Distributed Systems

15-440/640

Fall 2018

16 – Cluster Computing:
MPI & MapReduce

Readings: “MapReduce: Simplified Data Processing on Large Clusters” Sections 3,4

Daniel S.

Instructor OH & Regrade Requests

Thursday (Yuvraj + Daniel)

- after class to 1pm
- in GHC 4124

Thursday (Yuvraj)

- from 1pm to 2pm
- in Wean 5313

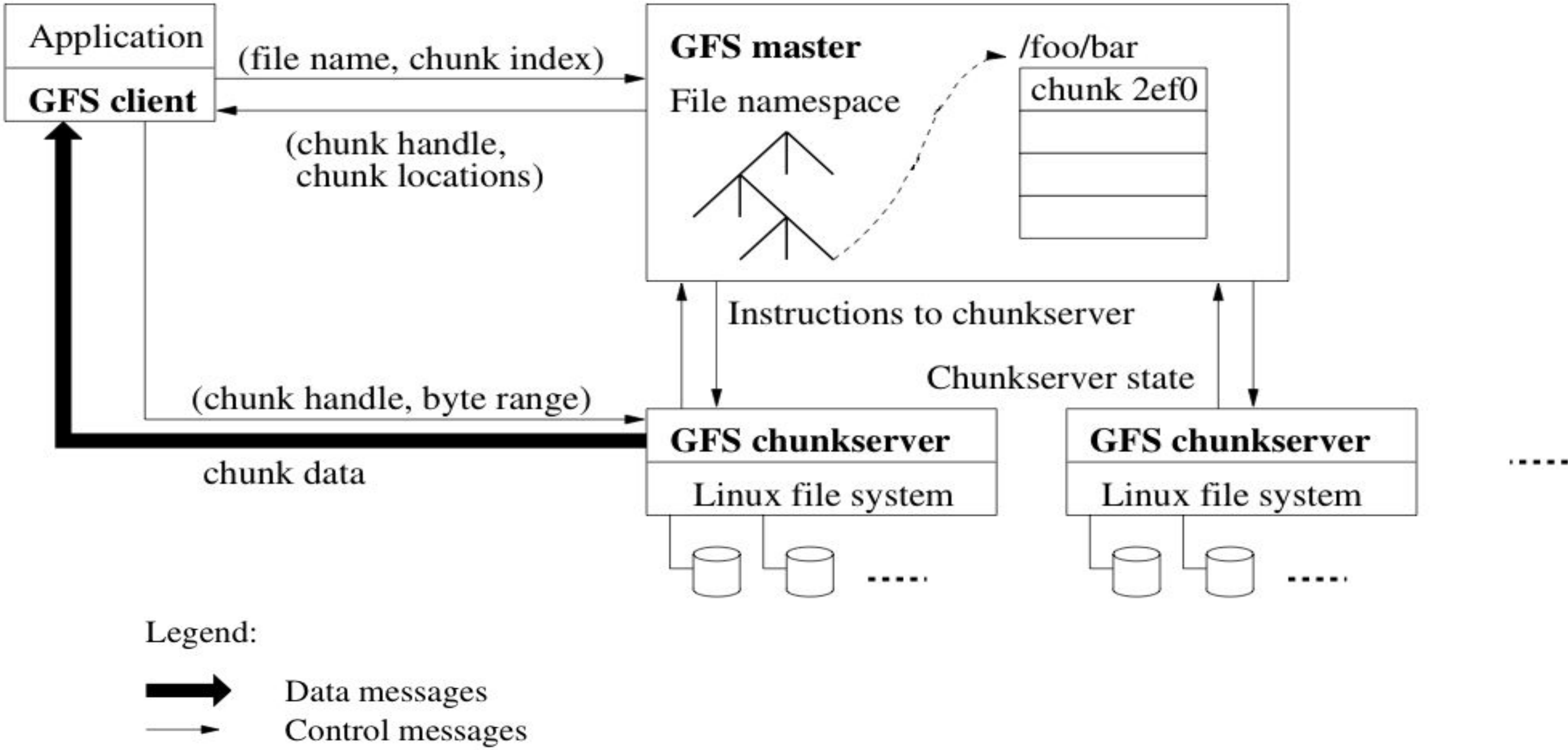
Idea: focus on small group / individual meetings. Put your name into list on our door.

We'll put lists on our doors (after class) and meet with you one by one to discuss grades, goals, … .

Today's Topics

- GFS and HDFS
  - Summary of last lecture
- High-performance computing (HPC)
  - Supercomputers
  - Message Passing Interface (MPI)
- Cluster computing
  - MapReduce
  - Implementation

GFS Architecture: Client/Master/Chunkservers



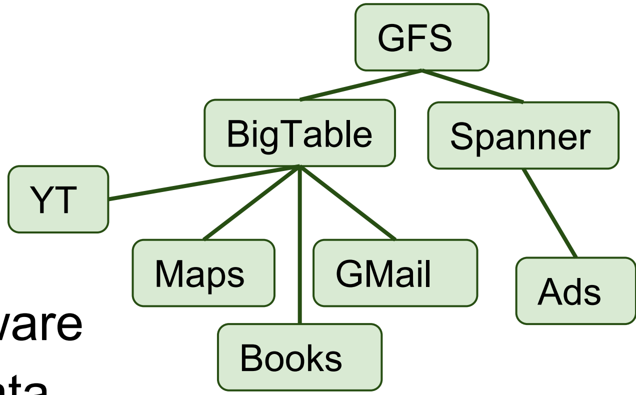
GFS Consistency Model (Metadata)

- Changes to namespace (i.e., metadata) are atomic
  - Done by single master server!
  - Master uses WAL to define global total order of namespace-changing operations

GFS Consistency Model (Data)

- Changes to data are ordered as chosen by a primary
  - But multiple writes from the same client may be interleaved or overwritten by concurrent operations from other clients
- Record append completes at least once, at offset of GFS's choosing
  - Applications must cope with possible duplicates
- Failures can cause inconsistency
  - E.g., different data across chunk servers (failed append)
  - Behavior is worse for writes than appends

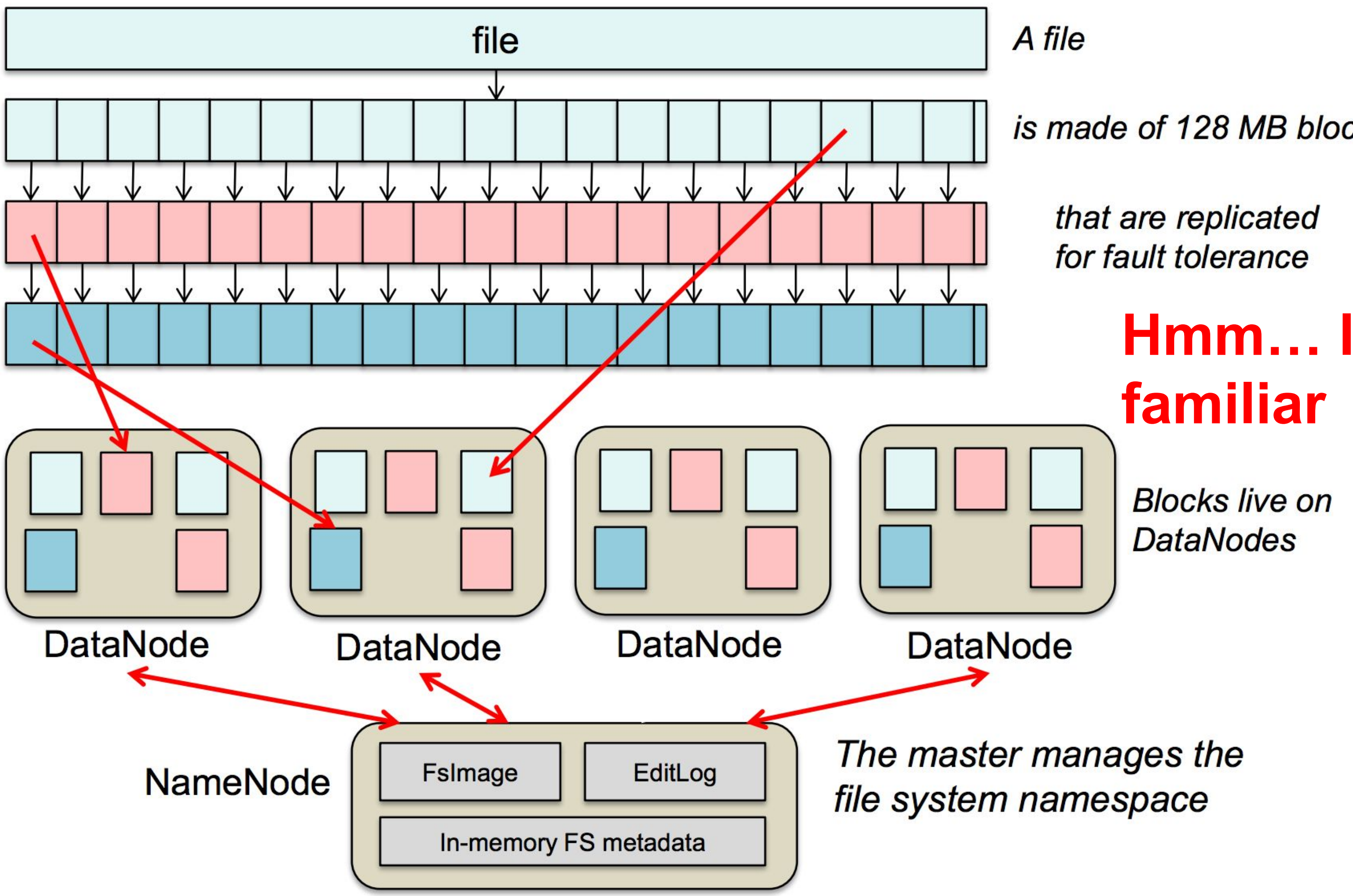
GFS Summary



- Success: used actively by Google
  - Availability and recoverability on cheap hardware
  - High throughput by decoupling control and data
  - Supports massive data sets and concurrent appends
- Semantics not transparent to apps
  - Must verify file contents to avoid inconsistent regions, repeated appends (at-least-once semantics)
- Performance not good for all apps
  - Assumes read-once, write-once workload (no client caching!)

