15-440/640
Distributed Systems

- Finish up cluster filesystems (GFS)
- Start distributed computation (HPC & cluster computing)

Announcements

- **Fill in P1 project partner survey**
- 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.

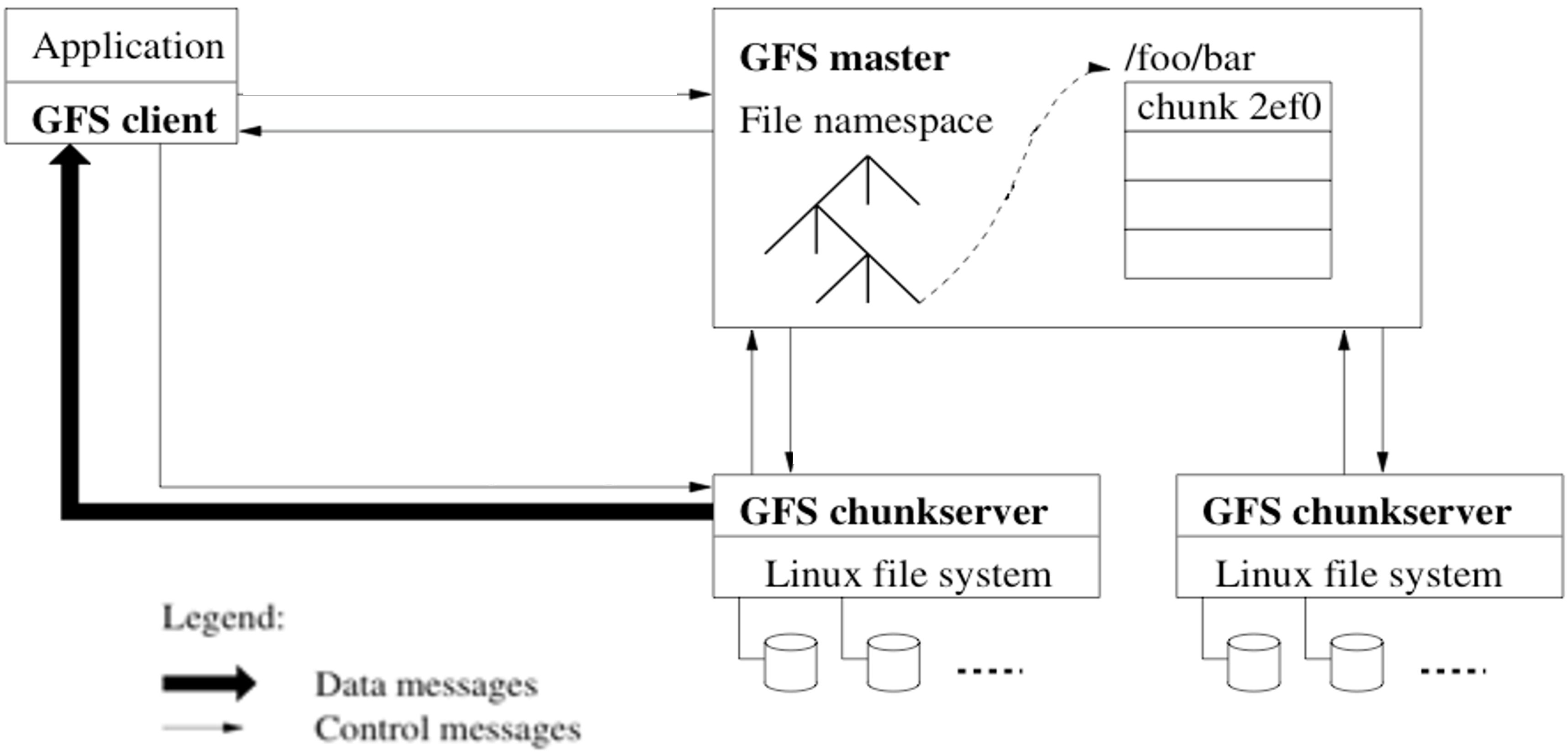
15-440/640 Distributed Systems

- Finish up cluster filesystems (GFS)
- Start distributed computation (HPC & cluster computing)

Outline: GFS

- Motivation and design goals
- Architecture
- Client operations
- Fault tolerance
- Consistency model
- Post-GFS

Recall: High-Level Picture of GFS Architecture



GFS Client: Record Append Operation

- Large files used as queues between multiple producers and consumers
  - Need atomic append operation

Why not use a regular GFS write (client, offset)?

$\Rightarrow$ multiple clients might use GFS write (client offset) operation to write records to the same region
$\Rightarrow$ Avoid using complex and expensive synchronization among clients (e.g., distributed lock manager)

- Client pushes data to last chunk's replicas; sends append request to primary without specifying byte offset

GFS Client: Record Append Operation

- Common case: request fits in last chunk
  - Primary appends data to own chunk replica
  - **Primary tells secondaries to do same at same byte offset in their chunk replicas**
  - Primary replies with success to client
- When data won't fit in last chunk
  - Primary fills current chunk with padding
  - Primary instructs other replicas to do same
  - Primary replies to client, "retry on next chunk"
- If record append fails at any replica, client retries

GFS Client: Record Append Operation

What guarantee does GFS provide after reporting success of append to application?

- Replicas of same chunk may contain different data
  - Can contain duplicates of all or part of record data
  - Some regions of a chunk consistent and some not

- Semantics?
- Data written **at least once** in atomic unit

  ⇒ GFS client retries until success

Outline: GFS

- Motivation and design goals
- Architecture
- Client Operations
- Fault tolerance
- Consistency model
- Post-GFS

GFS Fault Tolerance

High Availability

- Chunk replication
  - Each chunk is replicated on multiple chunkservers

- Master (i.e., state of the master) replication
  - Operation log and checkpoints replicated on multiple machines

Data Integrity

- Checksum checks
  - Each chunk has checksums
  - Checksum verified for every read and write
  - Checksum also verified periodically for inactive chunks

GFS Fault Tolerance: Chunkserver

Chunkservers can be temporarily down or fail

Insufficient chunk replicas

- Master notices missing heartbeats
- Master decrements count of replicas for all chunks on dead chunkserver
- Master re-replicates chunks missing replicas in background

Stale chunks

- Chunks have version numbers
  - Stored on disk at master and chunkservers
  - Each time master grants new lease to primary, increments version, informs all replicas
- Detect outdated chunks with version number
  - Outdated chunks are ignored and garbage collected

GFS Fault Tolerance: Master

What if GFS loses the master?

- Master has all metadata information
  - Lose master = lose the filesystem
- Master logs metadata updates to disk sequentially (→ WAL)
- Replicates log entries to remote backup servers
- Only replies to client after log entries safe on disk on self and backups

GFS Fault Tolerance: Master

- Replays log from disk
  - Recovers namespace (directory) and file-to-chunk-ID mapping (but not location of chunks)
- Asks chunkservers which chunks they hold
  - Recovers chunk-ID-to-chunkserver mapping
- If chunk server has newer chunk, adopt its version number
  - Master may have failed while granting lease

- Logs cannot be too long – why?
  - Master uses log to rebuild the filesystem state at startup
- How to avoid too long logs?
  - Periodic checkpoints taken to keep log short

Outline: GFS

- Motivation and design goals
- Architecture
- Client Operations
- Fault tolerance
- Consistency model
- Post-GFS

GFS Consistency Model

- Changes to namespace (i.e., metadata) are atomic
  - E.g., file creation
  - Due to: namespace locking (granular) + operation log

- Changes to data are ordered by a primary
  - Concurrent writes can be overwritten
  - Record appends complete at least once, at offset of GFS's choosing

    → Applications must cope with possible duplicates

GFS Consistency Model

- Failed operations can cause inconsistency
  - E.g., different data across chunk servers (failed append)
- Concurrent successful writes (to the same region) results in an "undefined" region
- Behavior is worse for writes than appends (why?)

