Review

Rong Chen

Institute of Parallel and Distributed Systems

Shanghai Jiao Tong University

http://ipads.se.sjtu.edu.cn/rong_chen

Syllabus

Basic Topics

- Sequential/Release
- Eventual/Causal
- Recovery: logging
- Concurrency: 2PL/SI
- Commit: 2PC
- Consensus: Paxos

Distributed Computation

- Optimizing DS
- DFS: NFS/GFS
- Data-parallel: MR/Dryad
- Graph-parallel computation
- AI systems: DP/TP
- AI systems: PP

ACID Transactions

A (Atomicity)	**All-or-nothing** with respect to failures
C (Consistency)	Transactions maintain any internal state **invariants**
I (Isolation)	Concurrently executing transactions don't **interfere**
D (Durability)	Effect to transactions **survive** failures

What is ACID ?

T1 Transfer $1000 from A to B

T2 Transfer $500 from B to A

A T1 **completes** or **nothing** (ditto for T2)

C Preserves **invariants**, e.g. account balance > 0

I No **race**, as if T1 happens either before or after T2

D Once T1/T2 commits, stays done, no updates **lost**

Consistency

What is Consistency?

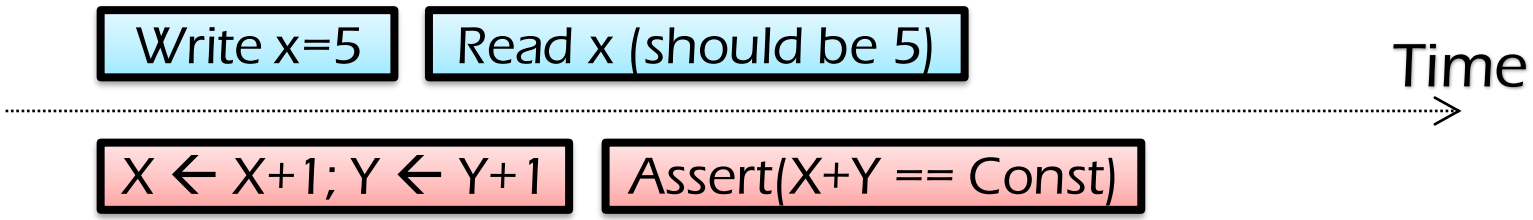
Consistency model defines **rules** for the apparent order and visibility of **updates**, and it is a continuum with **tradeoffs**

-- Todd Lipcon

Examples

Single object consistency is also called "**coherence**"

Local memory:



Database:

Consistency across multiple objects (**all** or **nothing**)

Consistency Challenges

No right or wrong consistency models

- Tradeoff between ease of programmability and efficiency
  (*e.g.*, what's the consistency model for web pages?)

Consistency is hard in (distributed) systems

- Data replication (Caching)
- Concurrency (no shared clock)
- Failures (machine or network)

Sequential Consistency

Strict consistency is not practical

- No global wall-clock available

Sequential consistency is the closest

Consistency rules

"global wall-clock" → "total order" of ops

1. Each CPUs' ops appear in order
2. All CPUs see results according to total order (*i.e.*, reads see most recent writes)

Sequential consistency is also intuitive results (just use the **total order** as **timestamp**)

Release Consistency

Introduce a special type of variables, called synchronization variables or locks

Locks can be acquired/released, not read/written

- Denoted by **acquire(L)** and **release**(L)

No more than one process can hold a lock L

- Hold a lock
  → acquired a lock and has not released it

K. Gharachorloo, et al. *Memory consistency and event ordering in scalable shared-memory multiprocessors*. ISCA, 1990

Sequential vs. Eventual

Sequential: pessimistic conflict handling

- Conflicting writes is common case
- Updates cannot take effect unless they are serialized **first**

Eventual: optimistic conflict handling

- Conflicting writes is rare case
- Let updates happen, worry about whether they can be serialized **later**

Eventual Consistency

For better performance

- Accept a write before being able to serialize it
- Reads can return a (possible) stale value than blocking for the latest value

Anomalies

- Write-write conflict
- Stale read (Loss of causality)