- Successor: Colossus
  - Eliminates master node as single point of failure
  - Storage efficiency: Reed-Solomon (1.5x) instead of Replicas (3x)
  - Reduces block size to be between 1~8 MB
  - Few details public ☹

Apache Hadoop DFS



Hmm… looks familiar

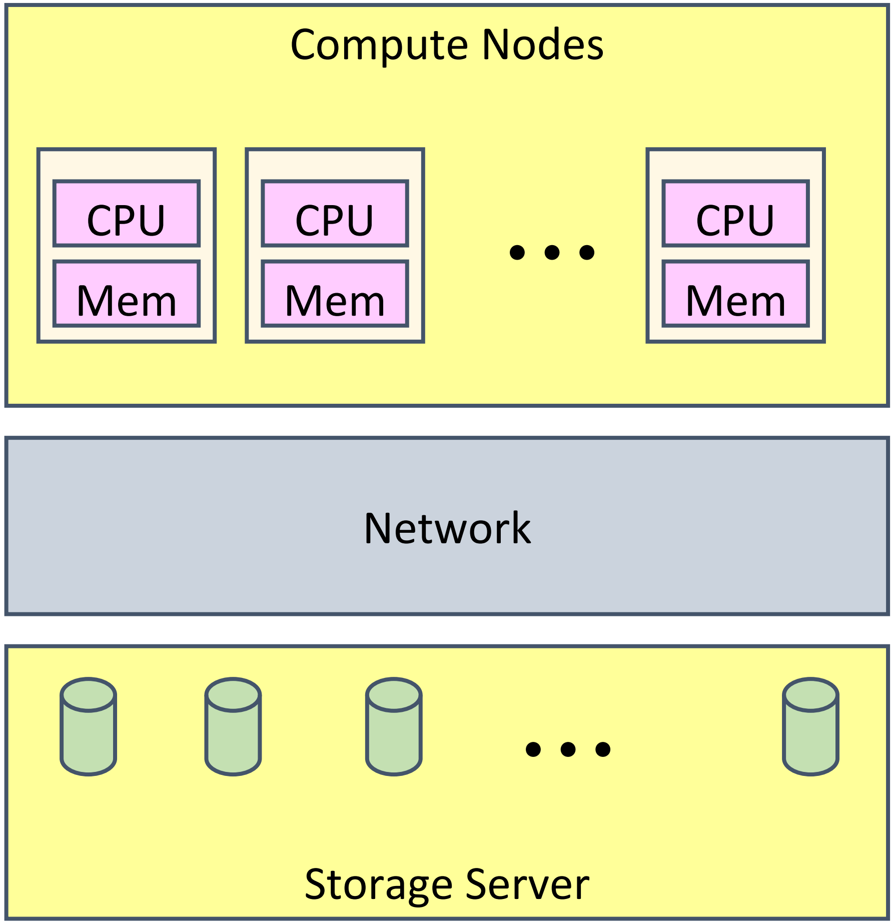
GFS vs. HDFS

GFS	HDFS
Master	NameNode
chunkserver	DataNode
operation log	journal, edit log
chunk	block
random file writes possible	only append is possible
multiple writer, multiple reader model	single writer, multiple reader model
chunk: 32bit checksum over 64KB data pieces (1024 per chunk)	per HDFS block, two files created on a DataNode: data file & metadata file (checksums, timestamp)
default block size: 64MB	default block size: 128MB

Today's Topics

- GFS and HDFS
  - Summary of last lecture
- High-performance computing (HPC)
  - Supercomputers
  - Message Passing Interface (MPI)
- Cluster computing
  - MapReduce
  - Implementation

Typical HPC Machine



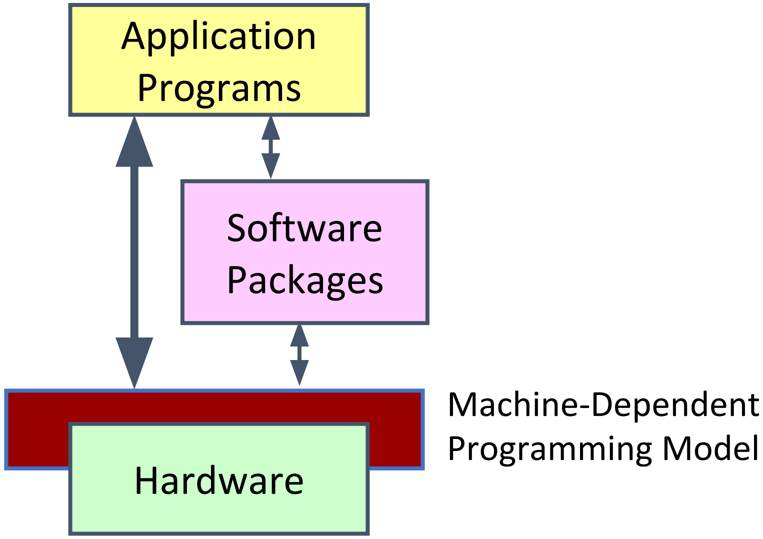
- Compute Nodes
  - High end processor(s)
  - Lots of RAM
- Network
  - Specialized
  - Very high performance
- Storage Server
  - RAID-based disk array

HPC Machine Example

Sunway TaihuLight

- Cores: 10,649,600
- Memory: 1,310,720 GB
- Architecture: Sunway SW26010 (custom built)
  - No caches, 65 cores / on-chip group @ 1.45 GHz
  - Interconnect: “Sunway Network” (custom built)
- 93,014.6 TFlop/s (Top 500: #2)

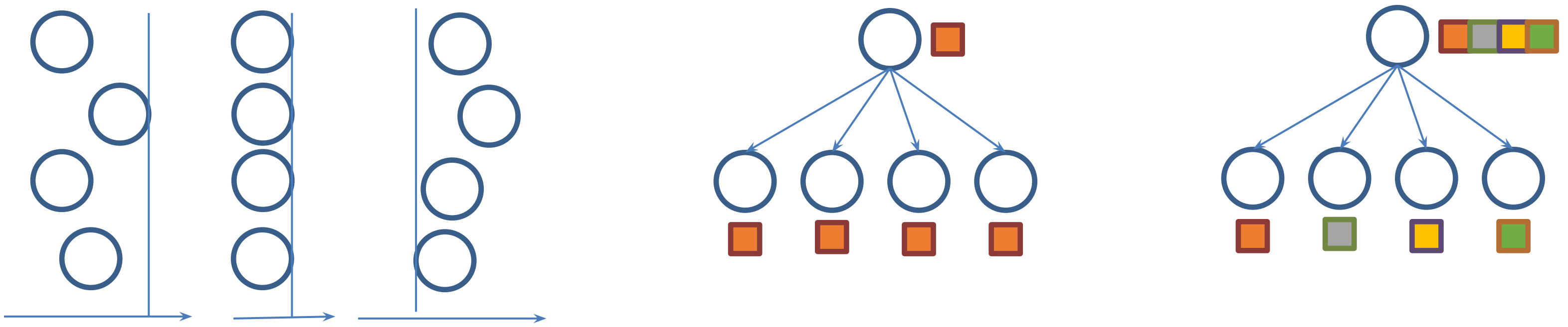
HPC Programming Model



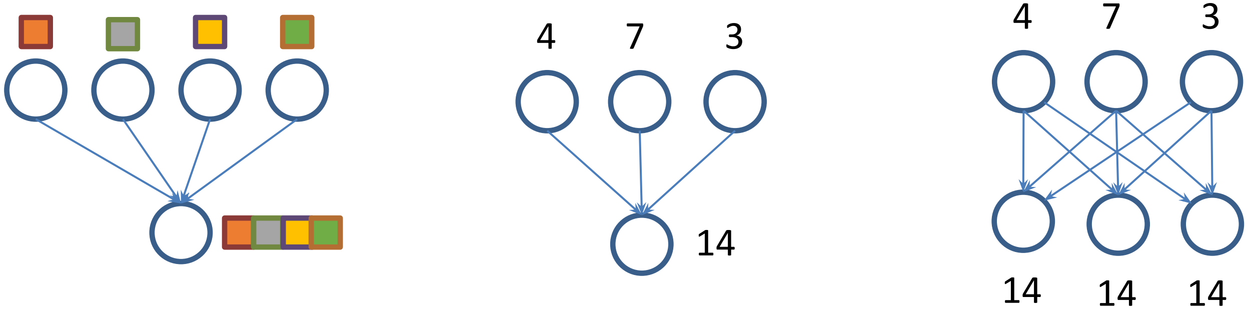
- Programs described at very low level
  - Specify detailed control of processing & communications
- Rely on small number of software packages
  - Written by specialists
  - Limits classes of problems & solution methods

Message Passing Interface (MPI)

- Standardized set of group communication methods
  - Sending: Barrier, Broadcast, Scatter



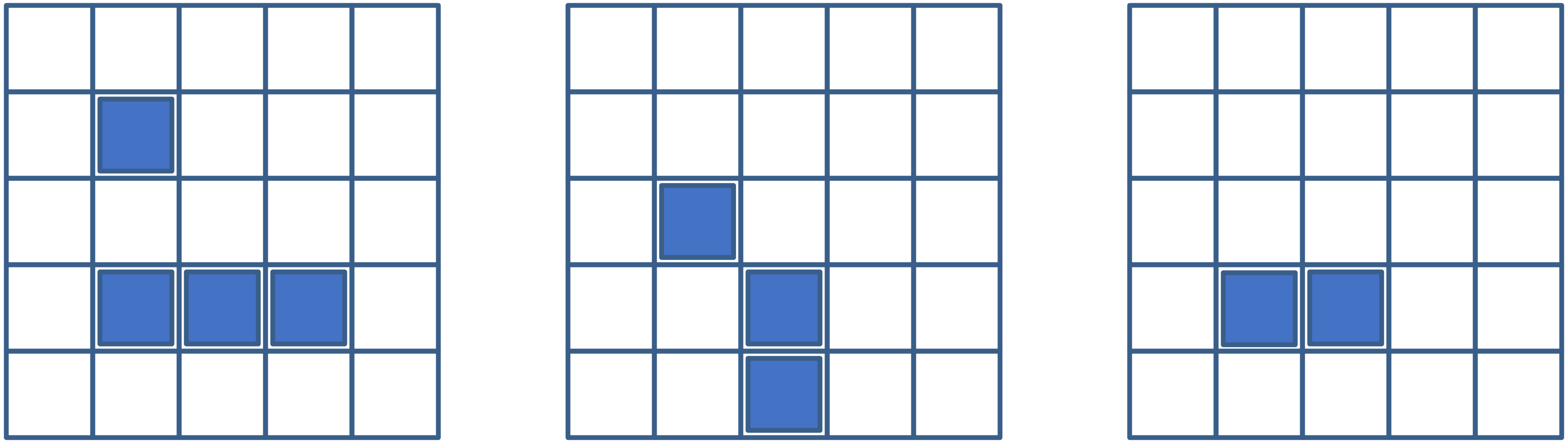
  - Receiving: gather, reduce, all-to-all, and many more