GFS applications designed to accommodate the relaxed consistency model

- Co-design of applications and the file system

Outline: GFS

- Motivation and design goals
- Architecture
- Client Operations
- Fault tolerance
- Consistency model
- Post-GFS

Post GFS

Open-source Implementation:

- Apache Hadoop Distributed File System (HDFS)
- Widely deployed in industry (esp. as underlying filesystem for data analytics clusters)

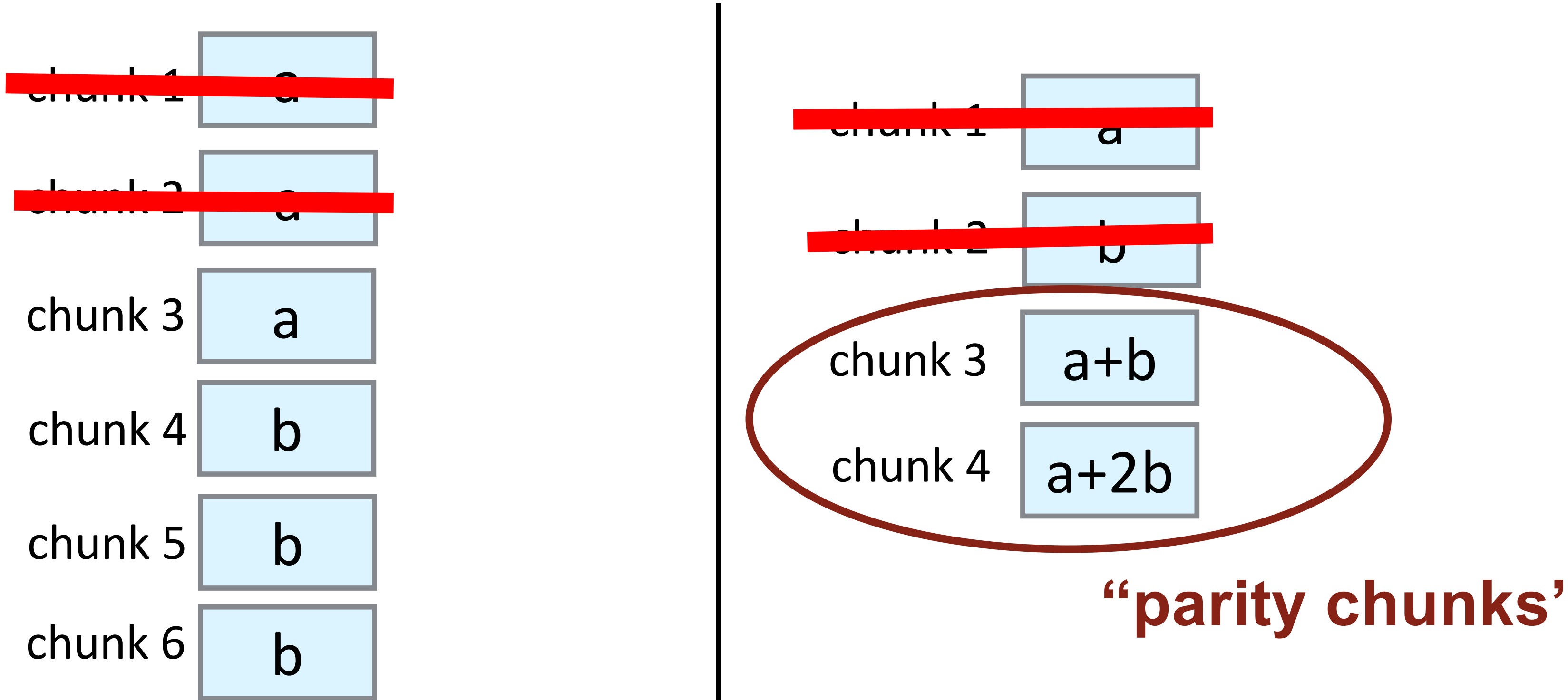
Successor at Google: Colossus

- Some of the key differences
  - Eliminates master node as single point of failure: Multiple/distributed masters
  - Improved storage efficiency: Employs **erasure coding** instead of replicas

Replication vs. erasure codes

Two data chunks to be stored: a and b

Tolerate any 2 failures



3-replication

Storage overhead = 3x

Erasure code

Storage overhead = 2x

Replication vs. erasure codes

Two data chunks to be stored: a and b

Tolerate any 2 failures

~~chunk 1 a~~

Erasure codes: much less storage for desired fault tolerance

chunk 5 b

chunk 6 b

"parity chunks"

3-replication

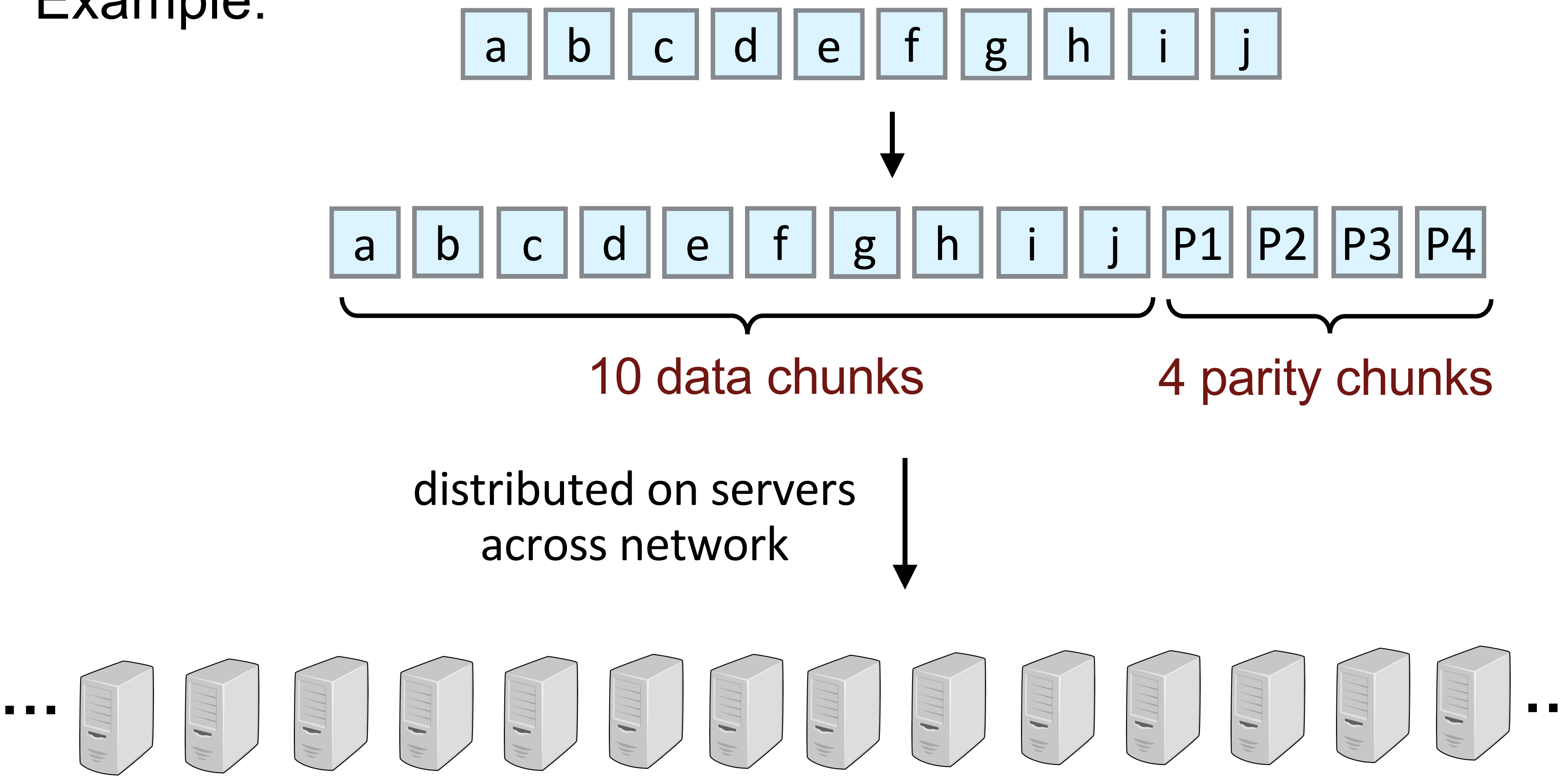
Storage overhead = 3x

Erasure code

Storage overhead = 2x

Erasure codes: how are they used in distributed storage systems?

Example:



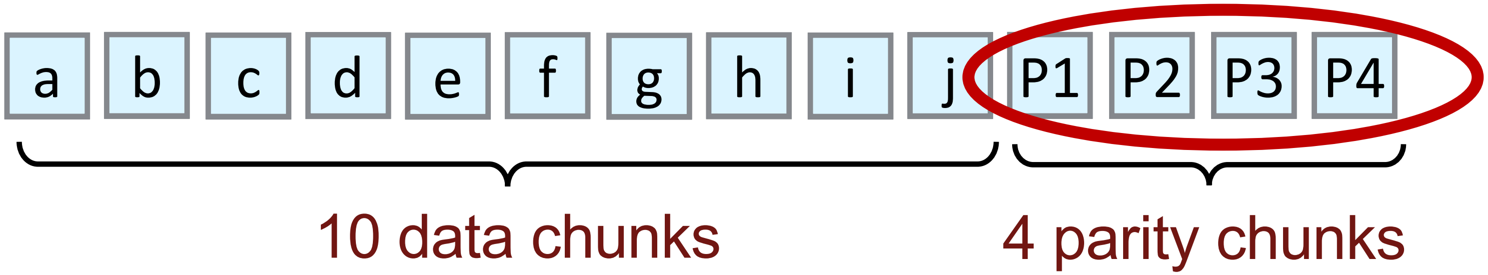
Most large-scale storage systems use erasure codes

Facebook, Google, Amazon, Microsoft...