Eventual **convergence** of state

Causality preserving


Atomicity

All-or-nothing Atomicity

All-or-nothing

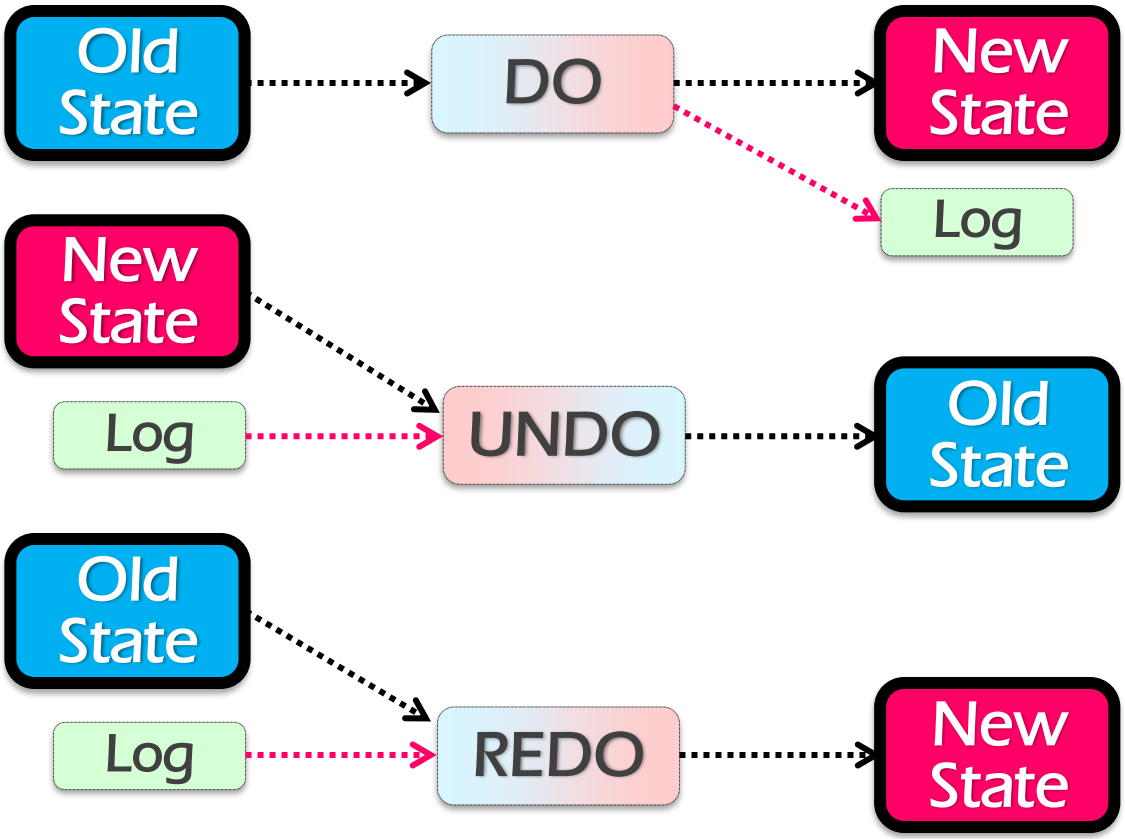
- A set of operations either all finish or none at all
- No intermediate state exist upon recovery

All-or-nothing is one of the guarantees offered by database transactions

Solution: Logging

Keep a log of all update actions Log

Each action has 3 required operations



Logging

Why records both **UNDO** and **REDO** logs

- A transaction might be very long
  → Must checkpoint w/ ongoing transactions
- A long transaction might be aborted
  → Must UNDO abort state when recovery