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

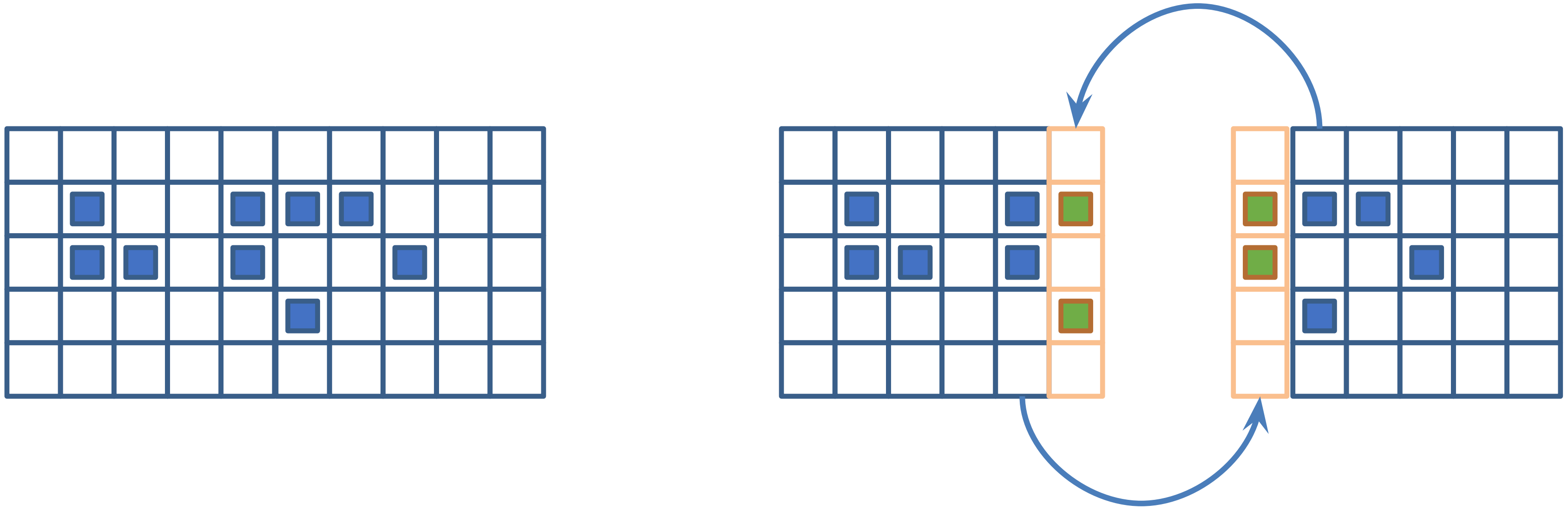
HPC Example: Iterative Simulation I

- 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

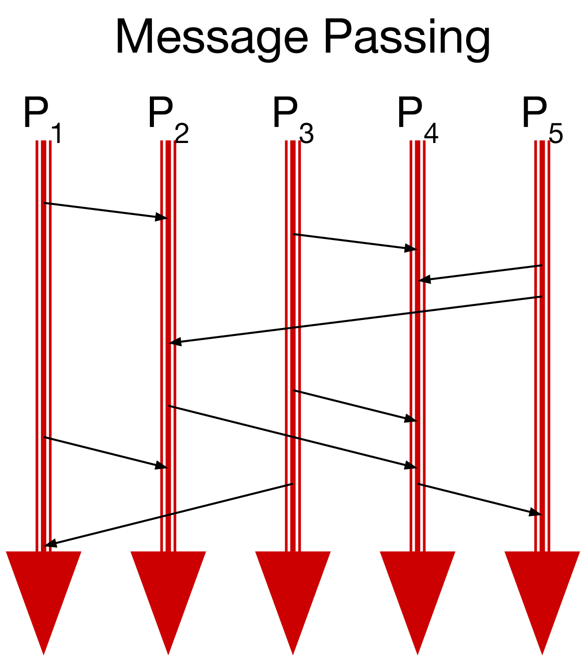


HPC Example: Iterative Simulation II

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

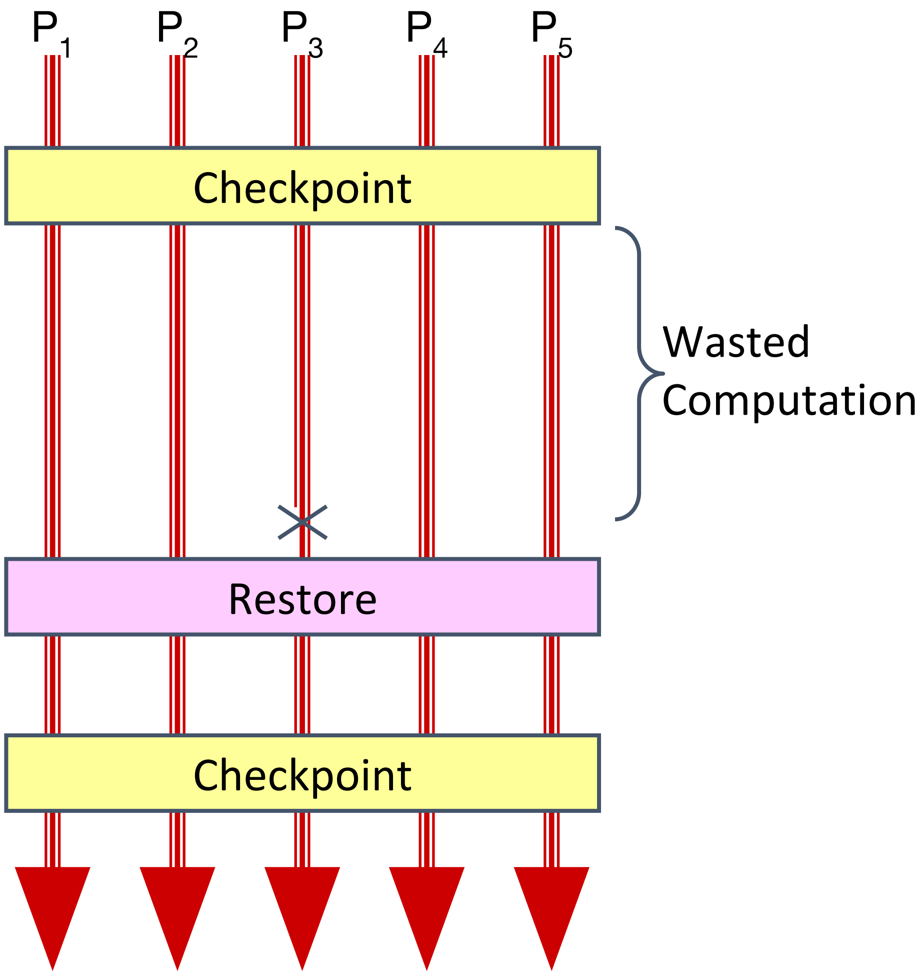


Typical HPC Operation



- Characteristics
  - Long-lived 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

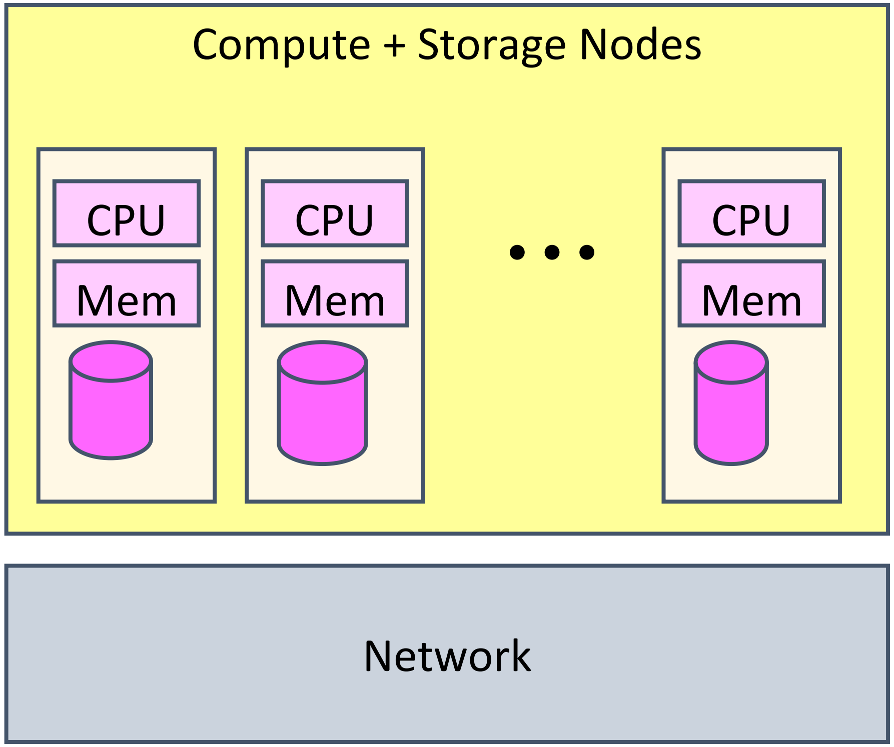


- Checkpoint
  - Periodically store state of all processes
  - Significant I/O traffic
- Restore
  - When failure occurs
  - Reset state to that of last checkpoint
  - All intervening computation wasted
- Performance Scaling
  - Very sensitive to number of failing components

Today's Topics

- GFS and HDFS
  - Summary of last lecture
- High-performance computing (HPC)
  - Supercomputers
  - Message Passing Interface (MPI)
- Cluster computing
  - MapReduce
  - Implementation

Typical Cluster Machine



- Collocate
  Compute + Storage
  - Medium-performance processors
  - Modest memory
  - A few disks
- Network
  - Conventional Ethernet switches
  - 10s-100 Gb/s

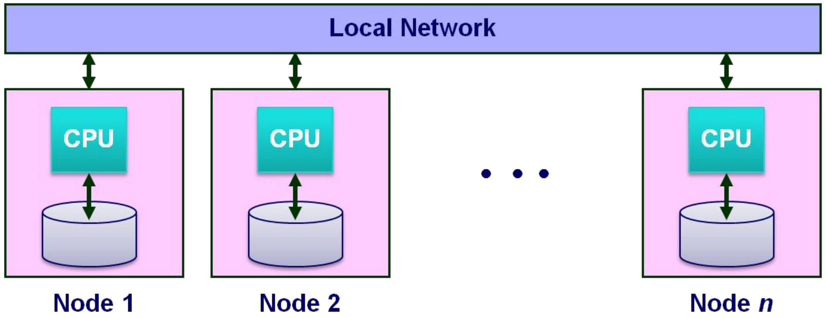
Oceans of Data, Skinny Pipes

- 1 Terabyte
  - Easy to store
  - Hard to move

Disks	MB / s	Time
Seagate Barracuda	115	2.3 hours
Seagate Cheetah	125	2.2 hours
Networks	**MB / s**	**Time**
Home Internet	< 16	> 1 day
Gigabit Ethernet	< 125	> 2.2 hours
PSC Teragrid Connection	< 3,750	> 4.4 minutes

Data-Intensive System Challenge

- For Computation That Accesses 1 TB in 5 minutes
  - Data distributed over 100+ disks
    - Assuming uniform data partitioning
  - Compute using 100+ processors
  - Connected by 10-Gbit-Ethernet



- System Requirements
  - Lots of disks
  - Lots of processors
  - Located in close proximity
    - Within reach of fast, local-area network

How To Program A Cluster?

Example I:

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

Stored on thousands of machines

Assume you can log into all those machines

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

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

e.g., machines fail, get rebooted

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

How To Program A Cluster?

Example II:

Social network graph, stored as Person -> Friend1 Friend2 ..