"Considering trends in data growth & datacenter hardware, we foresee HDFS **erasure coding** being an **important feature in years to come**"

- Cloudera Engineering (September, 2016)

Research on erasure codes for storage clusters



Mathematical structure of parities decide degree of reliability and overhead

- Traditional erasure code: Reed-Solomon code
- Recent research on erasure codes for distributed storage
  - Apache Hadoop Distributed File System (HDFS) v3.0
    - "A Piggybacking Design Framework for Read-and Download-efficient Distributed Storage Codes", IEEE ISIT 2013, IEEE Transactions on Information Theory, 2017.
    - "A "Hitchhiker's" Guide to Fast and Efficient Data Reconstruction in Erasure-coded Data Centers", ACM SIGCOMM 2014.
  - Microsoft Azure
    - "Erasure Coding in Windows Azure Storage", USENIX ATC, 2012.
    - “On the locality of codeword symbols”, Transactions on Information Theory, 2012.

15-440/640 Distributed Systems

- Finish up cluster filesystems (GFS)
- Start distributed computation (MPI & MapReduce)

Cluster Computing

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

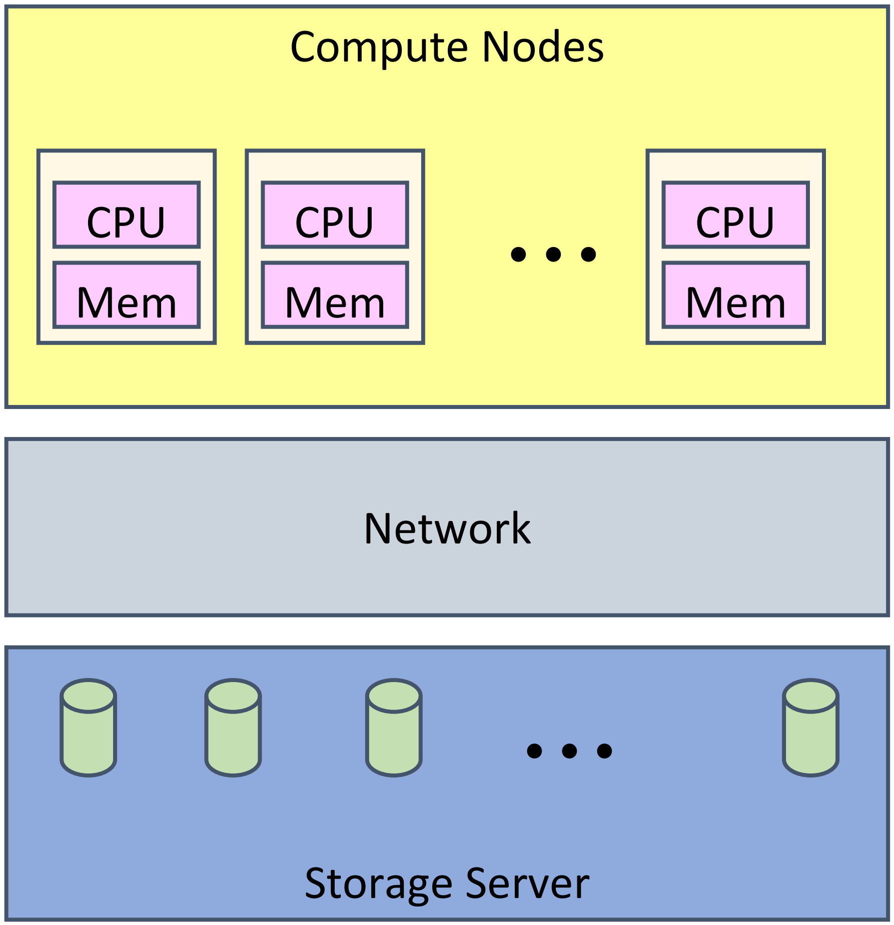
2. Cluster computing
   - MapReduce

Cluster Computing

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

2. Cluster computing
   - MapReduce

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

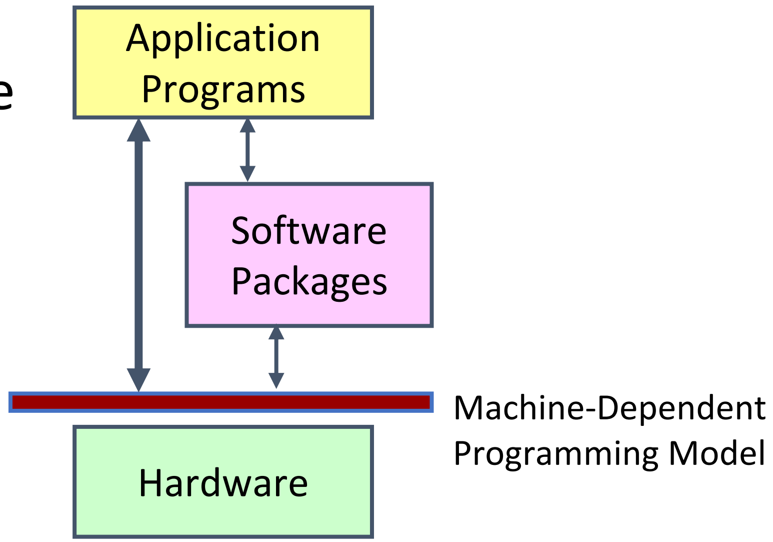
HPC Machine Example

SUMMIT
Supercomputer

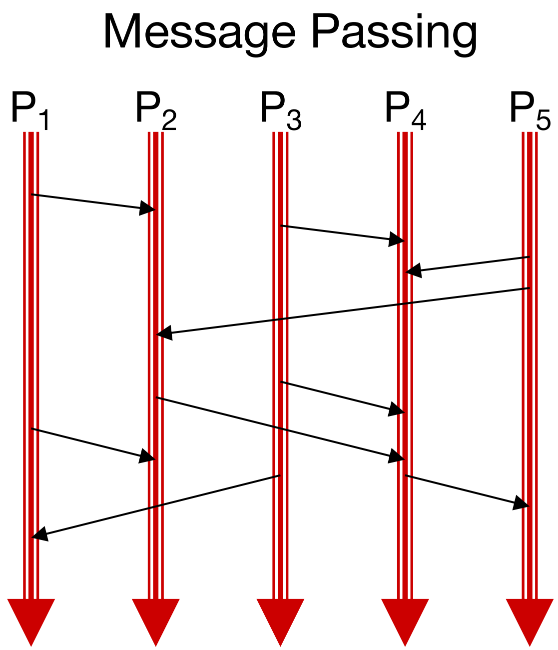
- Cores: ~200K CPU cores and ~27K GPU cores
- Total system memory: > 10 PB
- Interconnect: Mellanox EDR 100G InfiniBand

HPC Programming Model

- Message passing model
  - Processes communicate and synchronize via exchange of messages
- Programs described at very low level
  - Specify detailed control of processing & communications
- Rely on small number of software packages
  - Written by specialists
  - Limits classes of problems & solution methods



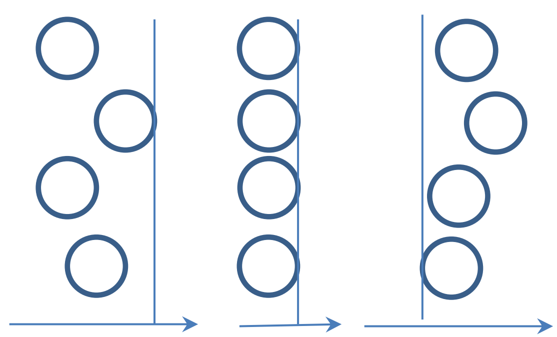
Typical HPC Operation



- Characteristics
  - Long-lived interdependent processes
  - Partitioning: exploit spatial locality
  - Hold all program data in memory (avoiding disk access)
  - High bandwidth communication