Why we need **checkpoint**

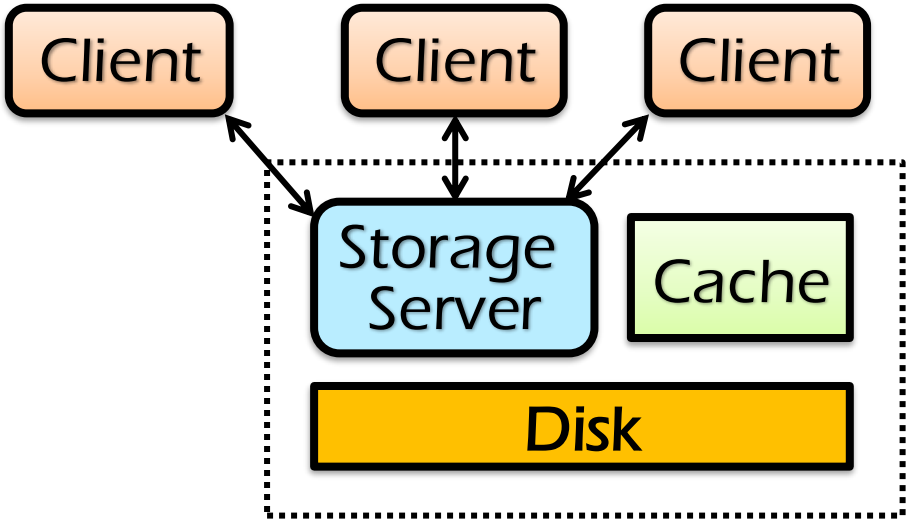
- Avoid aborting long transaction
- No need to recovery from a blank state
- . . .

Isolation

Concurrency Control

Concurrent transactions

- Multiple application clients are concurrently accessing a storage system
- Transactions might interfere
- Similar to data race in parallel programs



Two-phase Locking

Two-phase locking (**2PL**)

- A growing phase in which the transaction is acquiring locks
- A shrinking phase in which locks are released

In practice

- The growing phase is the entire transaction
- The shrinking phase is at the **COMMIT** time

Optimization

- Use read/write locks instead of exclusive locks

Multi-version CC

No locking during transaction execution

Each data item has multiple versions

Multi-version transactions:

- Buffer **WRITE**s during execution
- **READ**s choose the appropriate version
- At **COMMIT** time, system validates if OK to make writes visible (generating **new** versions)