Stored on thousands of machines **in any order**

Assume you can log into all those machines

How do you count the number of mutual friendships for all pairs of people, e.g., "you and Joe have 147 friends in common"

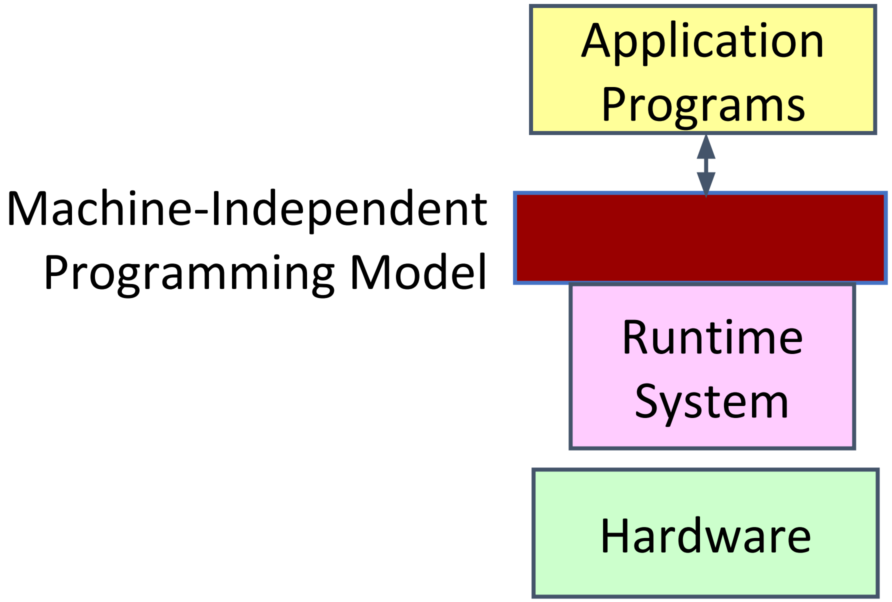
Input:

A -> B C D
B -> A
C -> A D
D -> A C
....

Ouput:

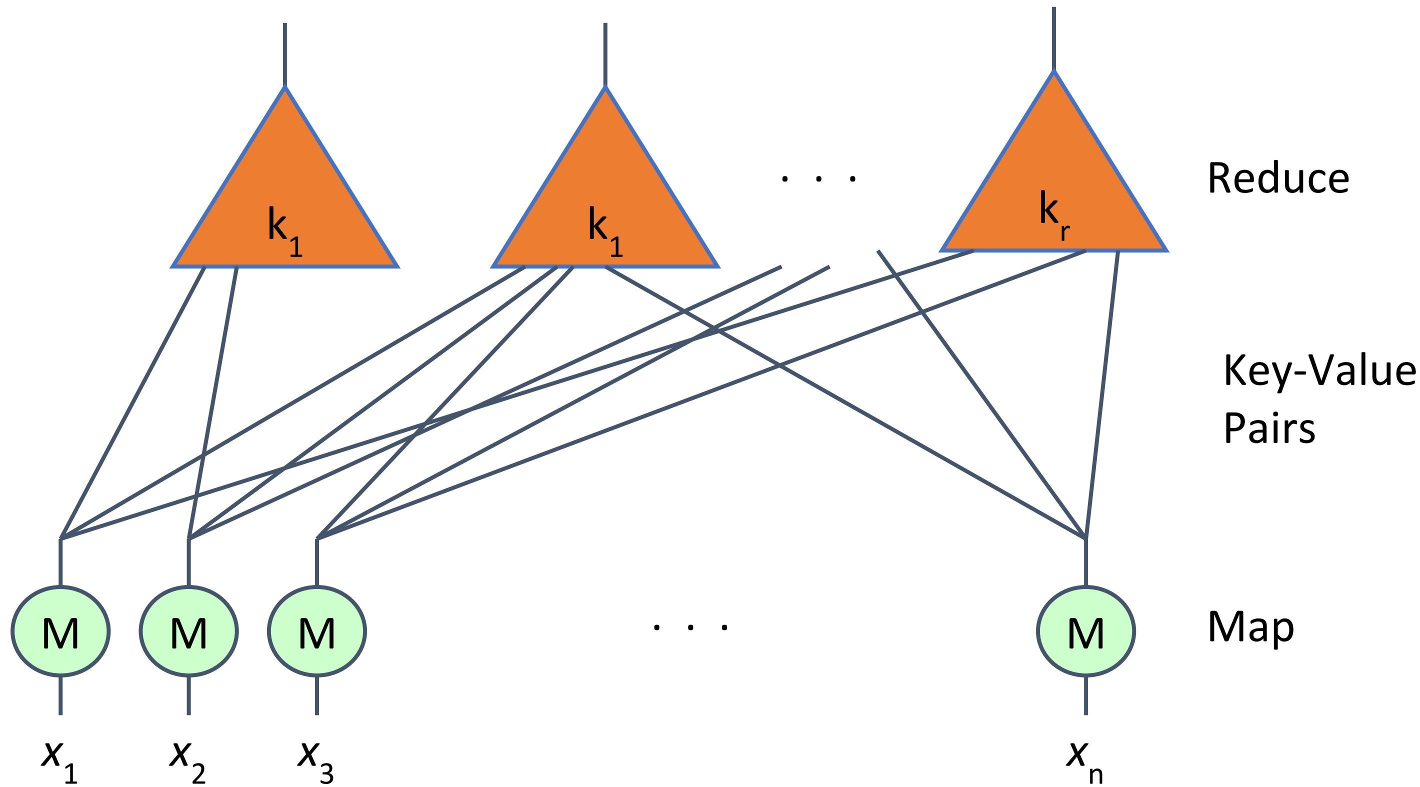
A B -> 0
A C -> 1
B C -> 0 ...

Cluster Programming Model



- Application programs written in terms of high-level data operations
- Runtime system controls scheduling, load balancing, …
- This is idealized. In practice, no perfect cluster programming model.
- Very common model: MapReduce

MapReduce Cluster Model

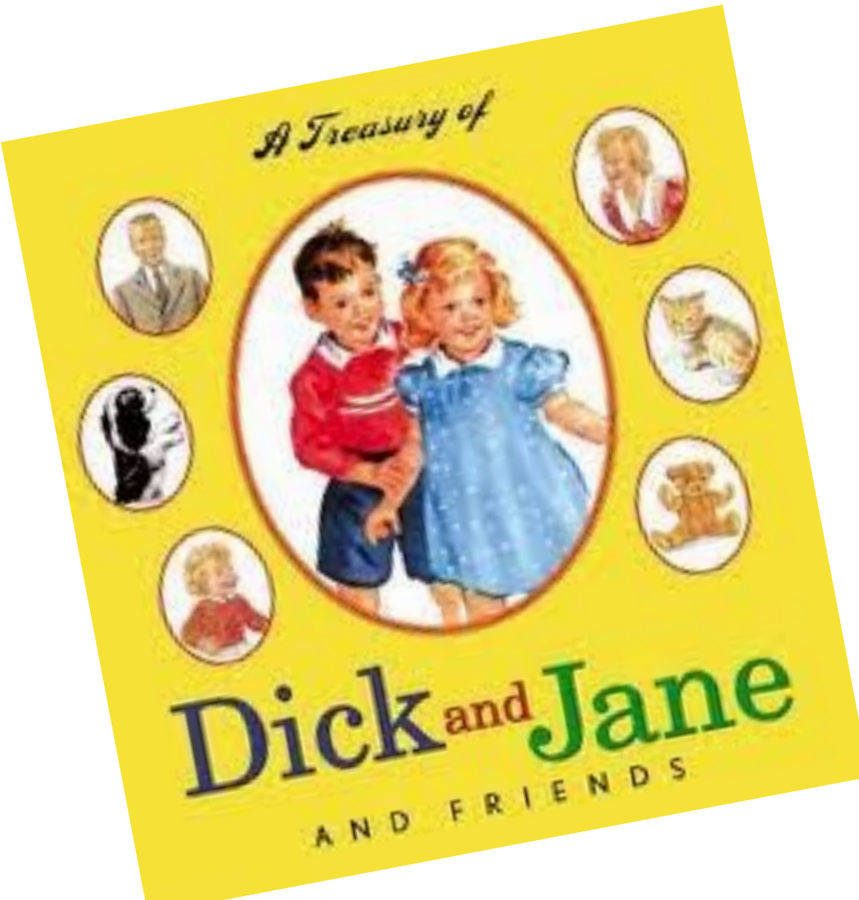


- Map computation across many objects
- Flexible aggregation of results
- System solves resource allocation & reliability

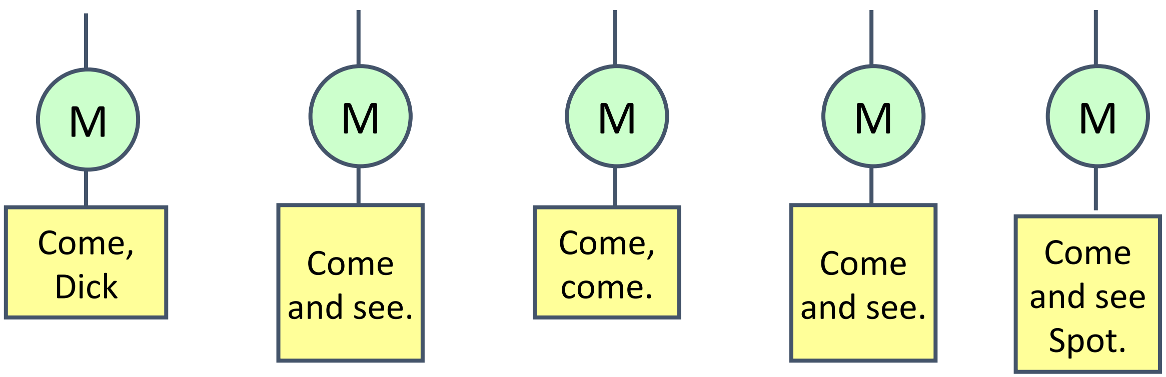
Dean & Ghemawat: "MapReduce: Simplified Data Processing on Large Clusters", OSDI 2004