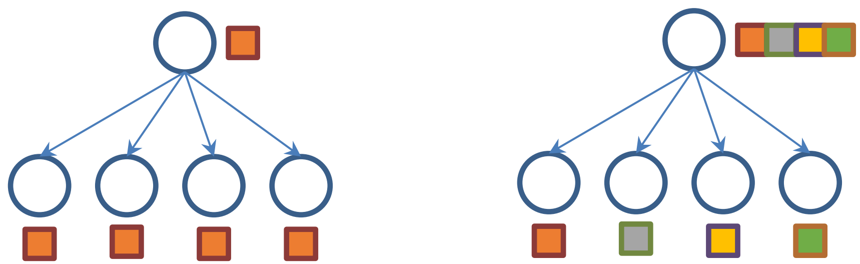
Message Passing Interface (MPI)

- Standardized communication protocol for programming parallel computers
- Specifies a range of functionality
  - Virtual topology, Synchronization, Communication
- Virtual topology
  - Finding number of processes, processor identity for a process, neighboring processes in a logical topology
- Synchronization: barrier

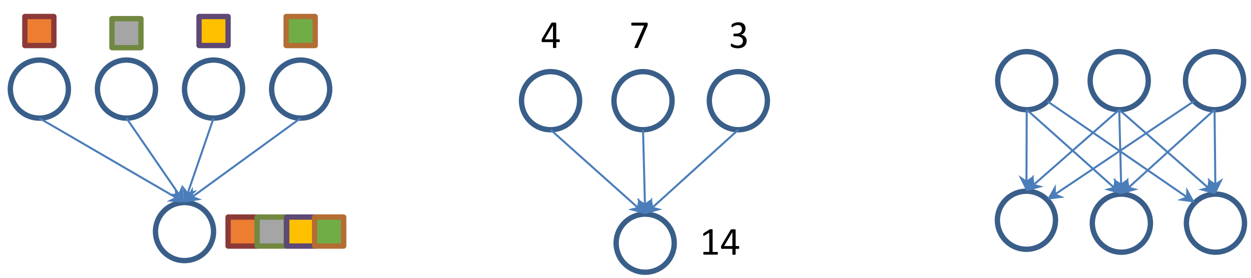


Message Passing Interface (MPI)

- Communication: both point-to-point and collective
  - Collective sending: E.g., broadcast, scatter



  - Collective receiving: E.g., gather, reduce, all-to-all

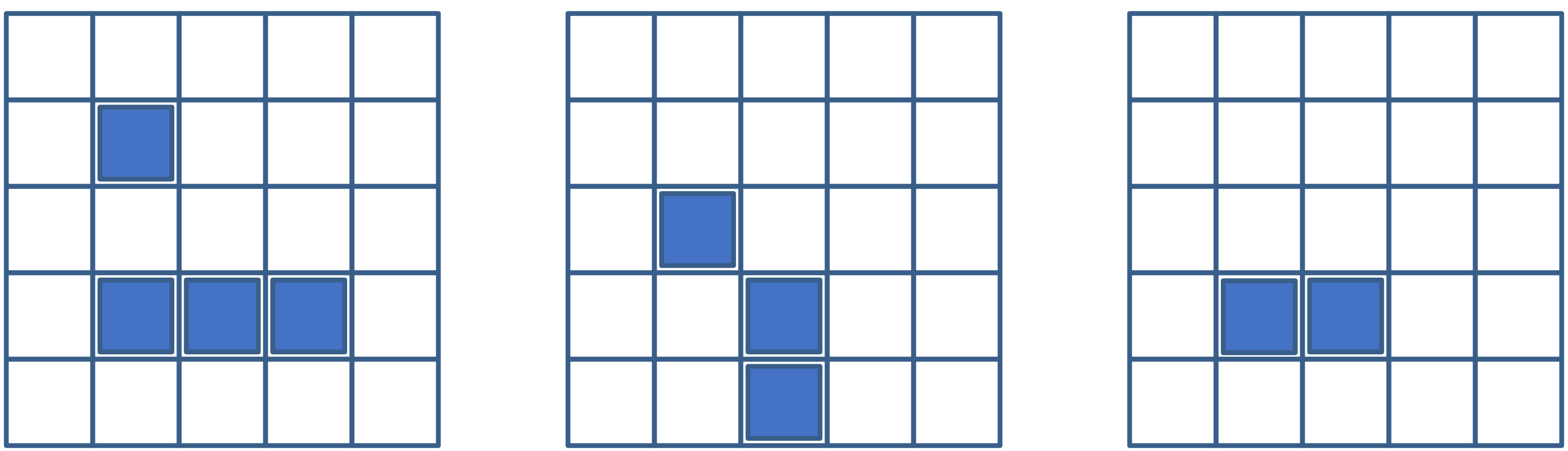


Involves both sending and receiving

- MPI implementations highly optimized for low latency, high scalability over HPC grids / LANs

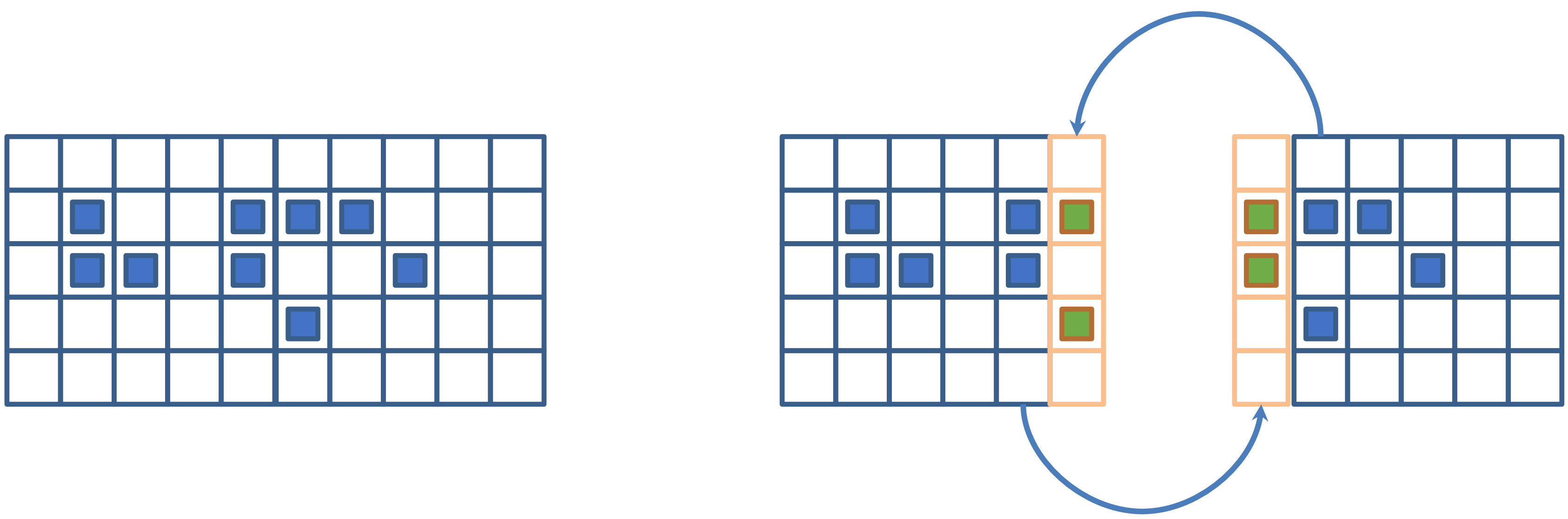
HPC Example: Iterative Simulation I

- Conway's Game of Life
  - Cellular automata on a square grid
  - Each cell "live" or "dead" (empty)
  - State in next "generation" depends on number of current neighbors:
    - 2 -> stays same
    - 3 -> becomes live
    - Other -> becomes empty

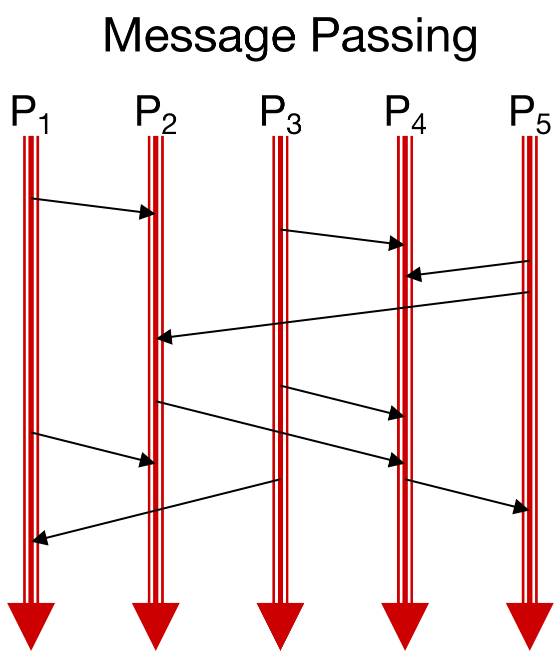


HPC Example: Iterative Simulation II

- Shard grid across nodes
- Simulate locally in each subgrid
- Exchange boundary information
- Repeat simulation, exchange steps