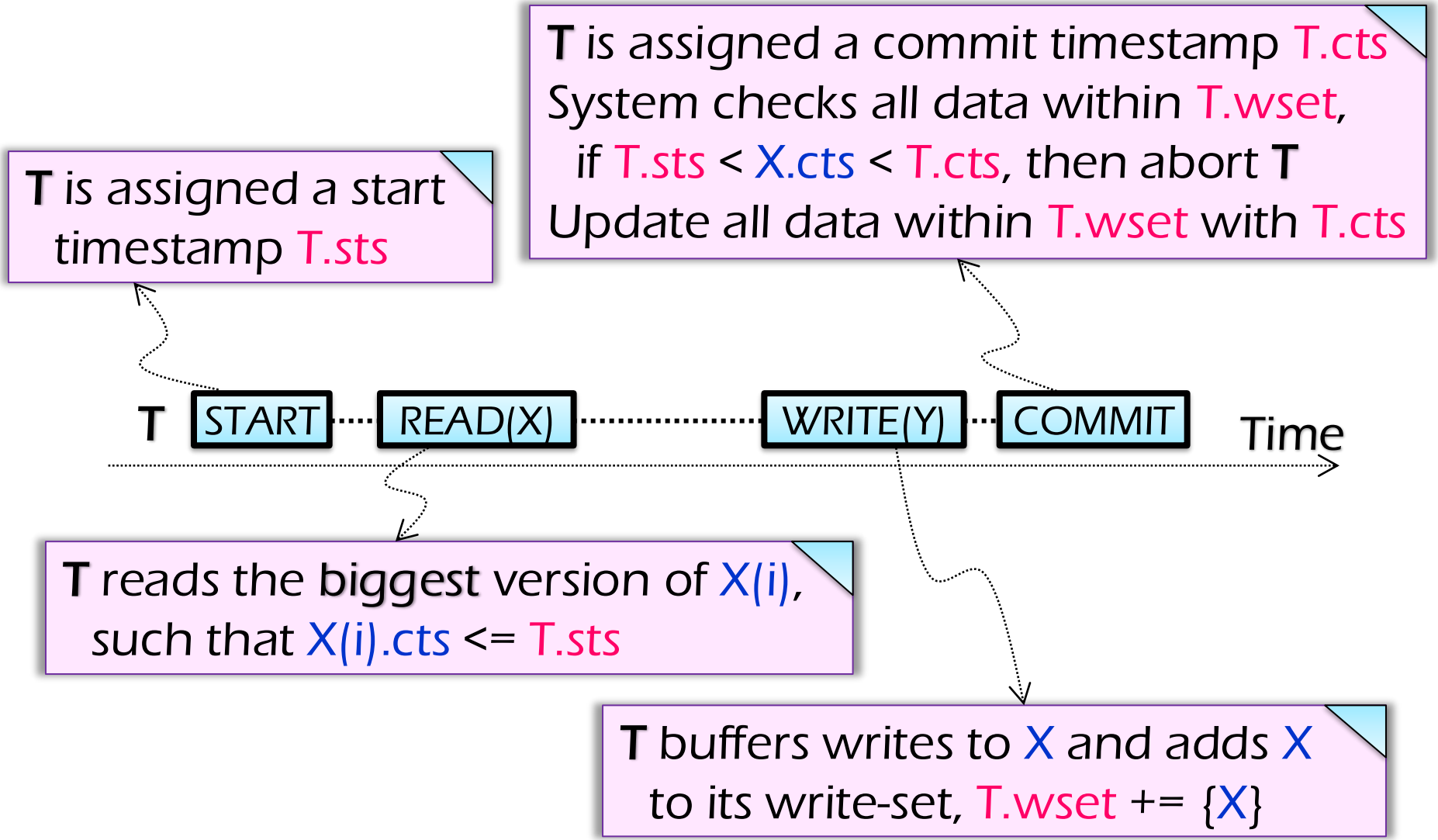
Snapshot Isolation

A popular multi-version concurrency control (MVCC) scheme

Transactions:

- **WRIT**s a lock buffer
- **READ**s a "snapshot" of entire data image
- **COMMIT** only if no write-write conflict

Snapshot Isolation



Distributed Commitment

Two-phase Commit (2PC)

The commit-step itself is two phase

Phase-1: Voting

- Each participant prepares to commit, and votes on whether or not it can commit

Phase-2: Committing

- Each participant actually commits or aborts

The Voting Phase

TC asks each **participant** canCommit(T) ?

Participants must prepare to commit using permanent storage before answering YES

- Objects are still locked
- Once a **participant** votes "**YES**"
  → it is not allowed to cause an **ABORT**

Outcome of **T** is uncertain until doCommit(T) or doAbort(T)

- Other **participants** might still cause an **ABORT**

The Committing Phase

TC collects all votes

- If unanimous "**YES**", cause **COMMIT**
- If any **participant** voted "**NO**", cause **ABORT**

The fate of the **T** is decided atomically at the **TC**, once all **participants** vote

- **TC** records fate using permanent storage
- Then broadcasts *doCommit(T)* or *doAbort(T)* to **participants**

Distributed Consensus

Consensus Goal

Create an algorithm that does not block if a majority of processes are working

Goal: agree on one result among a group of participants

- Nodes may fail (may need stable storage)
- Messages: lost, out of order, or duplicated
- If delivered, messages are not corrupted

The algorithm needs (**2P**+**1**) nodes survive the simultaneous failures of **P** nodes

Paxos

Paxos: fault-tolerant distributed consensus algorithm (the **only known**)

- All nodes agree on the same value despite node failure, network failure and delays

General Approach

- One **proposer** decides to be the leader
- Leader proposes a value and solicits acceptance from **acceptors** (majority)
- Leader announces result or try again

Paxos Pseudo-code

A node decides to be Leader

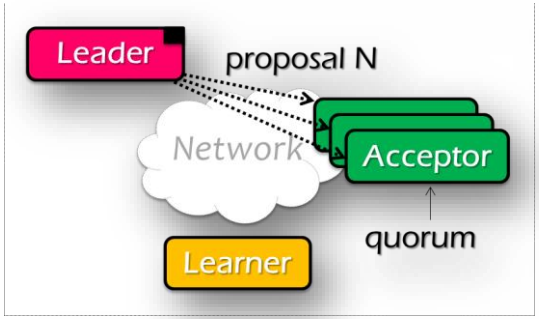
Leader choose $M_n > N_h$

Leader sends <proposal, M_n> to all nodes

Acceptor receives <proposal, N>

```
if N < Nh
    reply <promise-reject>
else
    Nh = N
    reply <promise-ok, Na, Va>
```