Example I MapReduce

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



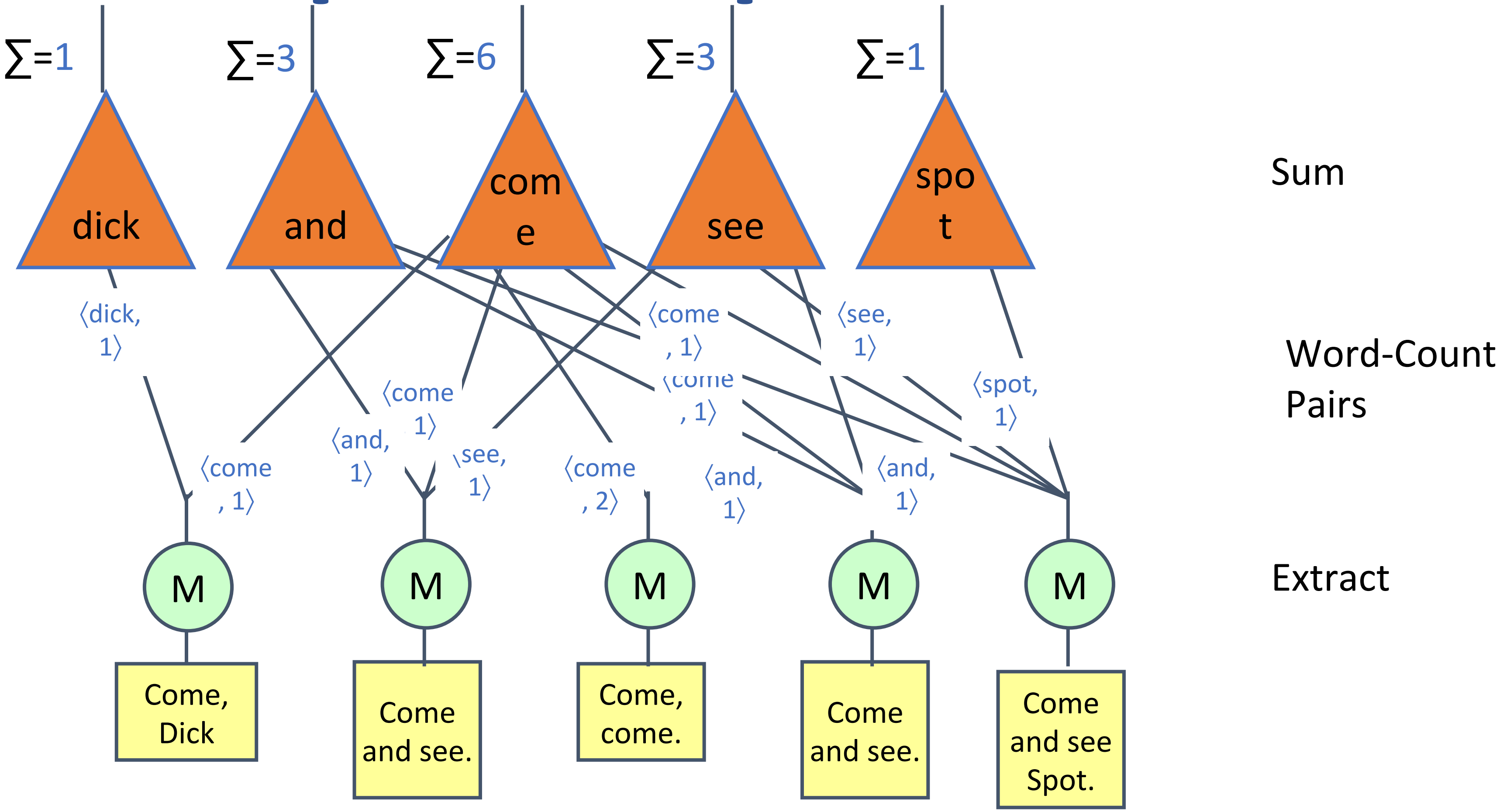
Example I MapReduce



Extract

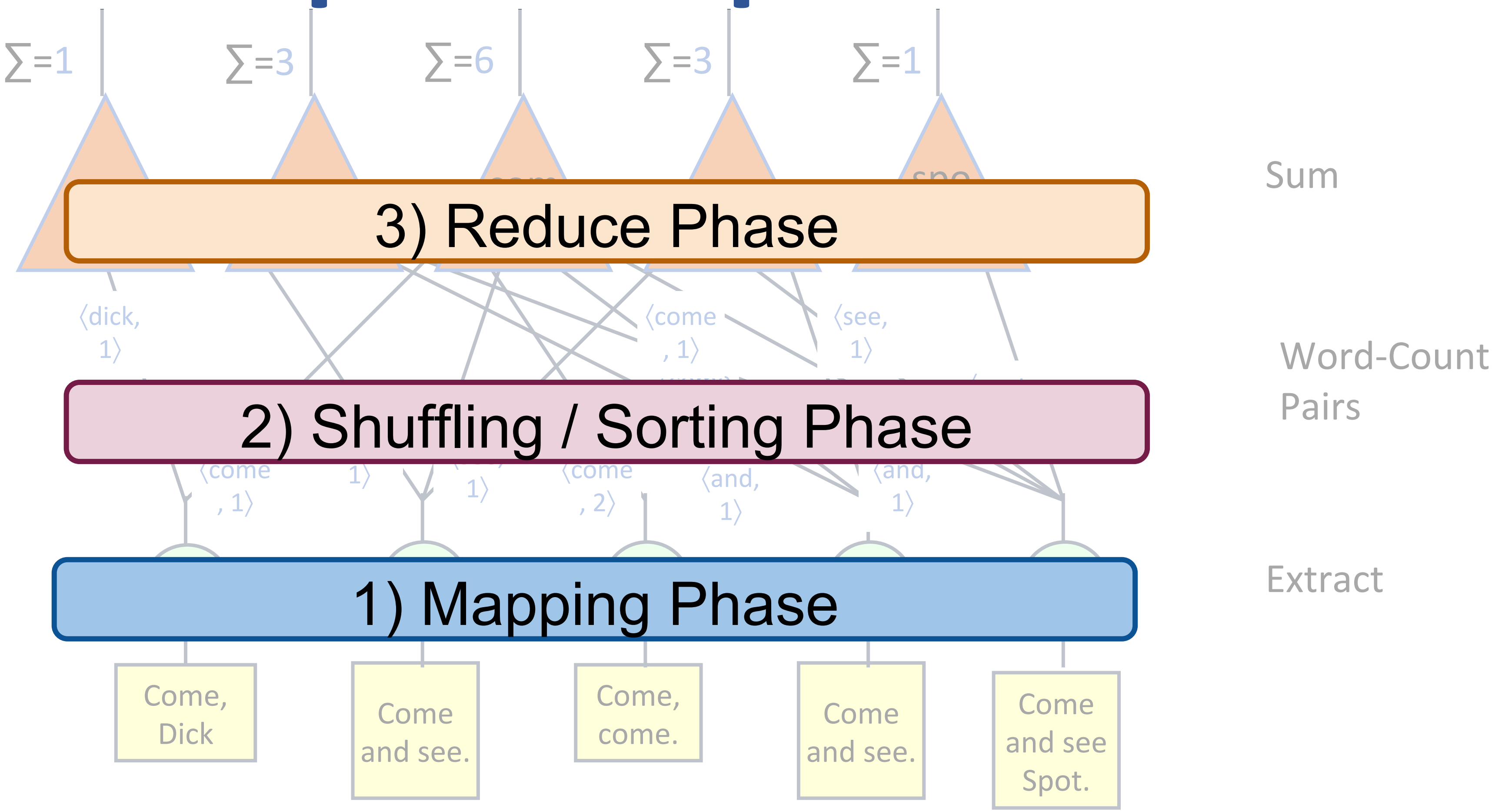
- Calculate word frequency of set of documents

Example I MapReduce



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

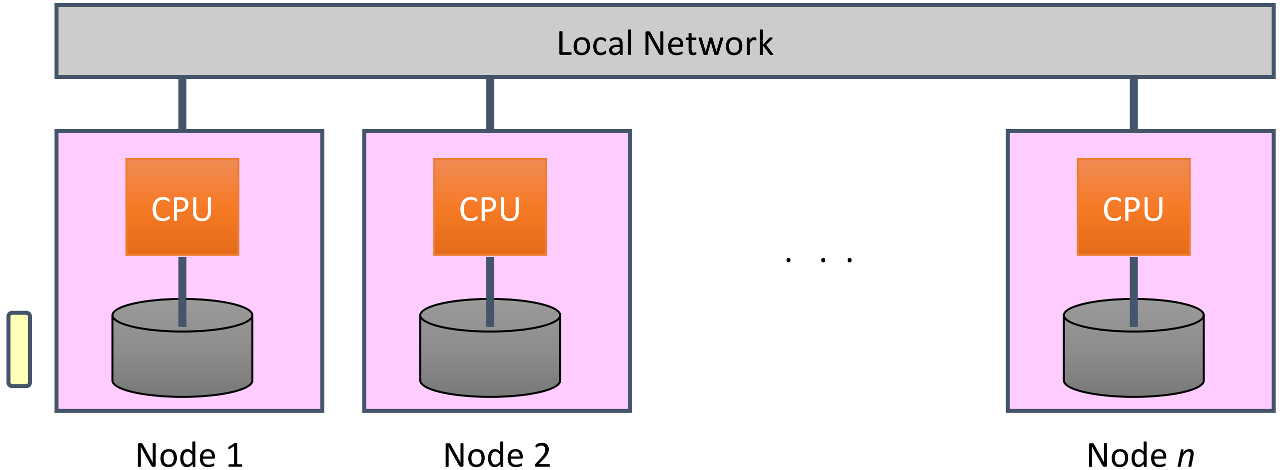
Example I MapReduce



- 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 prio 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 execution of tasks on nodes

Hadoop MapReduce API

- Requirements
  - Programmer must supply Mapper & Reducer classes
- Mapper
  - Steps through file one line at a time
  - Code generates sequence of <key, value> pairs
  - Default types for keys & values are strings
    - Can use anything "writable", lots of conversion methods
- Shuffling/Sorting
  - MapReduce's built in aggregation by key
- Reducer
  - Given key + iterator that generates sequence of values
  - Generate one or more <key, value> pairs

Example II MapReduce

Example II:

Social network graph

Stored as Person -> Friend 1, Friend 2, ...

Input:

A -> B C D
B -> A C D E
C -> A B D E
D -> A B C E
E -> B C D

Count the number of mutual friendships, e.g., "you and Joe have 147 friends in common"

How to do this in the MapReduce framework?

Example II MapReduce

High-level idea: first create all the pairs (map), then calculate intersection of friend lists (reduce).

<u>input:</u>

A -> B C D
B -> A C D E
C -> A B D E
D -> A B C E
E -> B C D

<u>map(A -> B C D):</u>

(A B) -> B C D
(A C) -> B C D
(A D) -> B C D

<u>map(B -> A C D E):</u>

(A B) -> A C D E
(B C) -> A C D E
(B D) -> A C D E
(B E) -> A C D E

<u>map(C -> A B D E):</u>

(A C) -> A B D E ...

<u>shuffling phase:</u>

(A B) -> (A C D E) (B C D)

(A C) -> (A B D E) (B C D)

(A D) -> (A B C E) (B C D)

(B C) -> (A B D E) (A C D E)

(B D) -> (A B C E) (A C D E)

Example II MapReduce

High-level idea: first create all the pairs (map), then calculate intersection of friend lists (reduce).

<u>map(A -> B C D):</u>

(A B) -> B C D

(A C) -> B C D

(A D) -> B C D

<u>map(B -> A C D E):</u>

(A B) -> A C D E

(B C) -> A C D E

(B D) -> A C D E

(B E) -> A C D E

<u>map(C -> A B D E):</u>

(A C) -> A B D E ...

...