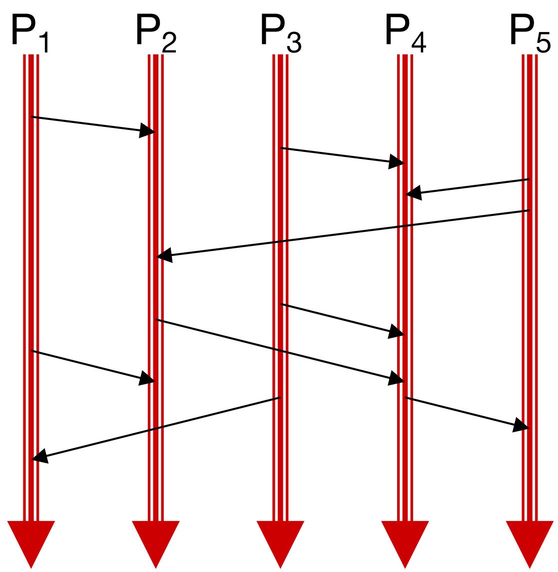


Typical HPC Operation



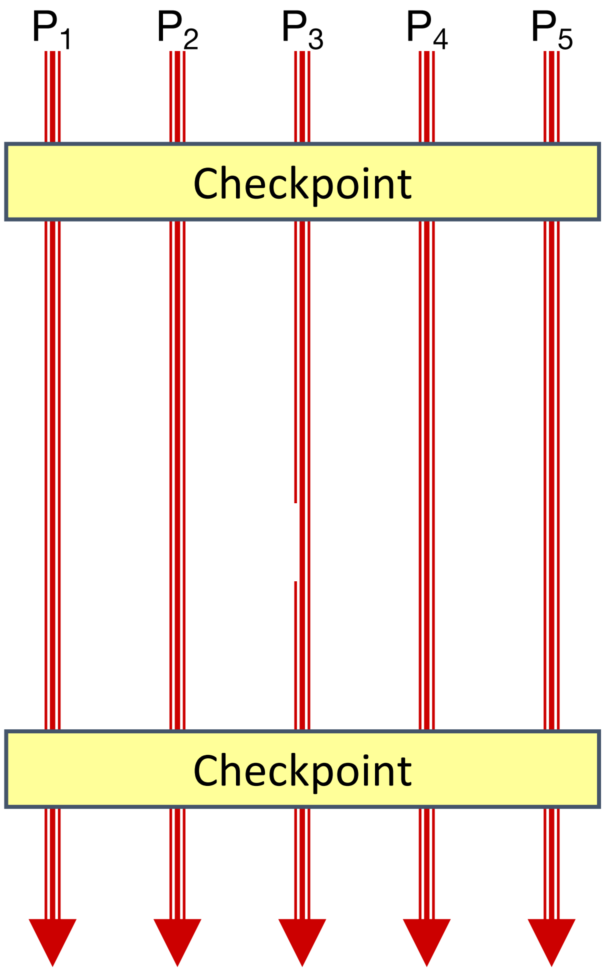
- Characteristics
  - Long-lived interdependent processes
  - Partitioning: exploit spatial locality
  - Hold all program data in memory (no disk access)
  - High bandwidth communication
- Strengths
  - High utilization of resources
  - Effective for many scientific applications
- Weaknesses
  - Requires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

HPC Fault Tolerance



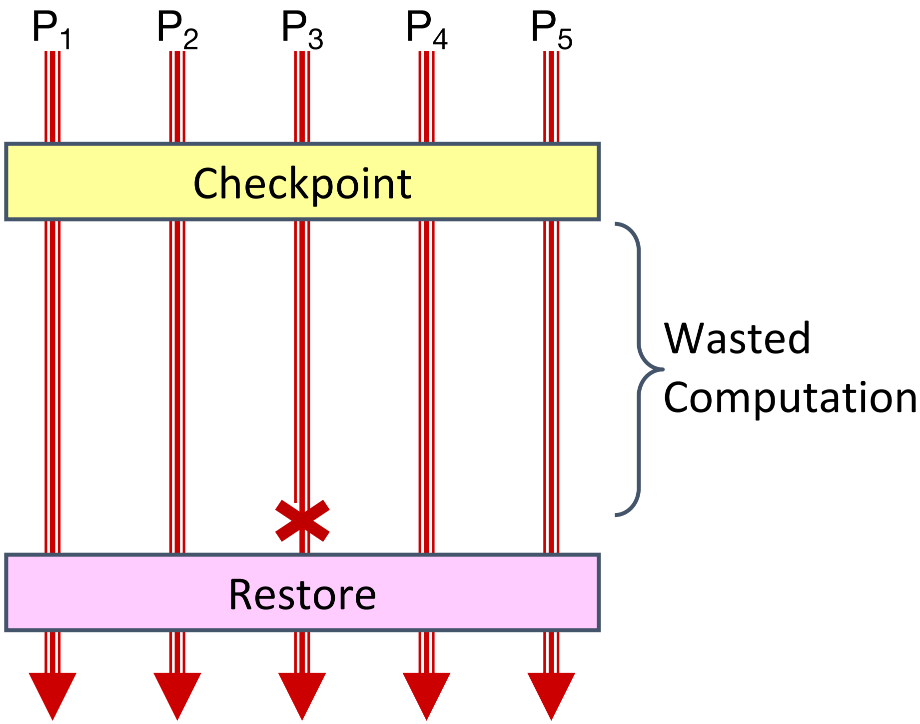
- Tightly coupled processes
  - Failure of one processes prevents all others from progressing
- How to ensure correct execution in presence of failures?

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



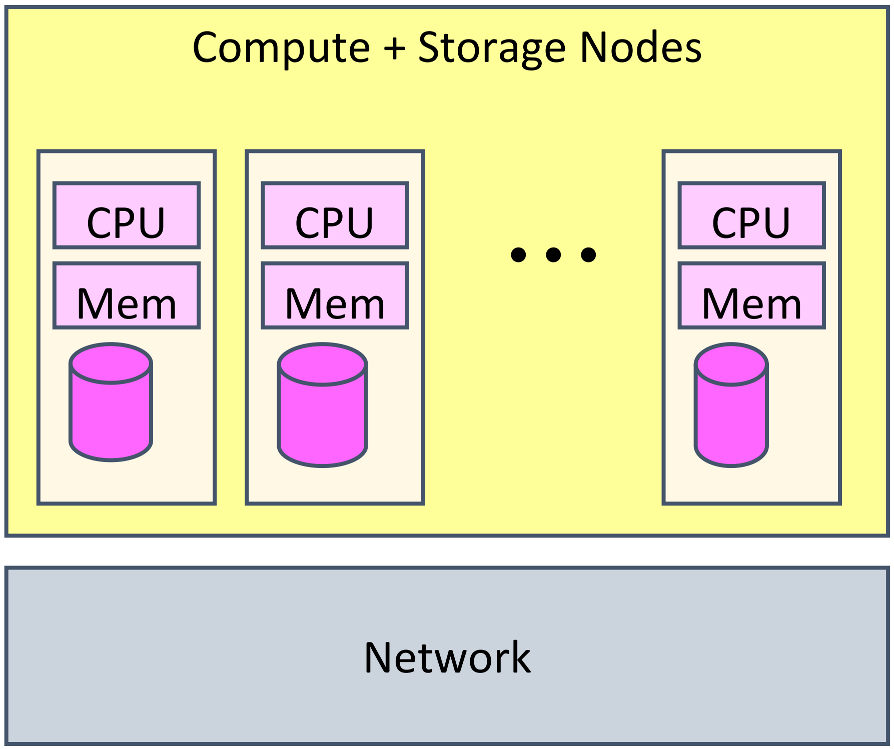
- Rollback upon failure
  - Restore state to that of last checkpoint
  - All intervening computation wasted
- Design decisions
  - Asynchronous or synchronous?
  - How often to checkpoint?
  - What data to checkpoint?
  - Who checkpoints: application or system?
- Significant I/O traffic
- Very sensitive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