Ignore all proposals < N_h



Paxos Pseudo-code

If Leader gets promise-ok from a majority

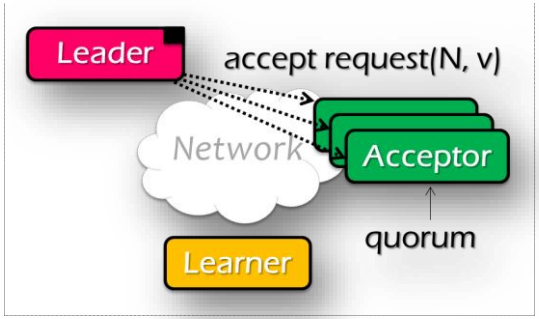
V = non-empty value (highest N_a received)
if **V = null**, then Leader can pick any **V**
send <accept, M_n, **V**> to all nodes

If Leader fails to get majority promise-ok

delay and restart Paxos

Upon receiving <accept, **N**, **V**>

if $N < N_h$
    reply <accept-reject>
else
    $N_a = N$; $V_a = V$; $N_h = N$;
    reply <accept-ok>



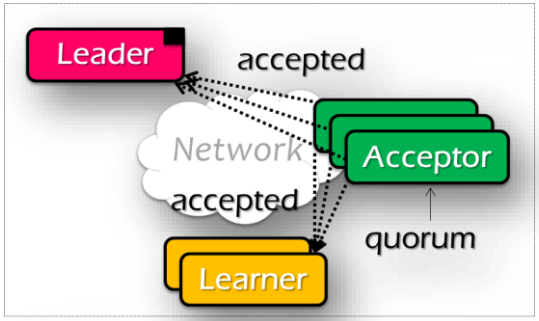
Paxos Pseudo-code

If Leader gets accept-ok from a majority

send <decide, V_a> to all nodes

If Leader fails to get majority accept-ok

delay and restart Paxos



Optimizing DS w/ New HD

HTM: Intel Restricted TM

Restricted Transactional Memory (RTM)

- Hardware transactional memory with limitations

Major limitations

- Working set is limited
- System events abort TX

New instruction set

- Xbegin, Xend, Xabort

```
RTM Usage
  if _xbegin() == _XBEGIN_STARTED
       do some critical work
      _xend()
  else
      fallback routine
```

Handle the abort event

RDMA: Remote Direct Memory Access

A network feature that allows **direct** access to the memory of a **remote** computer

High speed, low latency & low CPU overhead

- Interface: IPoIB, two-sided RDMA (SEND/RECV), and one-sided RDMA (READ/WRITE/CAS)
- Bypasses OS kernels: Zero copy
  Bypasses CPU (one-sided)
- Round-trip time: one-sided/~3μs,
  two-sided/~7μs, IPoIB/~100 μs

Opportunities: (not so) New HW Features

HTM: **H**ardware **T**ransaction **M**emory

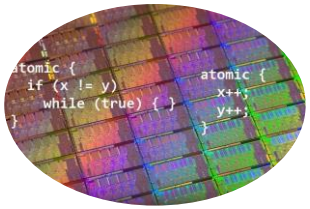
- Allow a group of load & store instructions to execute in an atomic, consistent and isolated (ACI) way

RDMA: **R**emote **D**irect **M**emory **A**ccess

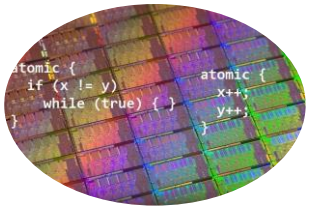
- Provide cross-machine accesses with high speed, low latency and low CPU overhead

