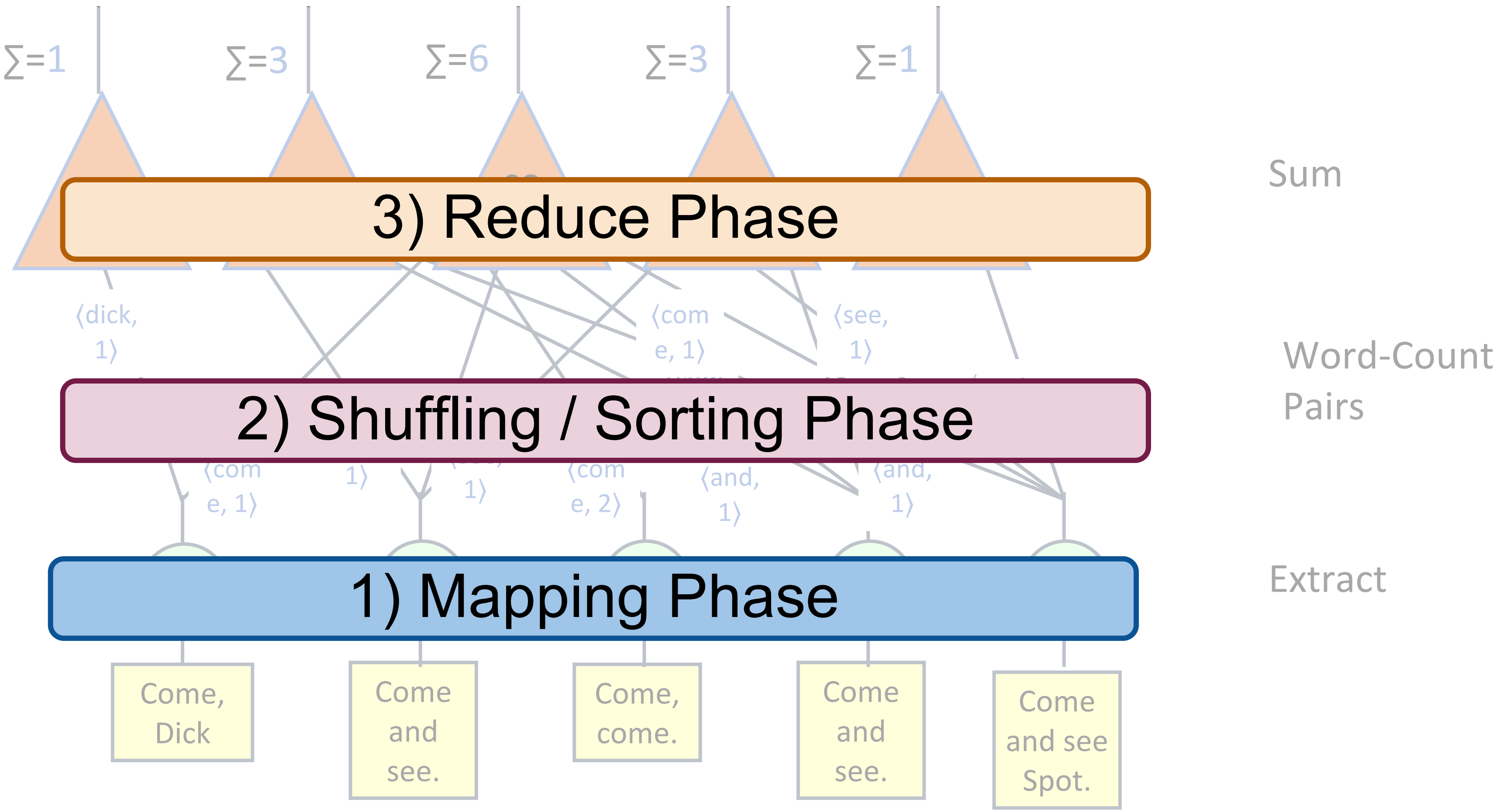
- Calculate word frequency of set of documents

Example MapReduce



- Map: generate ⟨word, count⟩ pairs for all words in document
- Reduce: sum word counts across documents

Example MapReduce



- Map: generate ⟨word, count⟩ pairs for all words in document
- Reduce: sum word counts across documents

MapReduce Implementation

- Built on Top of Cluster Filesystem
  - Provides global naming
  - Reliability via replication (3 replicas of every chunk)
- Breaks work into tasks
  - Typically #tasks >> #processors
  - Master schedules tasks on workers dynamically
- Net effect
  - Input: Set of files in reliable file system
  - Output: Set of files in reliable file system

Real-World Challenges

Fault Tolerance

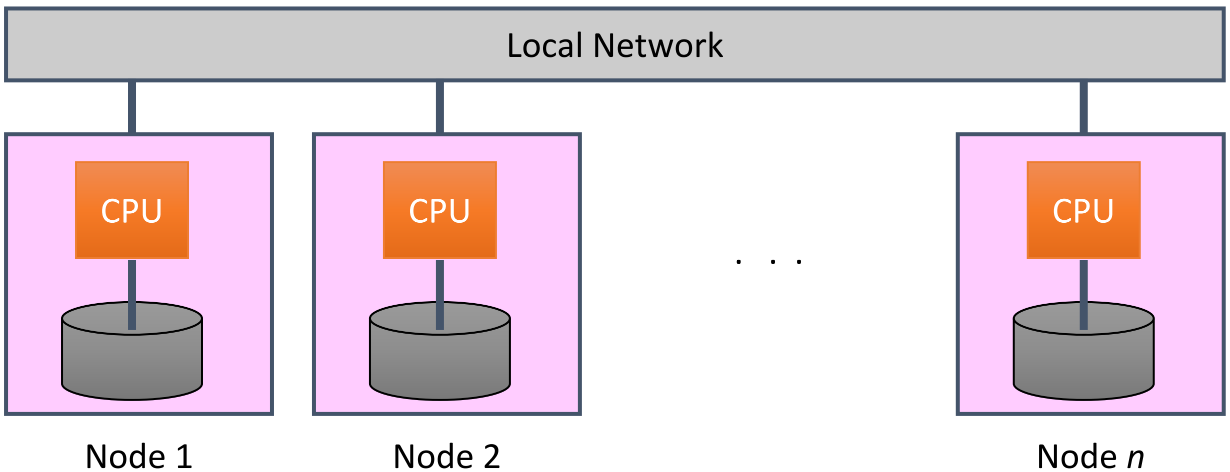
- Reliable file system is not enough
- Workers can fail even if input files available
- Map-Reduce solution
  - Detect failed worker (Heartbeat mechanism)
  - Reschedule failed task