<u>shuffling phase:</u>

(A B) -> (A C D E) (B C D)

(A C) -> (A B D E) (B C D)

(A D) -> (A B C E) (B C D)

(B C) -> (A B D E) (A C D E)

(B D) -> (A B C E) (A C D E)

<u>reduce phase:</u>

(A B) -> (C D)

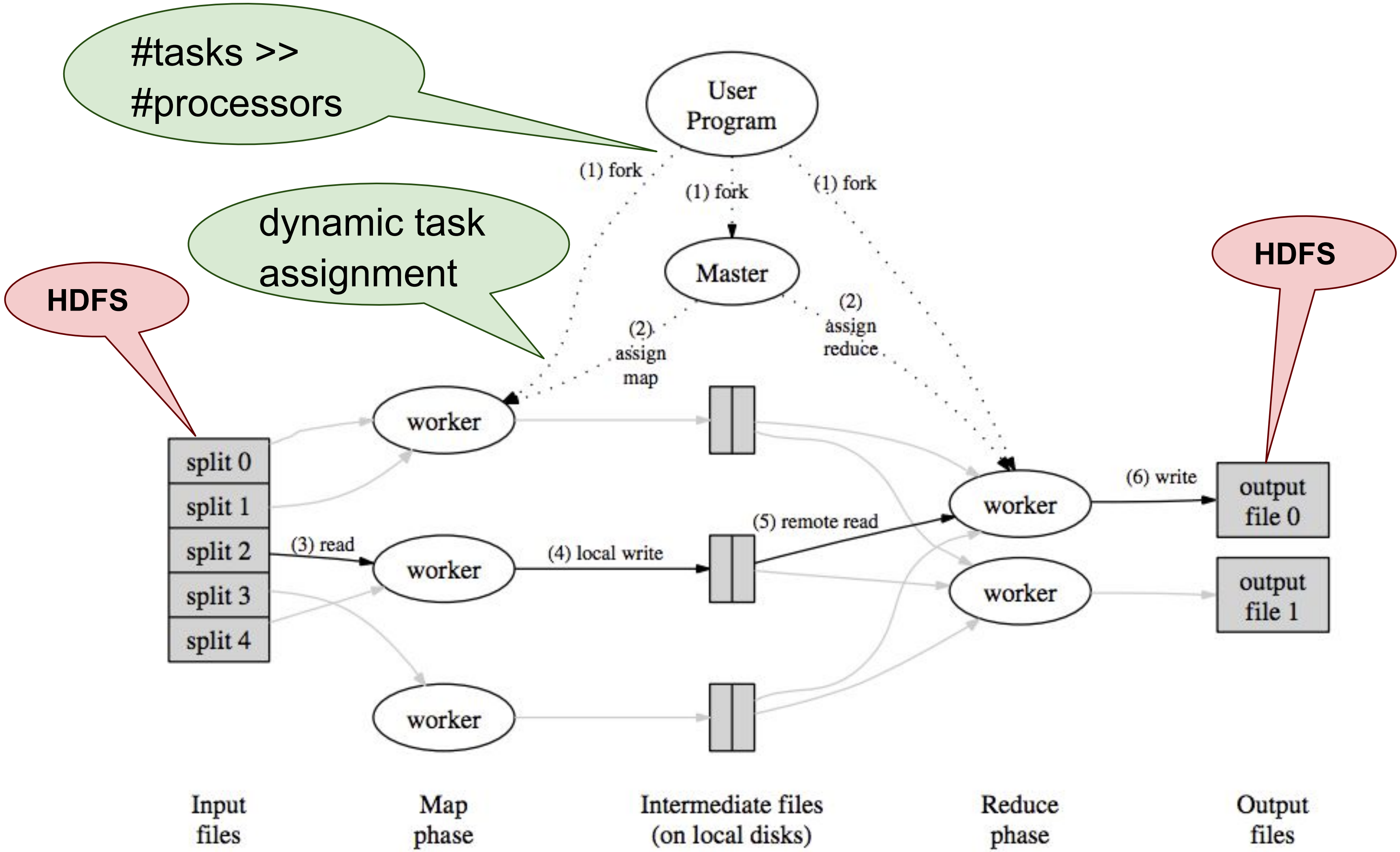
(A C) -> (B D)

(A D) -> (B C)

(B C) -> (A D E)

(B D) -> (A C E) ...

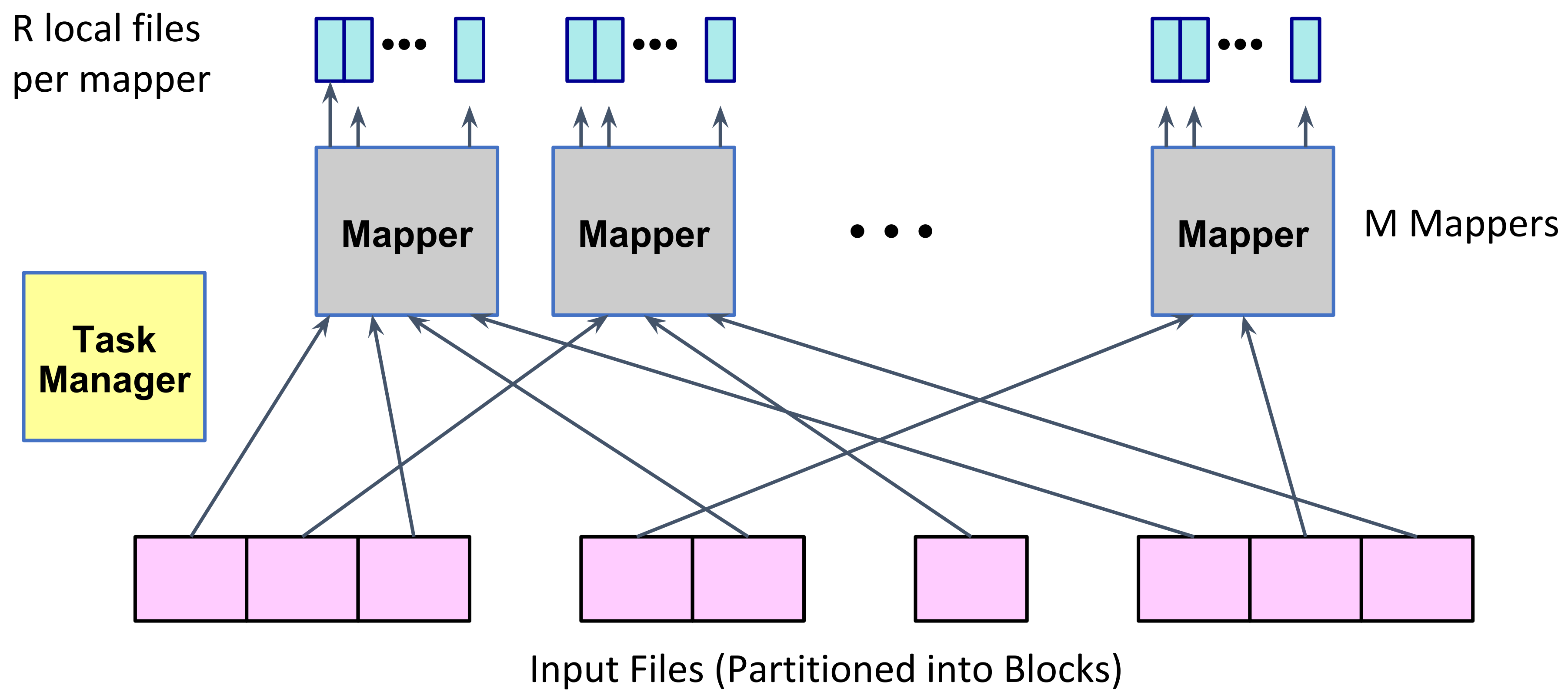
MapReduce Execution



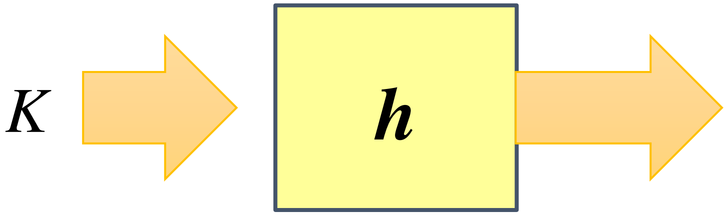
Dean & Ghemawat: “MapReduce: Simplified Data Processing on Large Clusters”, OSDI 2004

Mapping

- Dynamically map input file blocks onto mappers
- Each generates key/value pairs from its blocks
- Each writes R files on local file system

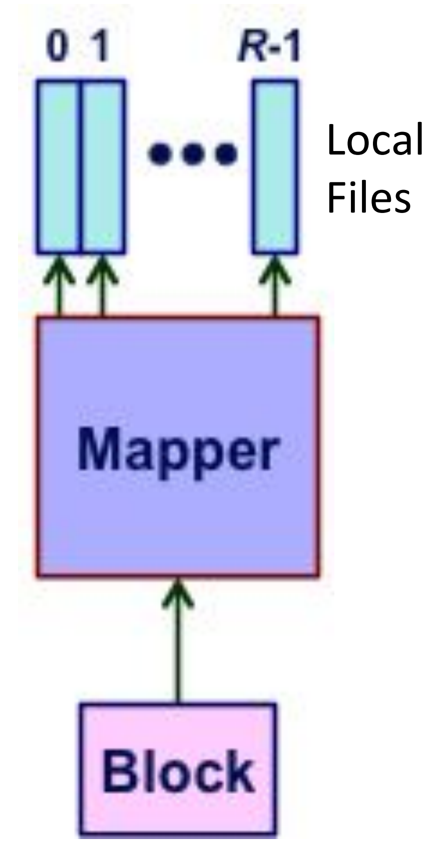


Hashing



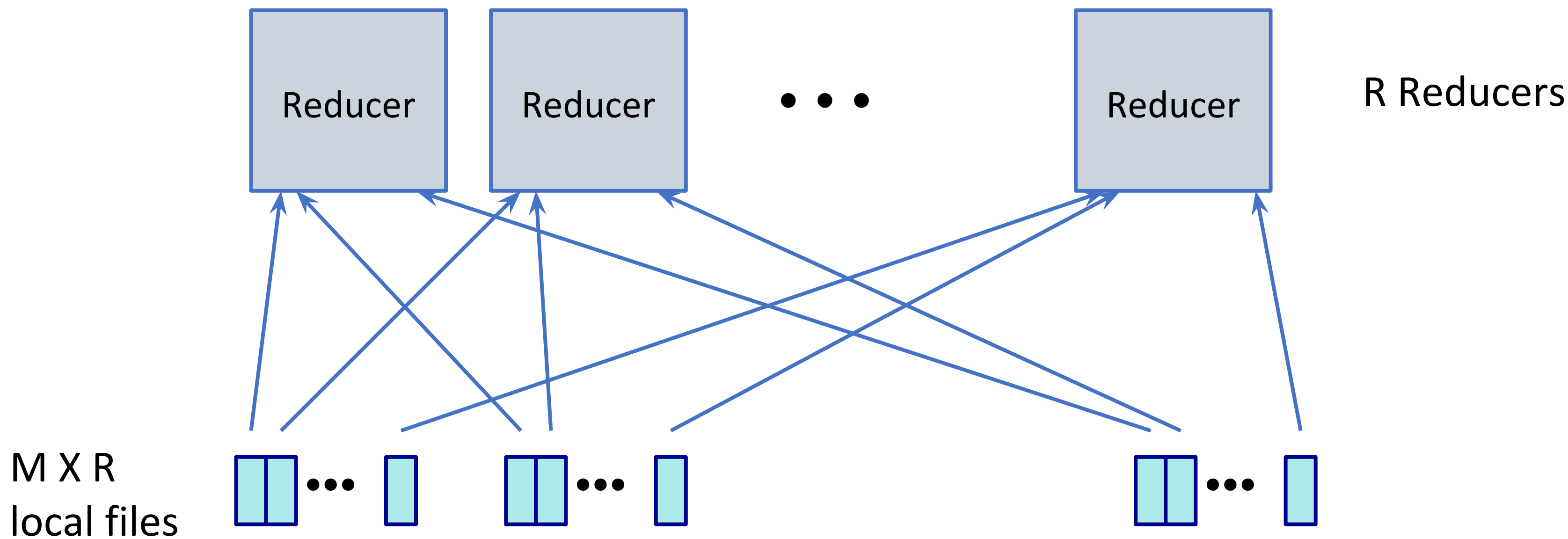
$$h(K) \in \{0, \ldots, R-1\}$$

- Hash Function *h*
  - Maps each key *K* to integer *i* such that $0 \le i < R$
- Mapper Operation
  - Reads input file blocks
  - Generates pairs $\langle K, V \rangle$
  - Writes to local file $h(K)$