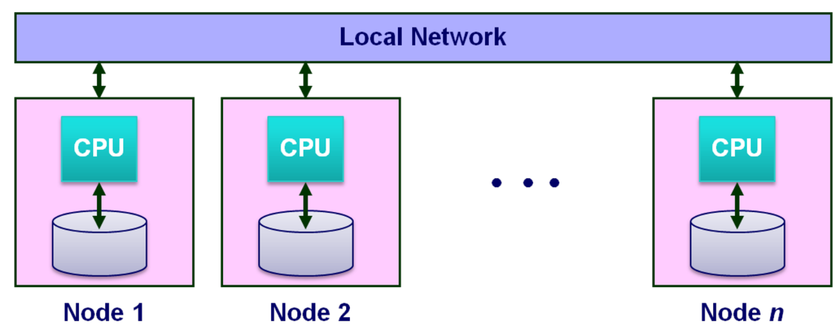
Disks	MB / s	Time
Seagate HDDs	~100s	> Few hours
Networks	**MB / s**	**Time**
Gigabit Ethernet	< 125	> 23 hours
10GE	< 1,200	> 2.4 hours
100GE	< 12,000	15 minutes

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

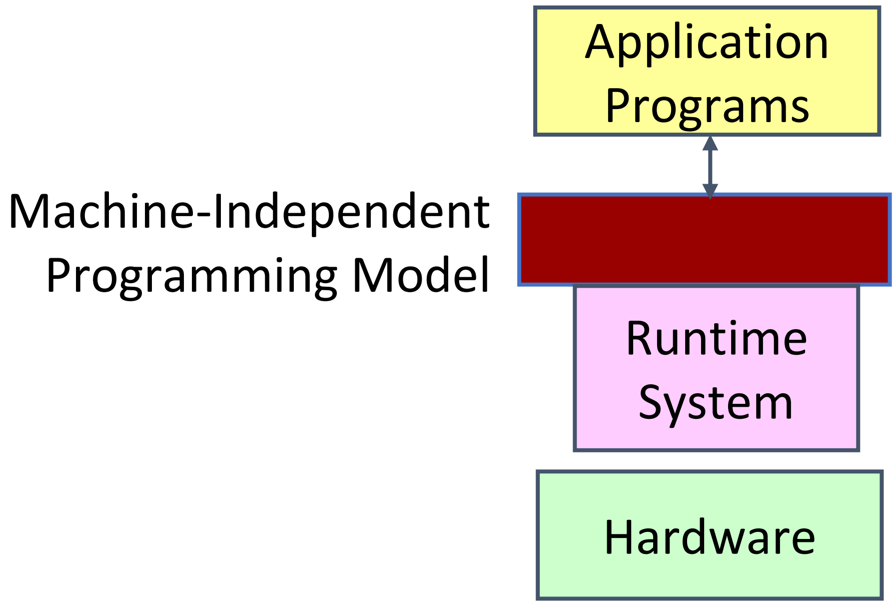
How do you find the frequency of words, such as , “440”, “error”, “p4” ?

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

e.g., machines fail, get rebooted

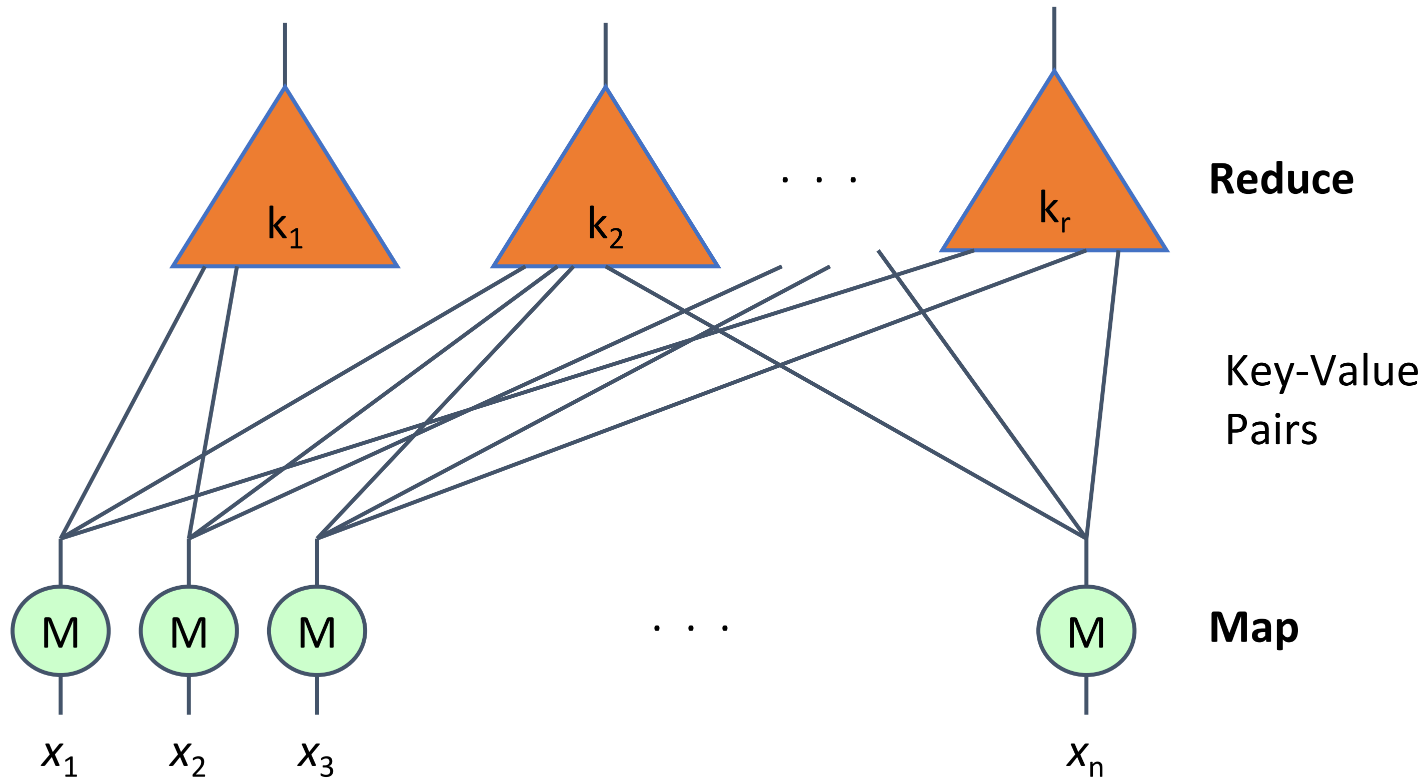
What do you do if a variant of this task comes up?

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

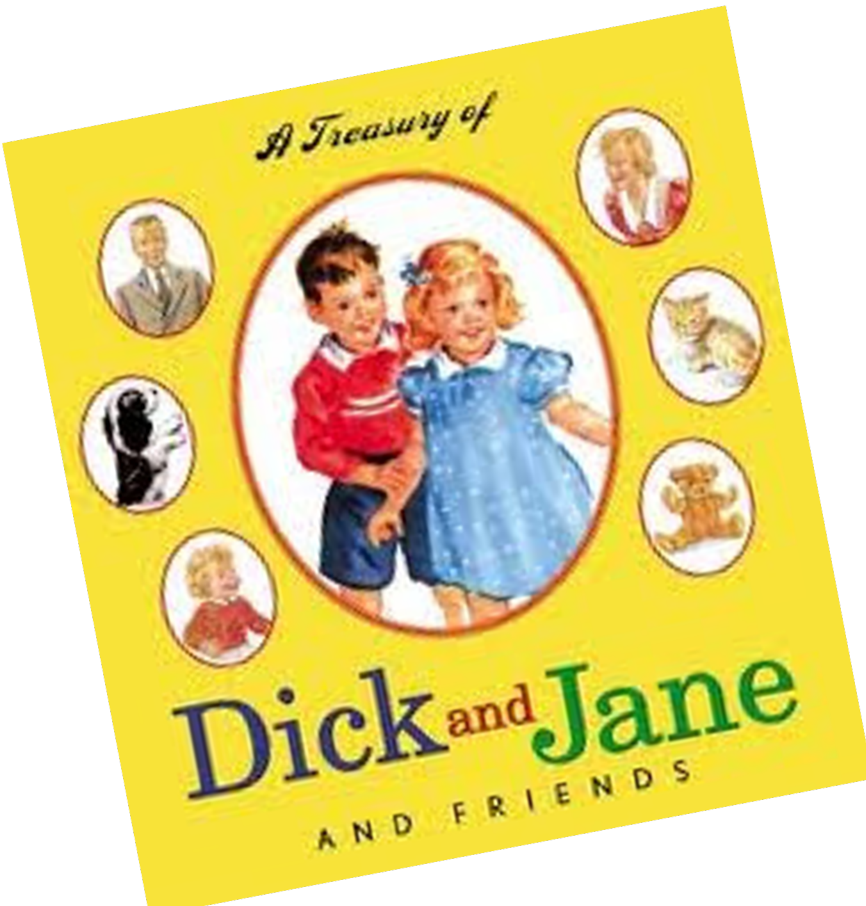


- Map: Map computation across many objects
  - Runtime schedules "mappers" so as to minimize data movement
- Reduce: Aggregation of results