Stragglers

- Tasks that take a long time to execute
  - Might be bugs, flaky/slow hardware (e.g., disk I/O), poor partitioning, etc.
- Map-Reduce solution:
  - When done with most tasks, reschedule any remaining executing tasks
  - Keep track of redundant executions
  - Significantly reduces overall run time

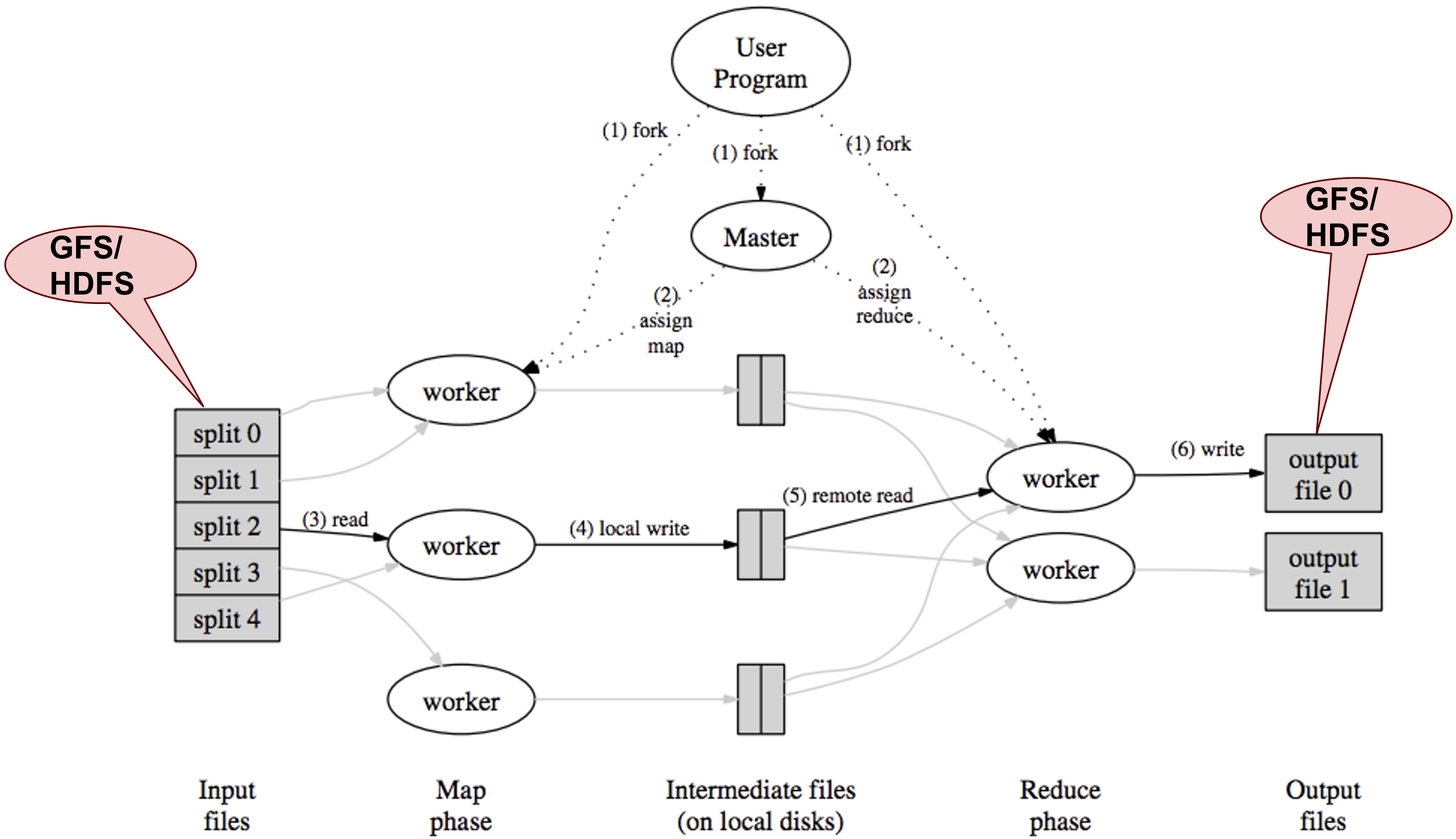
Hadoop Project

- Colocate compute and storage: HDFS + MapReduce



- HDFS Fault Tolerance (3 copies of file)
- "Locality-preserving" compute job placement priority order
    1) On same node as HDFS chunk
    2) On same rack as HDFS chunk
    3) Anywhere else (access over HDFS network)
- MapReduce programming environment

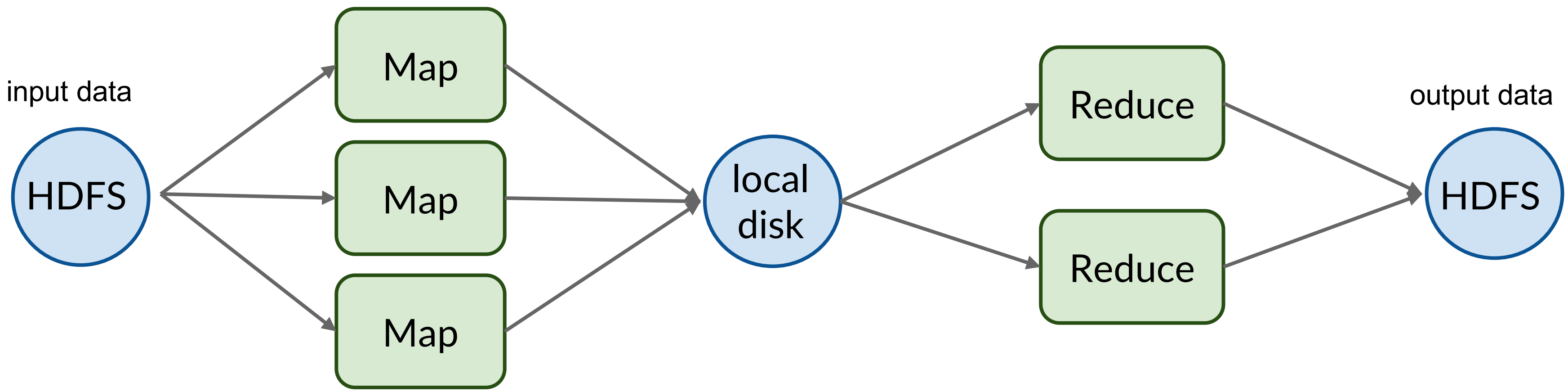
MapReduce Execution



Dean & Ghemawat: “MapReduce: Simplified Data Processing on Large Clusters”, OSDI 2004