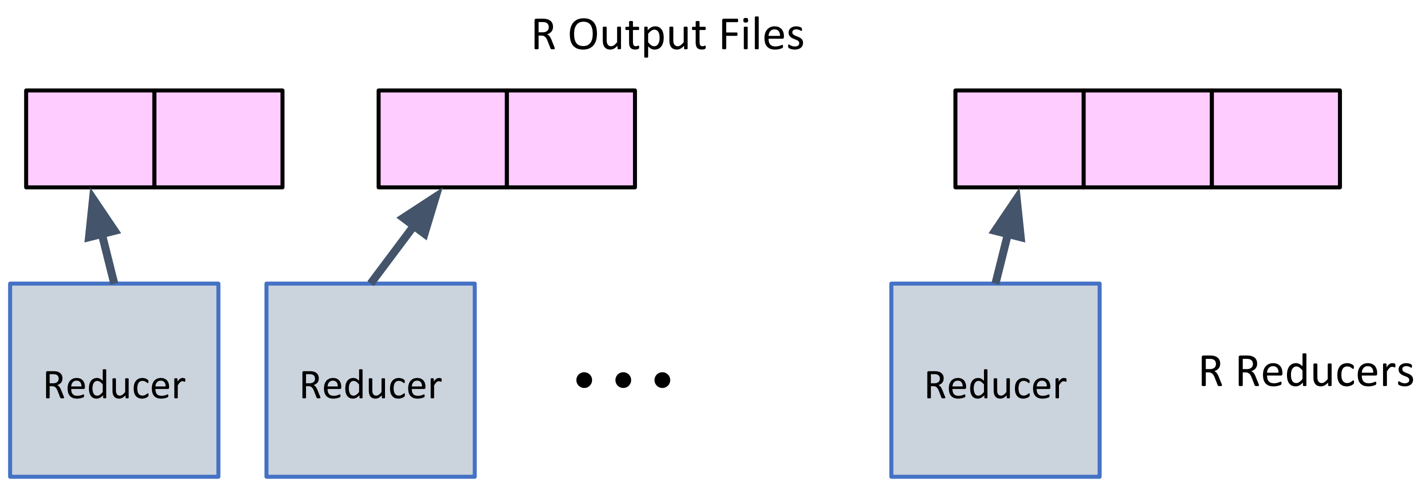
Shuffling

- Each Reducer:
  - Handles 1/R of the possible key values
  - Fetches its file from each of M mappers
  - Sorts all of its entries to group values by keys



Reducing

- Each Reducer:
  - Executes reducer function for each key
  - Writes output values to cluster filesystem



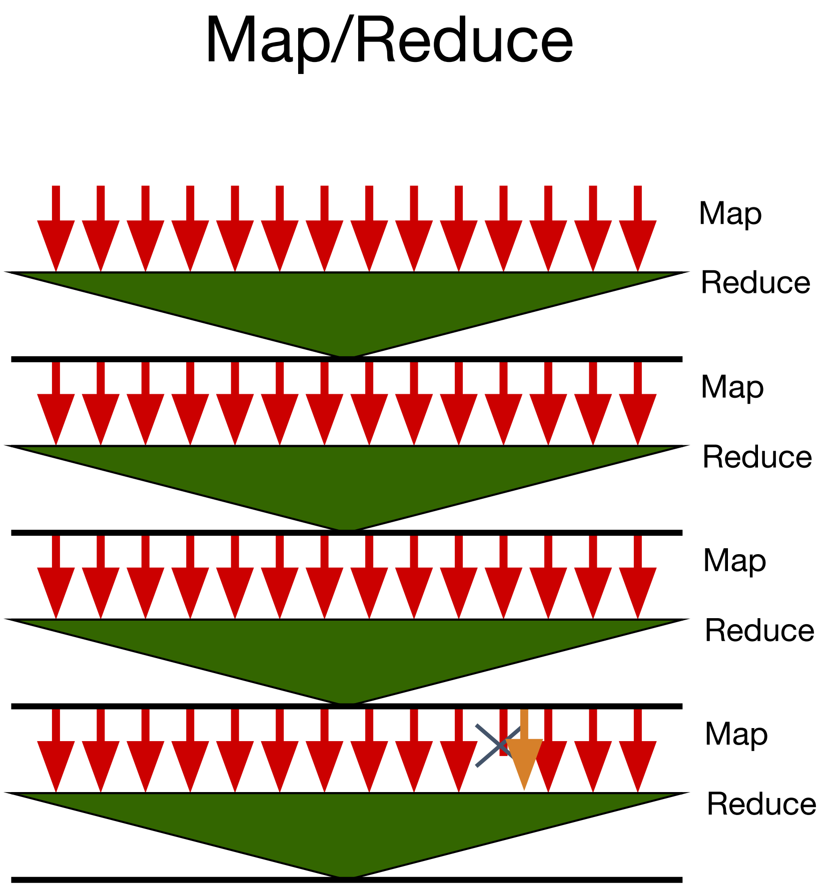
Cluster Scalability Advantages

- Framework following distributed system design principles
- Dynamically scheduled tasks with state in replicated files
- Provisioning Advantages
  - Can use consumer-grade components
    - maximizes cost-peformance
  - Can have heterogenous nodes
    - More efficient technology refresh
- Operational Advantages
  - Minimal staffing
  - No downtime

Real-World Challenges

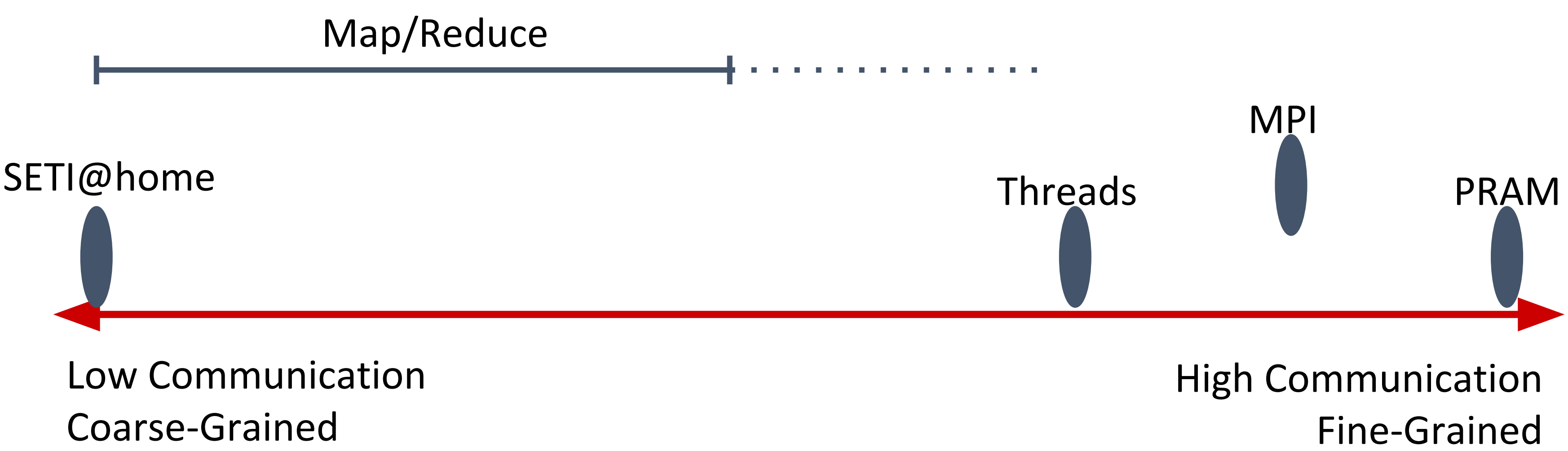
- Fault Tolerance
  - Assume reliable file system
  - Detect failed worker
    - Heartbeat mechanism
  - Reschedule failed task
- Stragglers
  - Tasks that take long time to execute
  - Might be bug, flaky hardware, or poor partitioning
  - When done with most tasks, reschedule any remaining executing tasks
    - Keep track of redundant executions
    - Significantly reduces overall run time

Map/Reduce Operation



- Characteristics
  - Computation broken into many, short-lived tasks
  - Use disk storage to hold intermediate results
  - Failure → Reschedule task
- Strengths
  - Great flexibility in placement, scheduling, and load balancing
  - Can access large data sets
- Weaknesses
  - Higher overhead
  - Lower raw performance

Exploring Parallel Computation Models



- MapReduce Provides Coarse-Grained Parallelism
  - Computation done by independent processes
  - File-based communication
- Observations
  - Relatively "natural" programming model
  - Research issue to explore full potential and limits

Map Reduce vs. MPI

- Both are examples of scale-out systems
- MPI:
  - + handles communicating components
  - + allows tightly-coupled parallel tasks
  - + good for iterative computations
  - – more complex model (explicit messaging)
  - – Failure handling left to application
- Map Reduce:
  - + simple programming, failure model
  - + good for loosely-coupled, coarse-grain parallel tasks
  - ± oriented towards disk-based data (that won’t fit into RAM)
  - – not good for interaction, highly-iterative computation

MPI/MapReduce Conclusions

- Distributed Systems Concepts Lead to Scalable Machines
  - Loosely coupled execution model
  - Lowers cost of procurement & operation
- MapReduce Used Everywhere
  - Hadoop makes it widely available
  - Great for some applications, good enough for many others, inefficient for specialized applications (e.g., simulations)
- Lots of Work to be Done
  - Richer set of programming models and implementations
  - Expanding range of applicability
    - Problems that are data and compute intensive
    - The future of supercomputing?