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

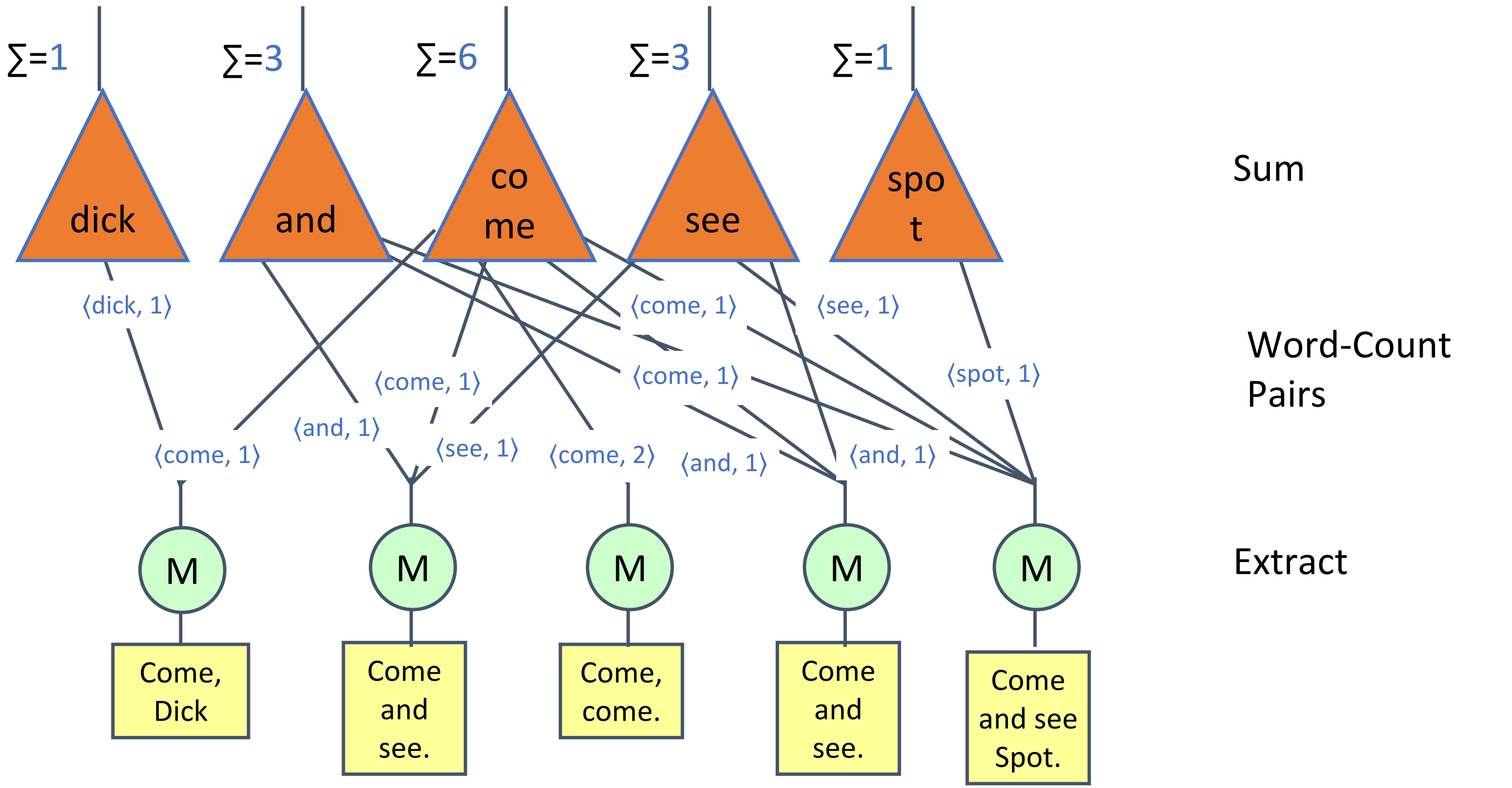


Example MapReduce

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

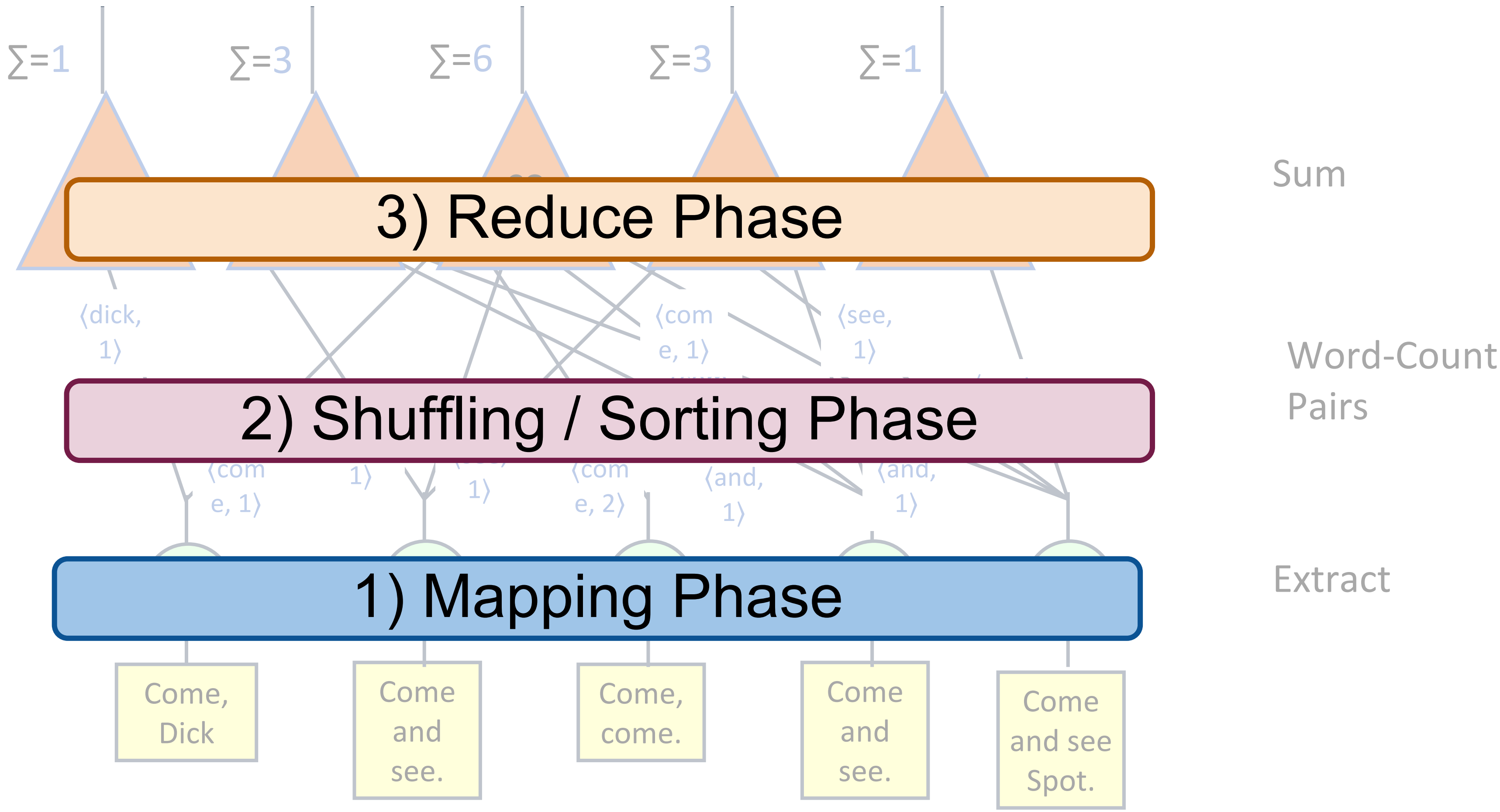
- Calculate word frequency of set of documents