Cluster Computing

MapReduce (Hadoop) Framework:



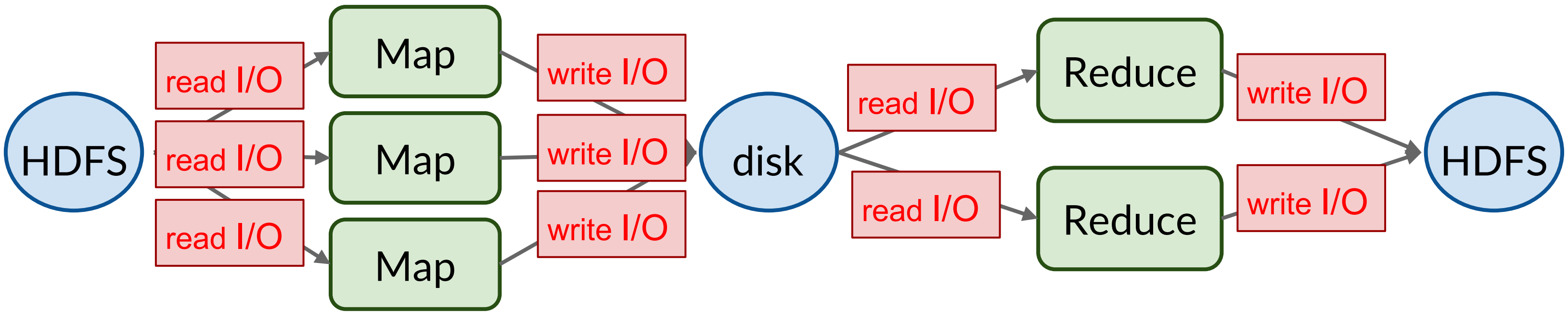
Key features: **fault tolerance** and **high throughput**

⇒ Simplified data analysis on large, unreliable clusters

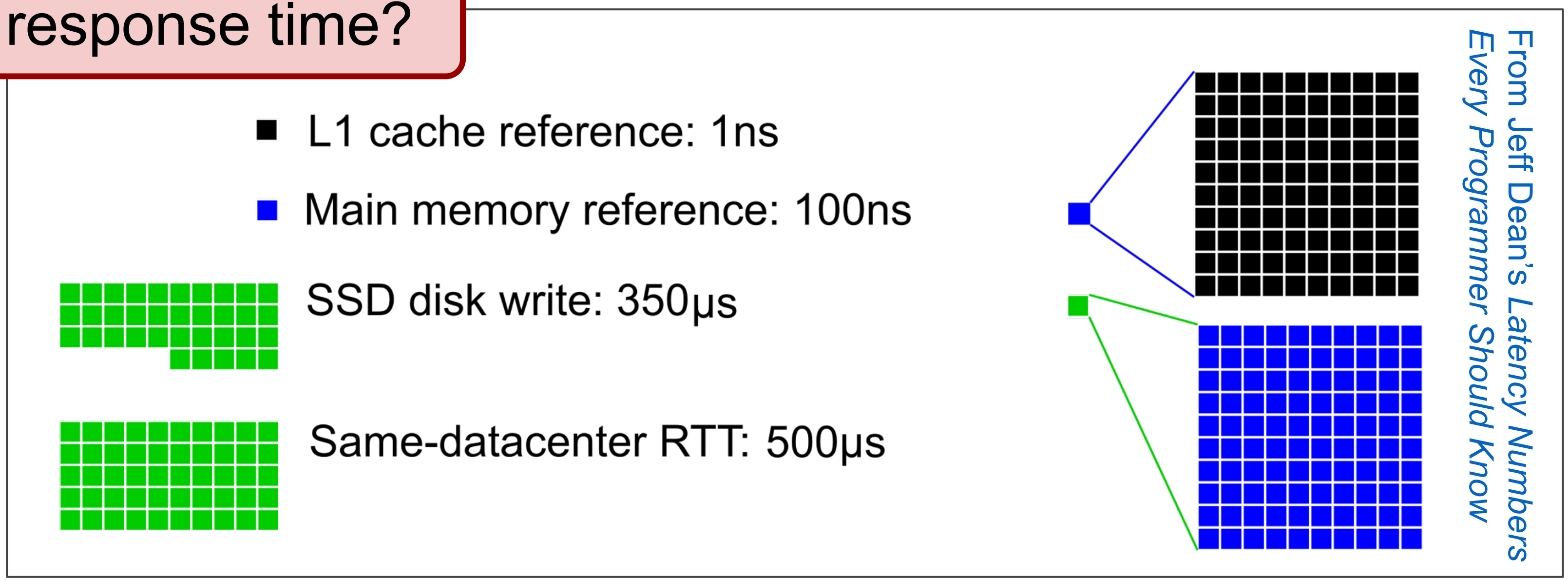
Can you think of limitations of the MapReduce framework?

Limitations of MapReduce I

Store input/output after every step on disk



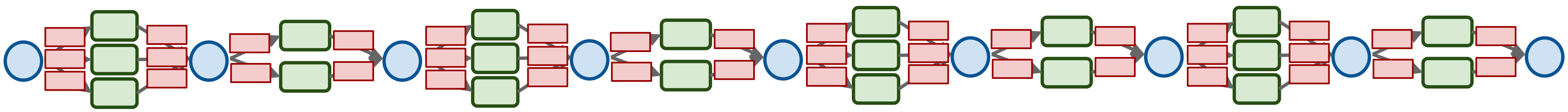
Effect on response time?



I/O penalty makes interactive data analysis impossible

Limitations of MapReduce II

Many applications require iterating MapReduce steps



Each iteration steps is small.