Cluster Computing on Graphs

Lots of valuable data in graphs

about **people**: social networks, facebook.com

about **products**: advertising, amazon.com

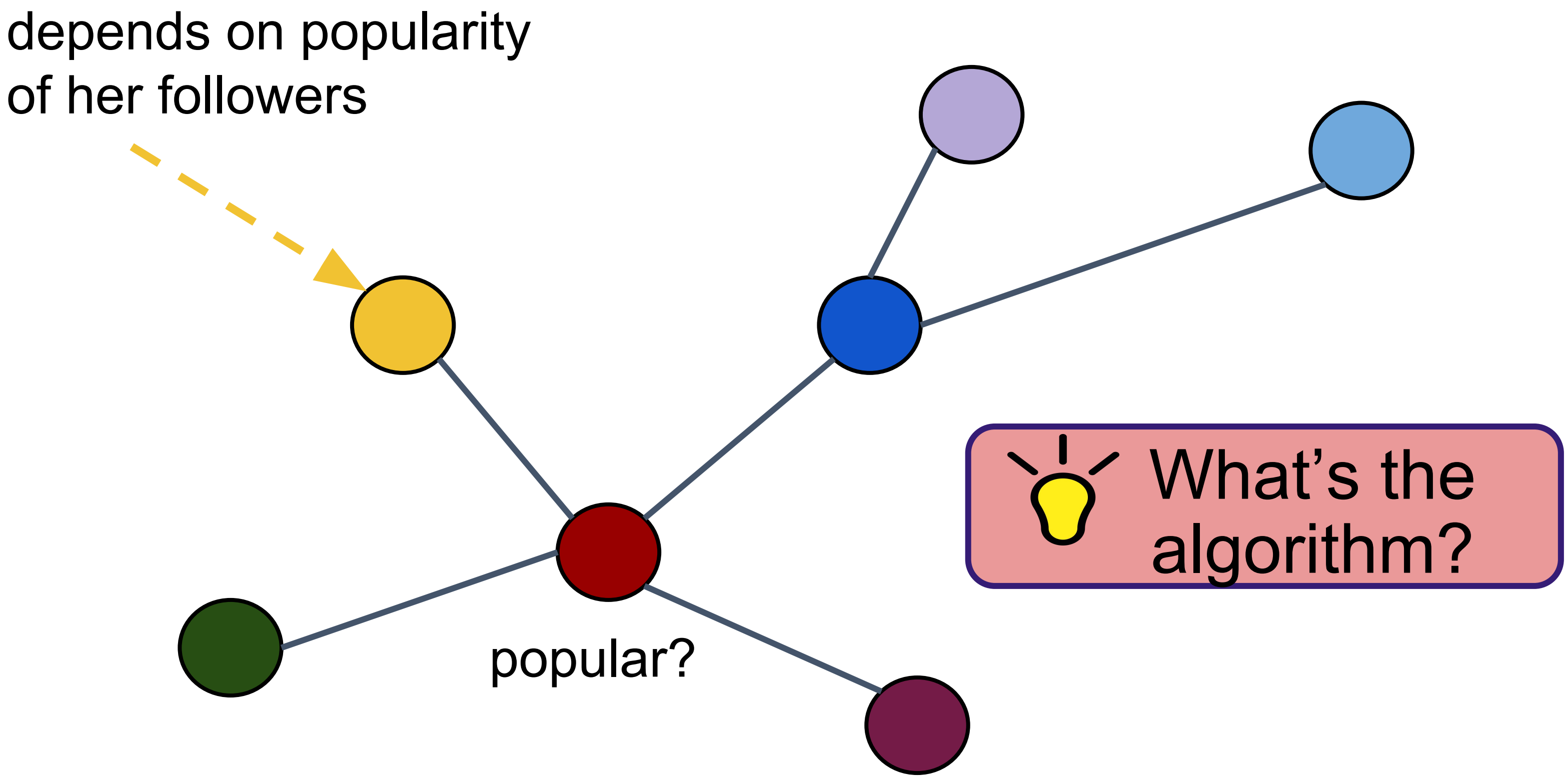
about **interests**: online streaming, netflix.com

about **ideas**: collaborative encyclopedias, wikipedia.org

… and **the relationships between them**

Cluster Computing on Graphs

Popular graph algorithm:



Page Rank: R[i] = 0.15 + weighted sum of R[j]

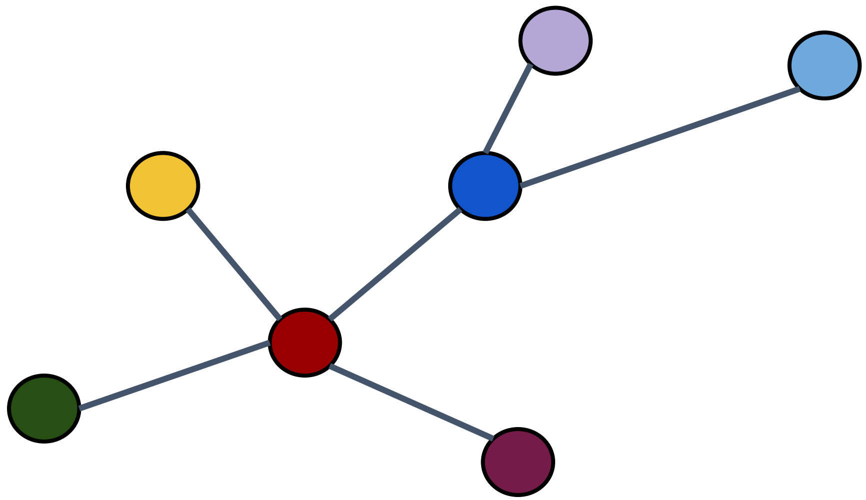
for all neighbors j

Cluster Computing on Graphs

Implementation idea:

update ranks in parallel

iterate until converged



Framework 1: MapReduce

many iterations, always save to disk

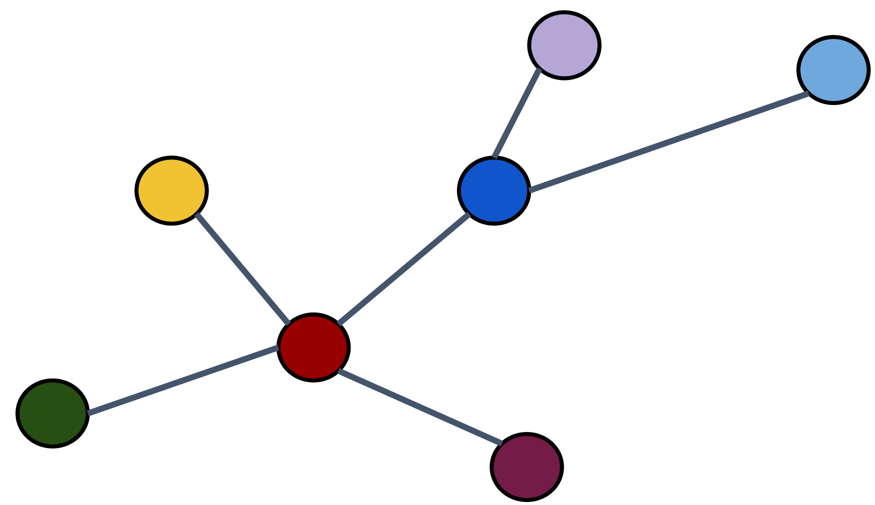
slow, hard to work with graph abstraction

Cluster Computing on Graphs

Implementation idea:

update ranks in parallel

iterate until converged



Framework 2: Google Pregel (MPI on graphs)

abstraction: messaging between vertices in graph

receive message: neighbors' ranks

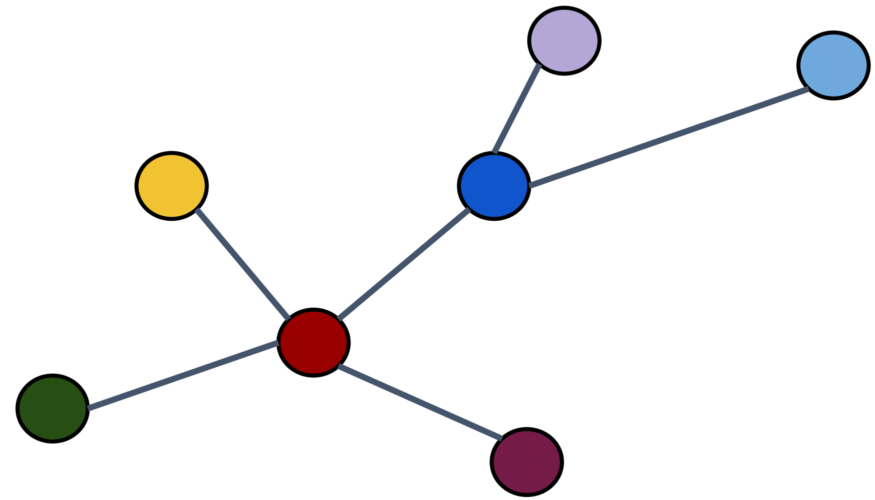
send message: our own rank (to all neighbors)

Cluster Computing on Graphs

Implementation idea:

update ranks in parallel

iterate until converged



Framework 3: CMU Graphlab (shared state model)

abstraction: “emulate all nodes on same machine”

iterate (foreach) over neighbors [j]:

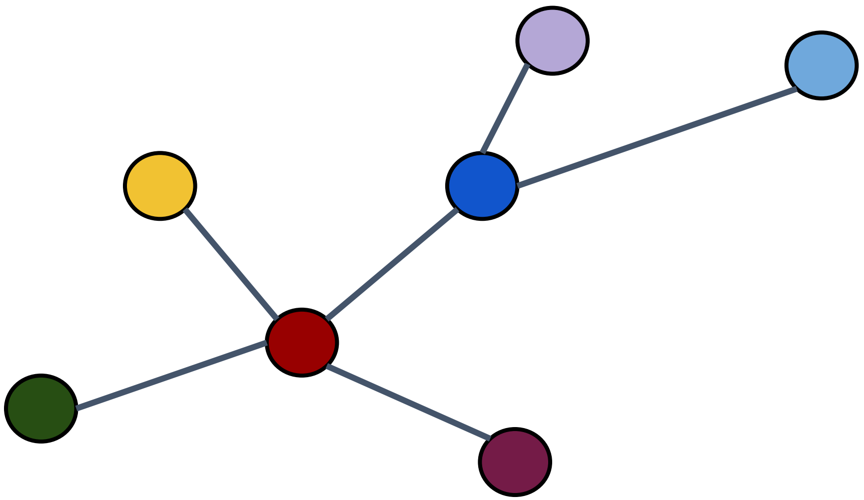
access Rank[j]

Cluster Computing on Graphs

Implementation idea:

update ranks in parallel

iterate until converged



Practical challenge:

vertex-degree distributions typically follow power-laws in practice → a few vertices have very high degrees

iterating over neighbors is always going to be slow