Example MapReduce



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

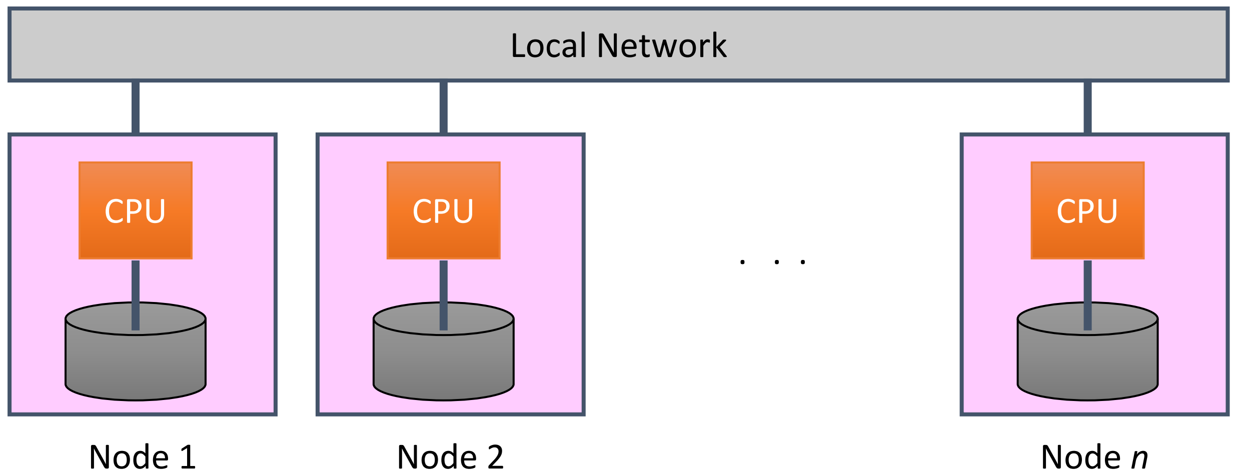
Example MapReduce



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

Hadoop Project

- Colocate compute and storage: HDFS + MapReduce

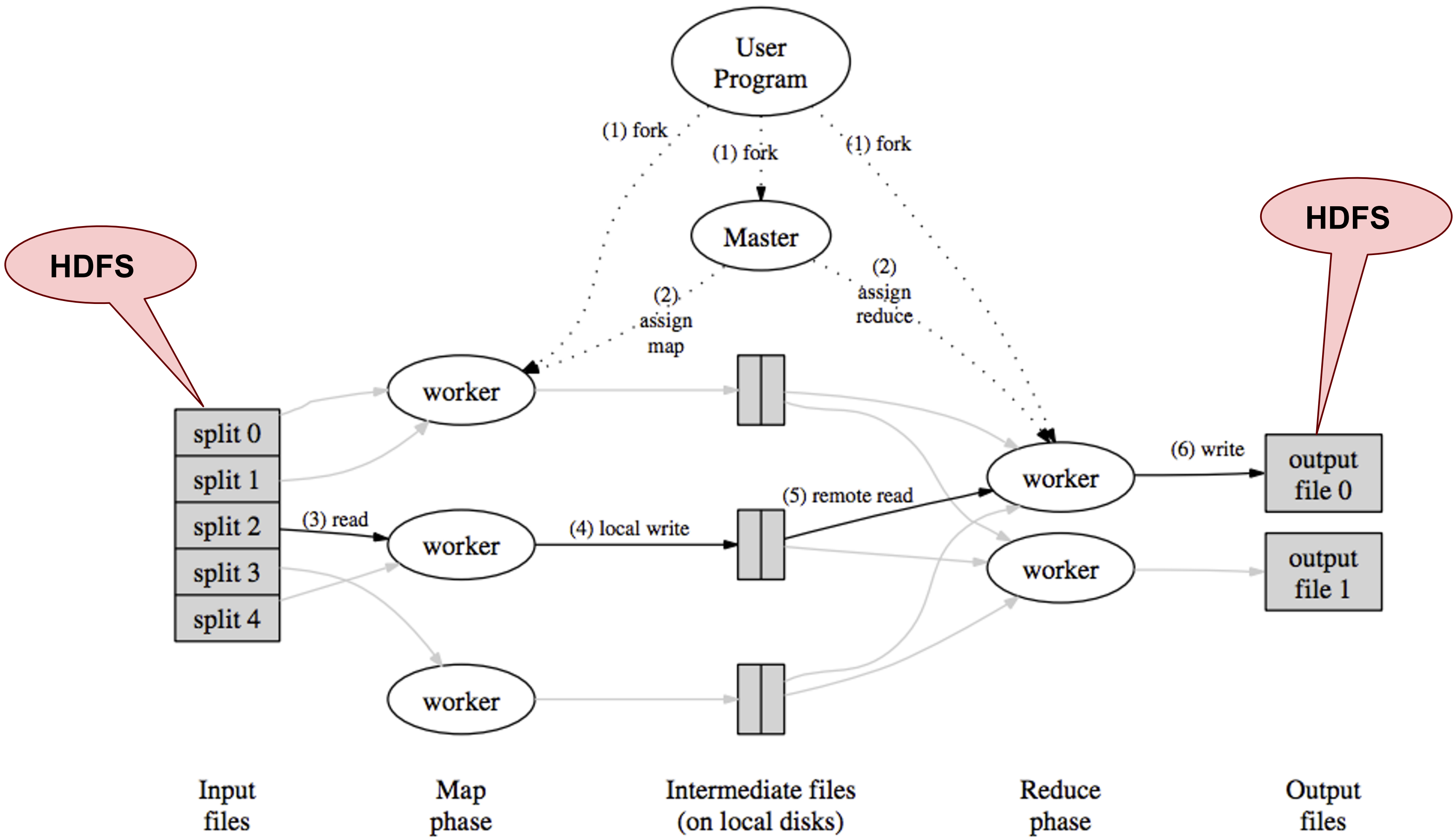


- HDFS Fault Tolerance (3 copies of file)
- “Locality-preserving” compute job placement priority order
  1) On same node as HDFS chunk
  2) On same rack as HDFS chunk
  3) Anywhere else (access over HDFS network)
- MapReduce programming environment
  - Software manages (fault tolerant) execution of tasks on nodes

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

MapReduce Execution



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

Real-World Challenges

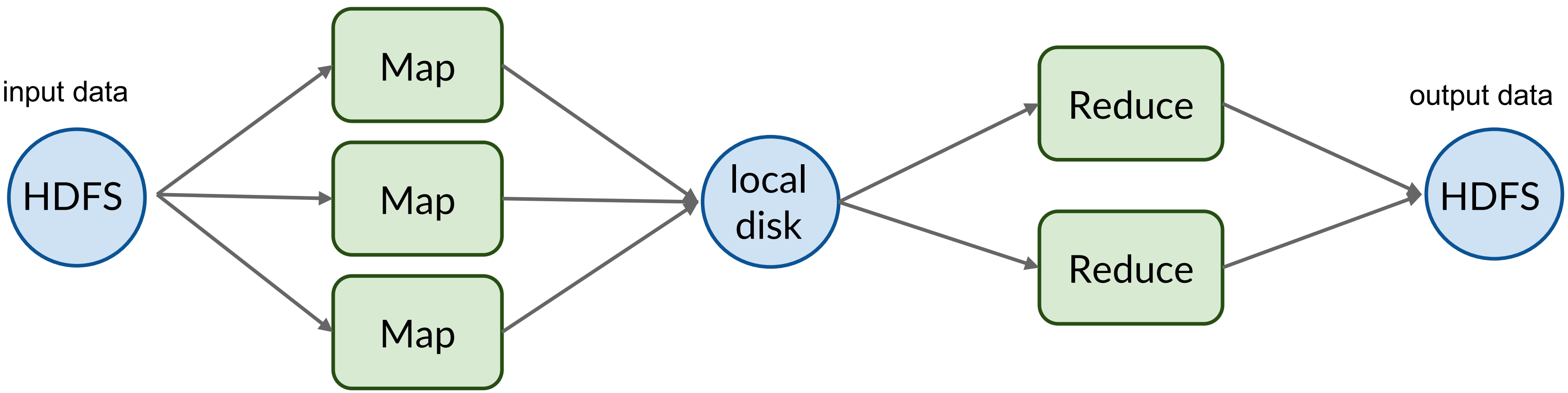
- Fault Tolerance
  - Reliable file system is not enough
  - Workers can fail even if input files available
  - Detect failed worker
    - Heartbeat mechanism
  - Reschedule failed task
- Stragglers
  - Tasks that take a long time to execute
  - Might be bugs, flaky/slow hardware (e.g., disk I/O), poor partitioning, etc.
  - When done with most tasks, reschedule any remaining executing tasks
    - Keep track of redundant executions
    - Significantly reduces overall run time

Cluster Scalability Advantages

- Framework automatically manages fault tolerance
- Dynamically scheduled tasks with state in replicated files
- Provisioning Advantages
  - Can use consumer-grade components
    - maximizes cost-performance
  - Can have heterogeneous nodes
    - More efficient technology refresh
- Operational Advantages
  - Minimal staffing
  - Minimize downtime (operator errors…)

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?