But: we need many iterations

⇒ 90% spent on I/O to disks and over network

⇒ 10% spent computing actual results

Does not work for iterative applications
(⇒ distributed machine learning)

Limitations of MapReduce III

MapReduce abstraction not expressive enough

Explosion of specialized analytics systems

- Streaming analytics, Iterative ML algorithms, Graph/social data

STORM
GraphLab
Google Dremel
Pregel
APACHE GIRAPH
APACHE kafka
S4: Distributed Stream Computing Platform

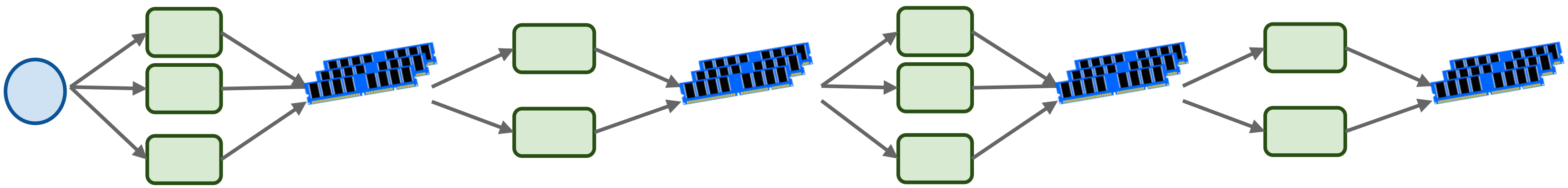
Learn all of them? Share data between them?

15-440/640
Distributed Systems

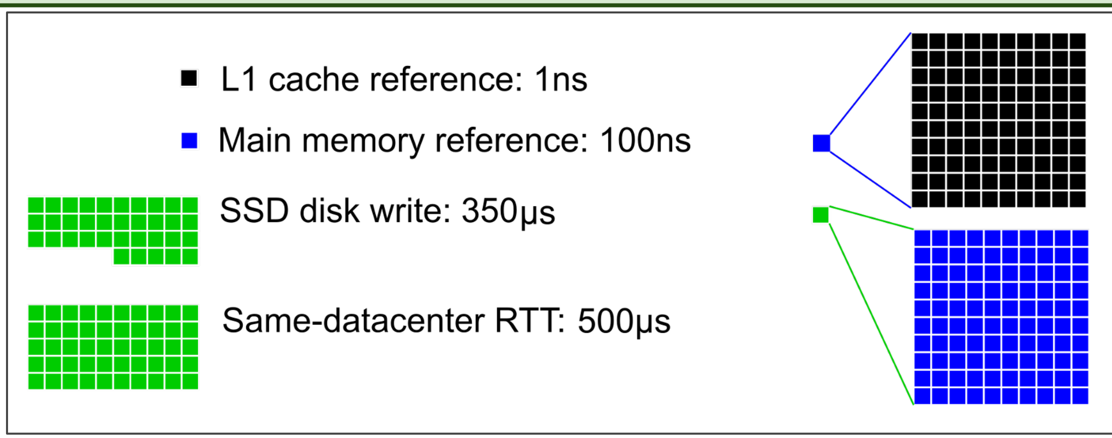
- Finish up distributed computation (MPI & MapReduce)
- In-memory cluster compute (Spark)
- Distributed ML

Apache Spark: In-Memory Computation

Key idea: keep and share data sets in main memory

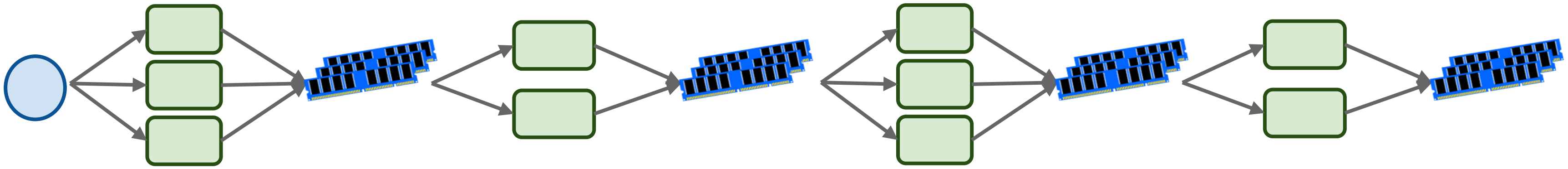


Much faster response time (in practice: 10x-100x)



Why didn't we do that in the first place? Problems?

In-memory computation and data-sharing



How to build **fault-tolerant** and **efficient** system?

Fault tolerance techniques from lectures so far?

- Logging each operation to node-local persistent storage
- Replicating data across nodes (+ persistent storage)
- Checkpointing (checkpoints need to be stored persistently)

⇒ Expensive (10-100x slowdown)

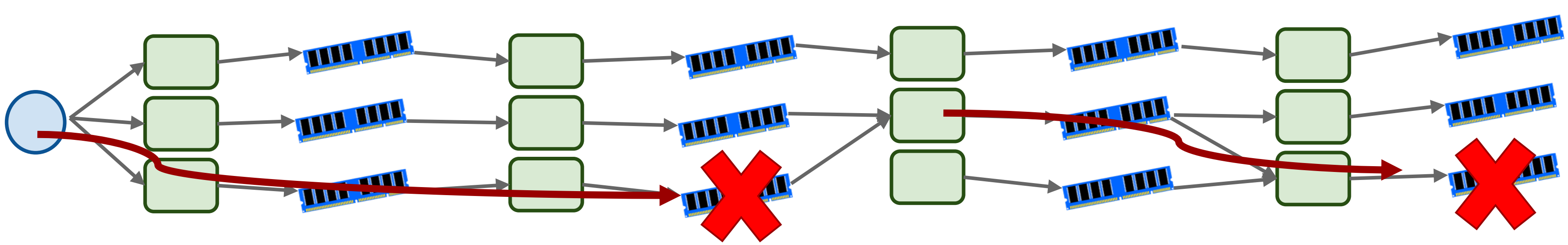
How common are fine-grained (bit-level) data updates?

Spark Approach: RDDs and Lineage

Zaharia et al. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. NSDI 2012.

Resilient **D**istributed **D**atasets

- Limit update interface to coarse-grained operations
  - Map, group-by, filter, sample, ...
- Efficient fault recovery using **lineage**
  - Small partition size → individual operations are cheap
  - Master server tracks coarse-grained operator sequence
  - Recompute lost partitions on failure

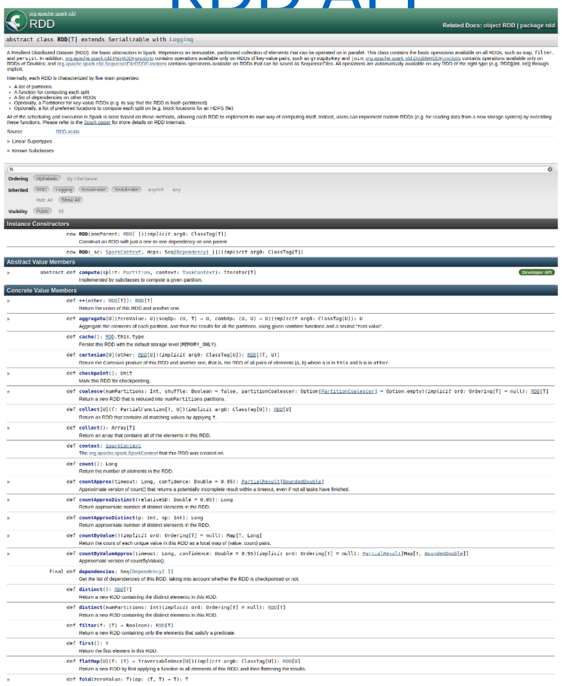


RDD Consistency and Fault Recovery

RDDs are **immutable** datasets

- Deterministic functions of input
  - recreate any RDD any time
- Simplifies consistency (caching, sharing, ..)
- Still need periodic RDD **checkpoints**
  - stored persistently on disks/ in HDFS

RDD API

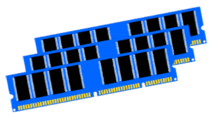


Design implications?

- High overhead: copying data (no mutate-in-place)
- Needs lots of memory (might not be able to run your workload)

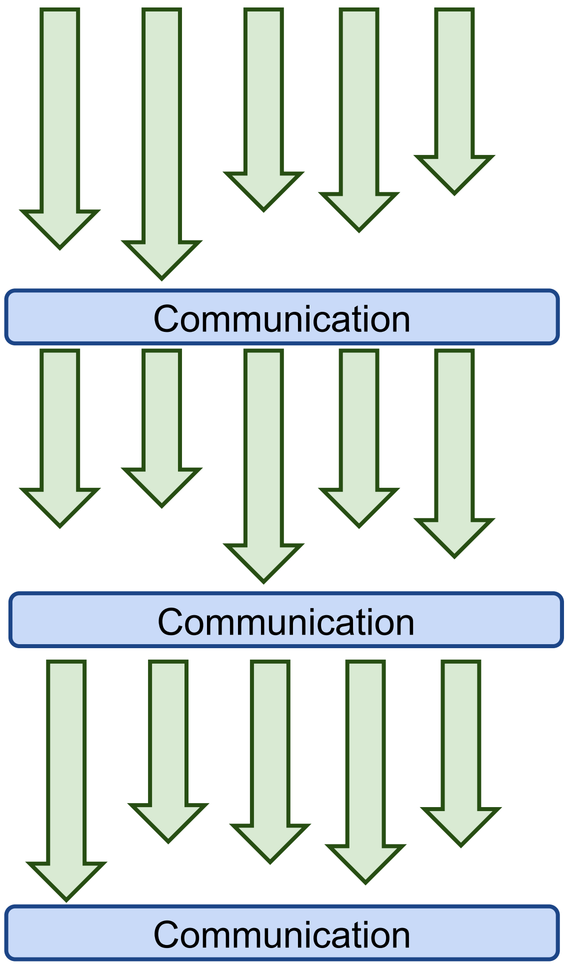
Towards a New Unified Framework

Two Goals

1. In-memory computation and data-sharing
   -  10-100x faster than disks or network
   - Key problem: fault tolerance
2. Unified computation abstraction
   - Power of iterations (“local work + message passing”)
   - Key problem: ease-of-use and generality

BSP computation abstraction

- Surprising power of iterations
  - (e.g., iterative Map/Reduce)
- Explained by theory of bulk synchronous parallel (BSP) model



<u>Theorem (Leslie Valiant,1990):</u>
"Any distributed system can be emulated as local work + message passing" (=BSP).

Spark implements BSP approximately

Spark as a Uniform Framework

Graph processing like

GraphLab/Pregel on Spark (Bagel)

⇒ “200 lines of Spark code”

Iterative MapReduce

⇒ “200 lines of Spark code“

Hive SQL on Spark (Shark)

⇒ “500 lines of code”

ML-lib and other distributed ML implementations

Should You Always Use Spark?

Some examples for which Spark is not a good fit for

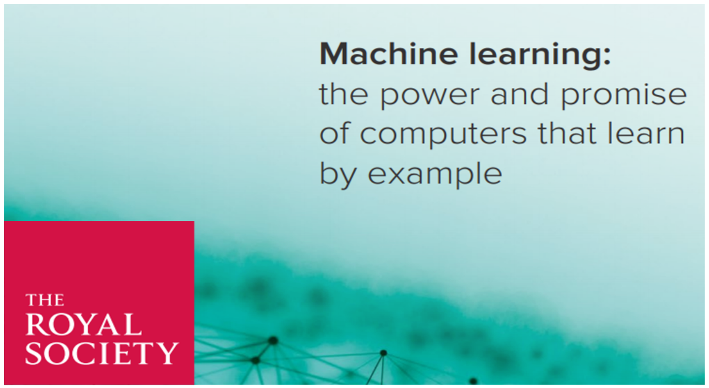
- Applications with fine-grained updates to shared state
- Datasets that don’t fit into memory

15-440/640
Distributed Systems

- Finish up distributed computation (MPI & MapReduce)
- In-memory cluster compute (Spark)
- Distributed ML

Machine Learning

The ML hype

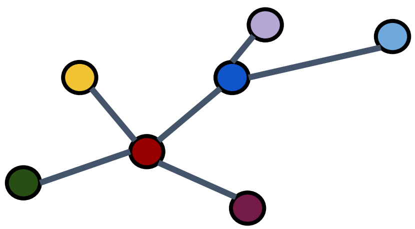


Enabled by huge leap in parallelization

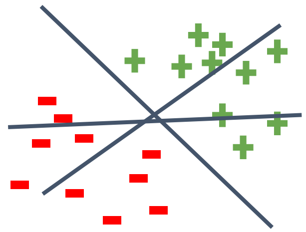
https://www.rockpapershotgun.com/2015/02/26/sli-good-or-bad/

ML systems out scale even powerful machines (GPUs et al)
=> Distributed ML

What Do ML Algorithms look like?



Page Rank



Regression

Other examples: Bayes, K-means, Neural Networks...

Common feature when computing these algorithms?

Eric Xing, Strategies & Principles for Distributed Machine Learning, Allen AI, 2016

Three key challenges:

1) lots of data

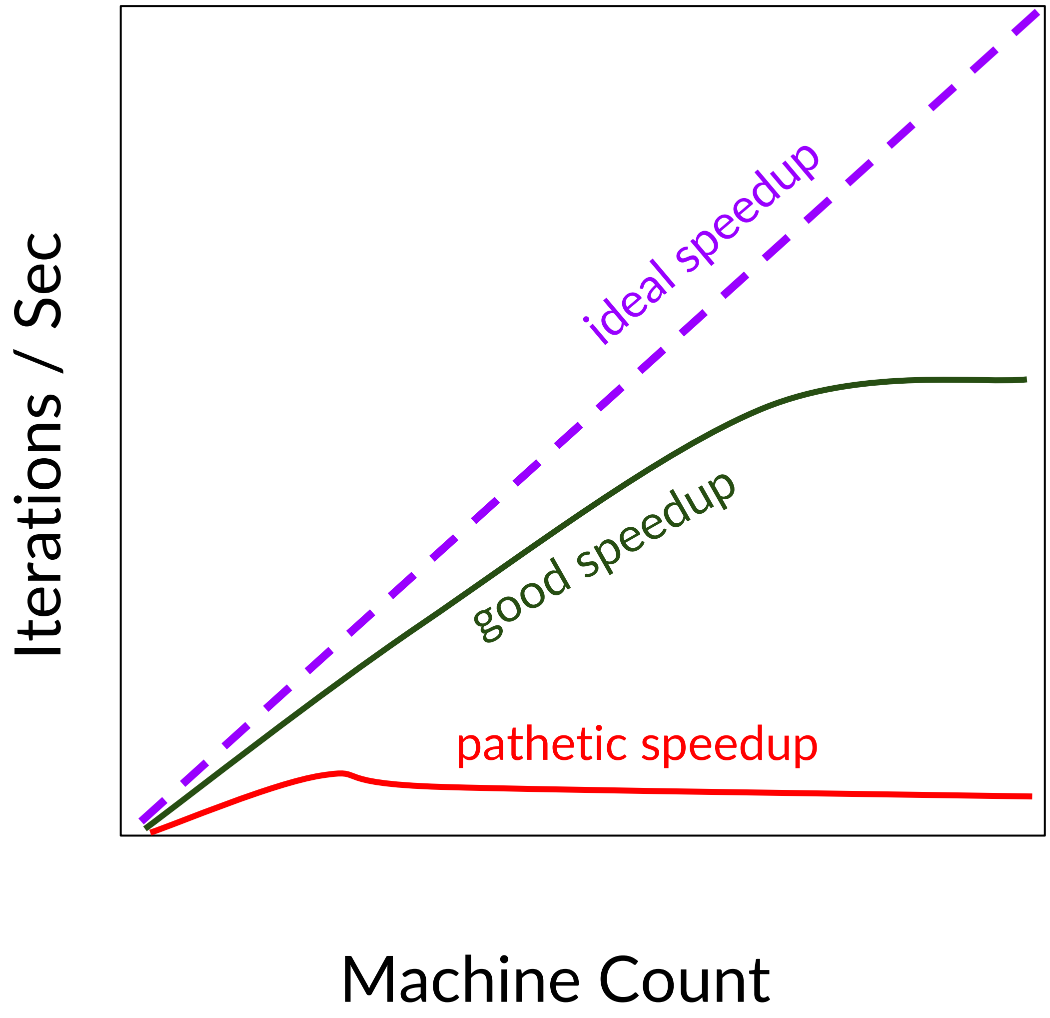
2) lots of parameters

3) lots of iterations

Distributed Machine Learning

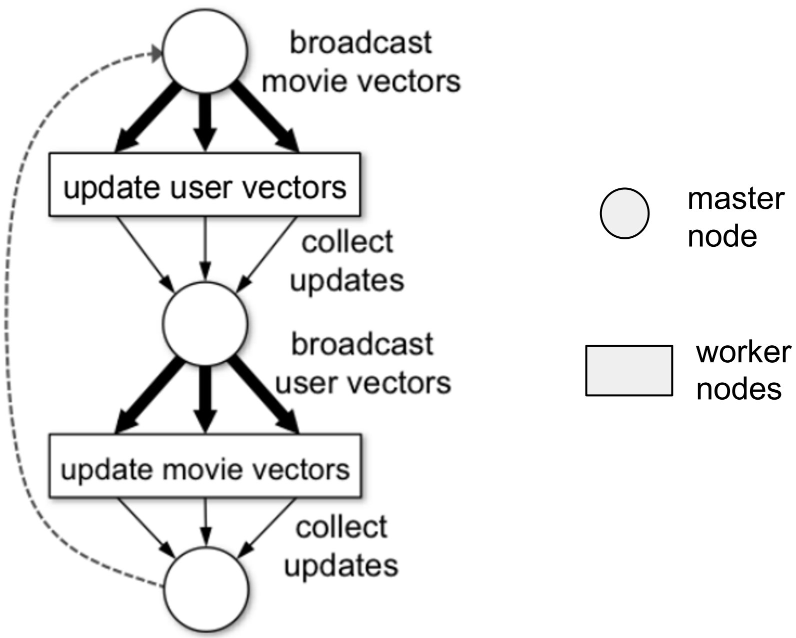
Data/model often fits into 10s-100s of nodes

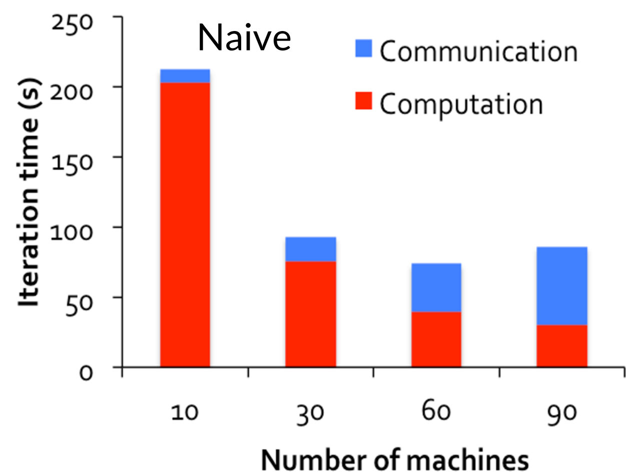
Goal: more iterations / sec



Challenge of Communication Overhead

- Communication overhead scales badly
- E.g., for Netflix-like recommender systems





From: Chowdhury, et al., Managing data transfers in computer clusters with orchestra., SIGCOMM 2011.

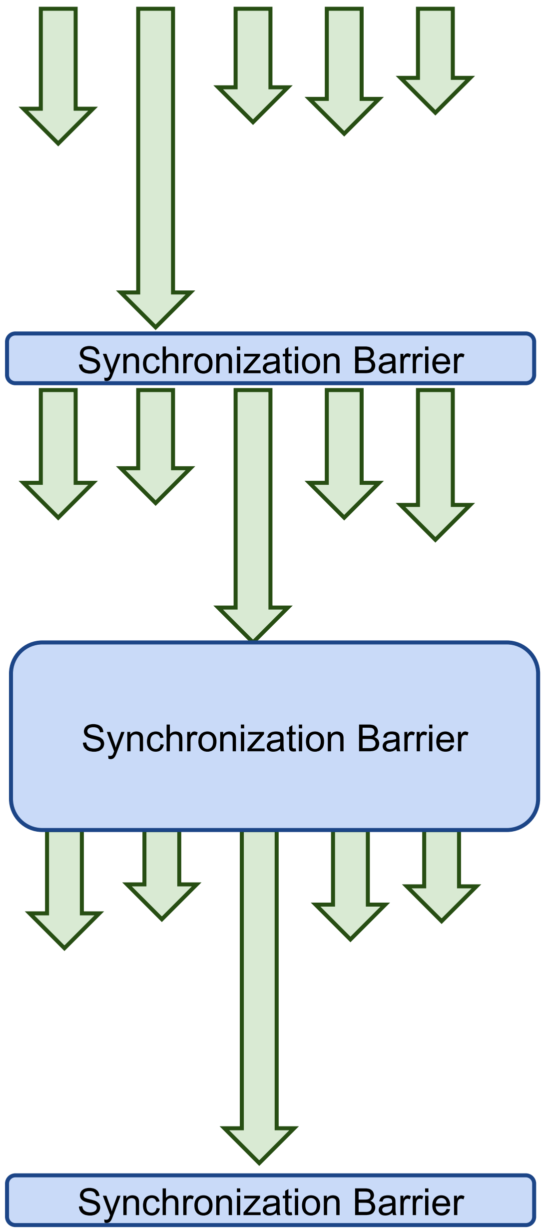
Challenge of Synchronization Overhead

BSP model:

- No computation during barrier
- No communication during computation

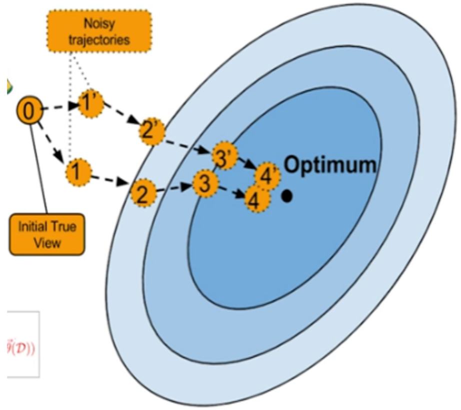
Fundamental limitation in BSP model

Constantly waiting for **stragglers**

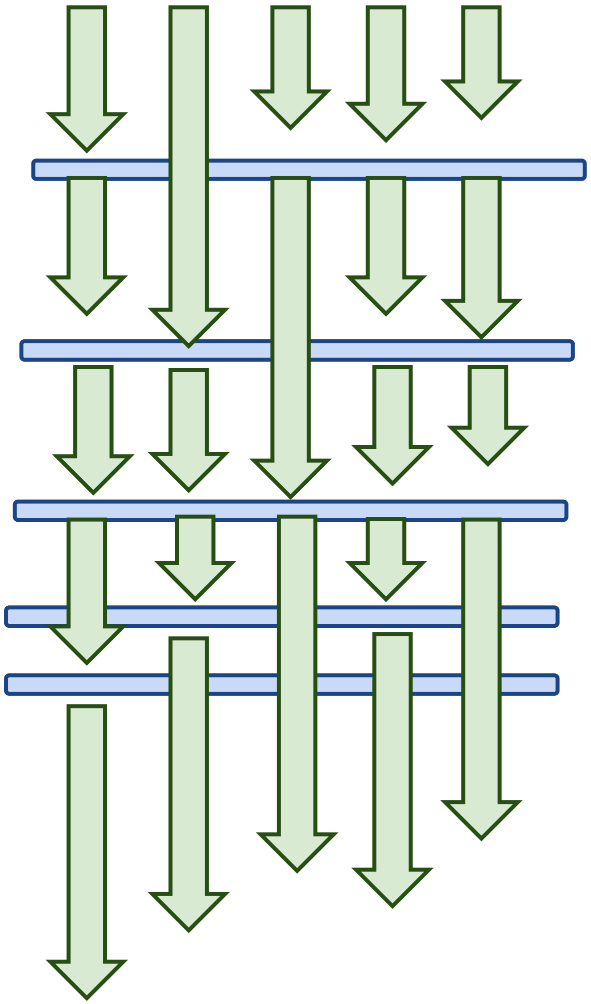


Relaxing BSP Consistency

Idea: nodes can accept slightly stale state



From: Eric Xing, Strategies & Principles for Distributed Machine Learning, Allen AI, 2016



ML algorithms are robust

⇒ converge even with some stale state

How can we incorporate stale state into the BSP model?

Opposite Extreme: No Synchronization

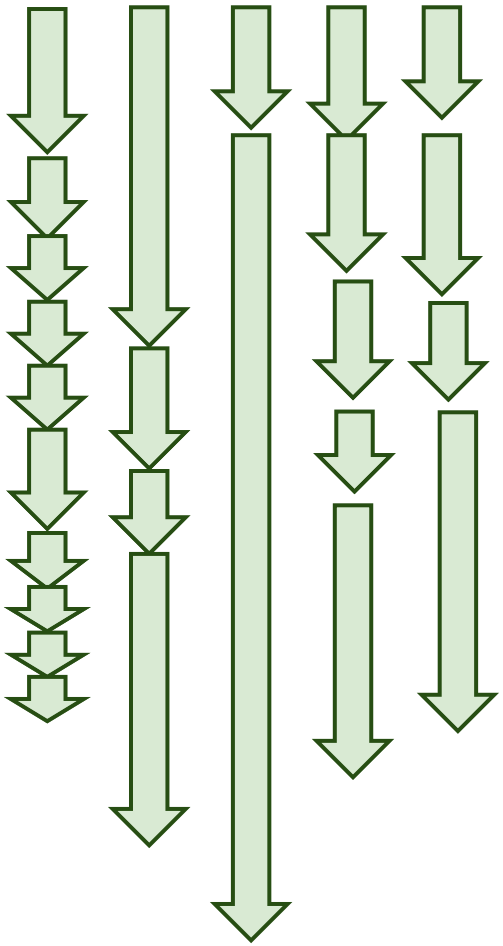
What if we fully remove BSP's synchronization barriers?

Asynchronous communication:

- no communication at all, or
- communication at any time

Observation through experiments:

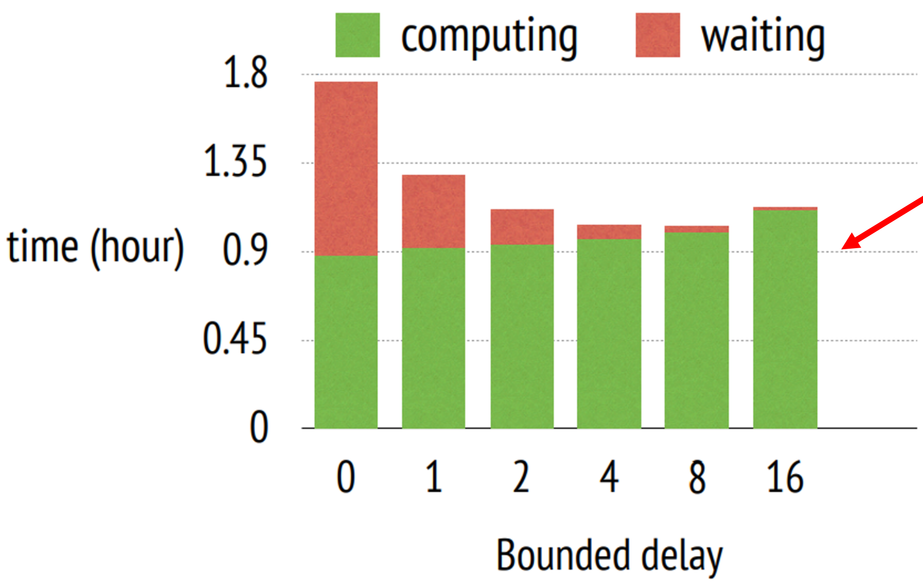
Iterative algorithms won't converge



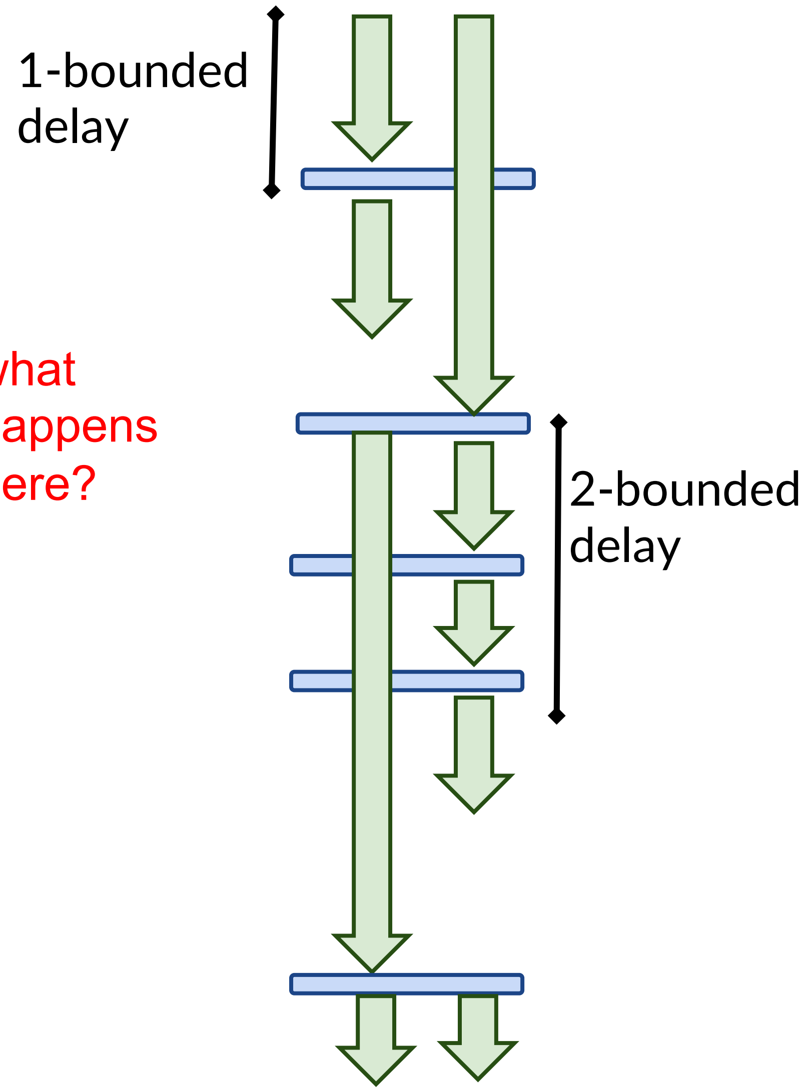
Bounded-delay BSP for Distributed ML

Bound stale state by N steps:

⇒ N-bounded delay BSP



what happens here?



From: Li et al, Scaling Distributed Machine Learning with the Parameter Server OSDI 2014

Many Challenges Remain

Trade-Off:

Stale state -> throughput (iter / sec)

Misleading design decisions:

Higher throughput

Less progress / iteration

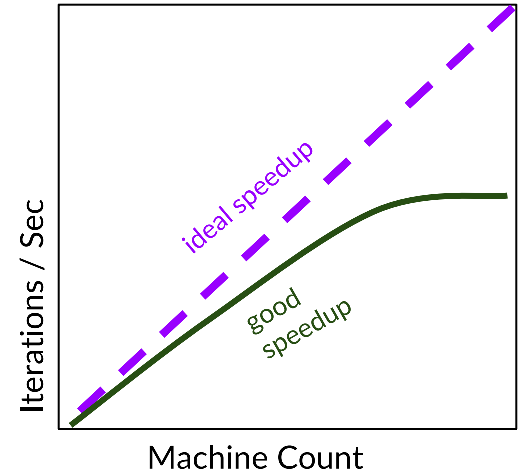
Many open challenges

Automatic model partitioning

How to schedule many parallel jobs on ML clusters

How to build a framework for interactive ML applications

⇒ Very active field of research



PETUUM

October 12, 2017

CMU Spinoff Petuum Receives $93M in New Round of Funding