Rethink the design of low-COST scalable in-memory transaction systems

RDMA-friendly
Distributed Key-value Store



Fast Distributed Transactions using RDMA & HTM



Distributed Storage

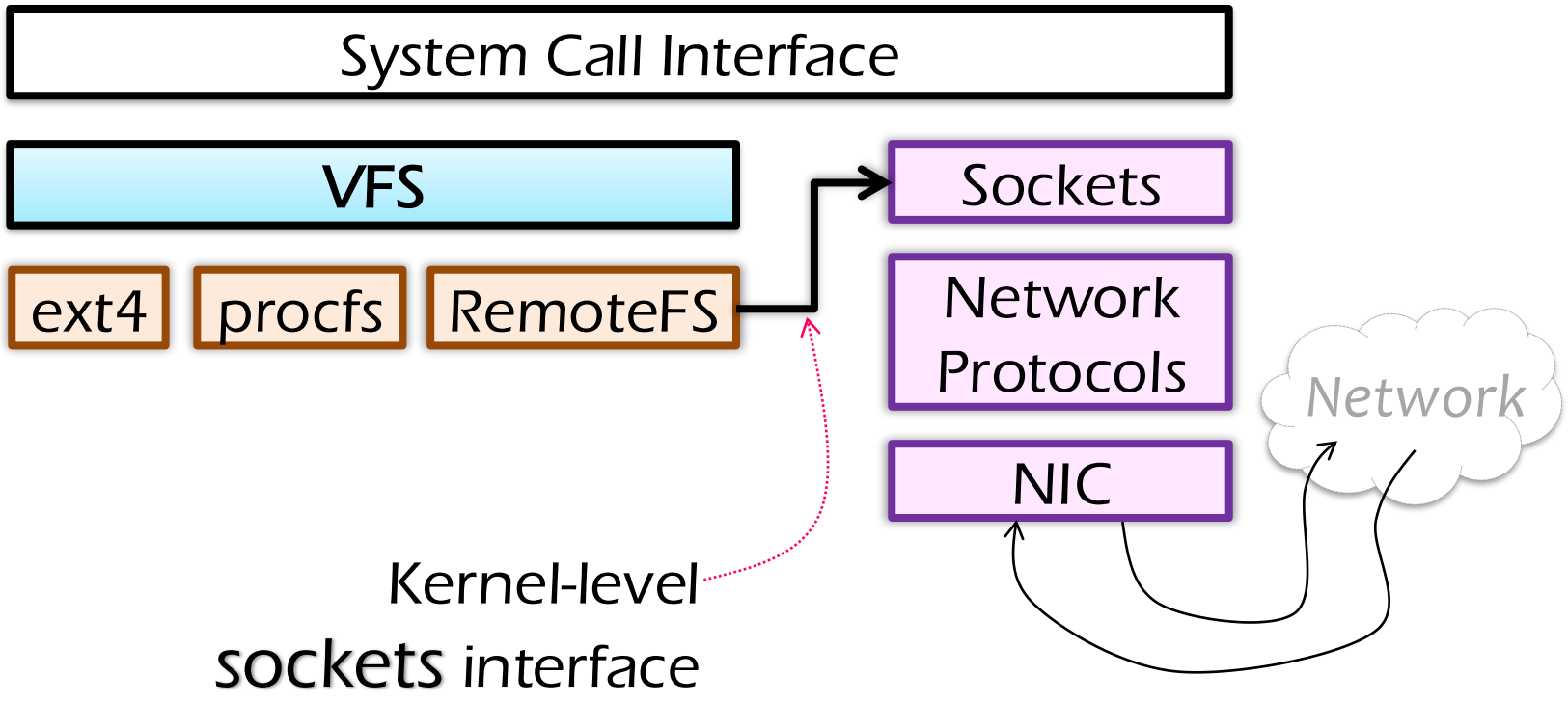
NFS: Network File System

Design Goals

- Any machine can be a client or server
- Must support diskless workstations
- Heterogeneous system must be supported
  - Different HW, OS, underlying file system
- Access transparency
- Recovery from failure
  - Stateless, UDP, client retries
- High performance
  - Use caching and read-ahead

Accessing Remote Files

Implement the client module as a FS under VFS



GFS Goals

Scalable distributed file system

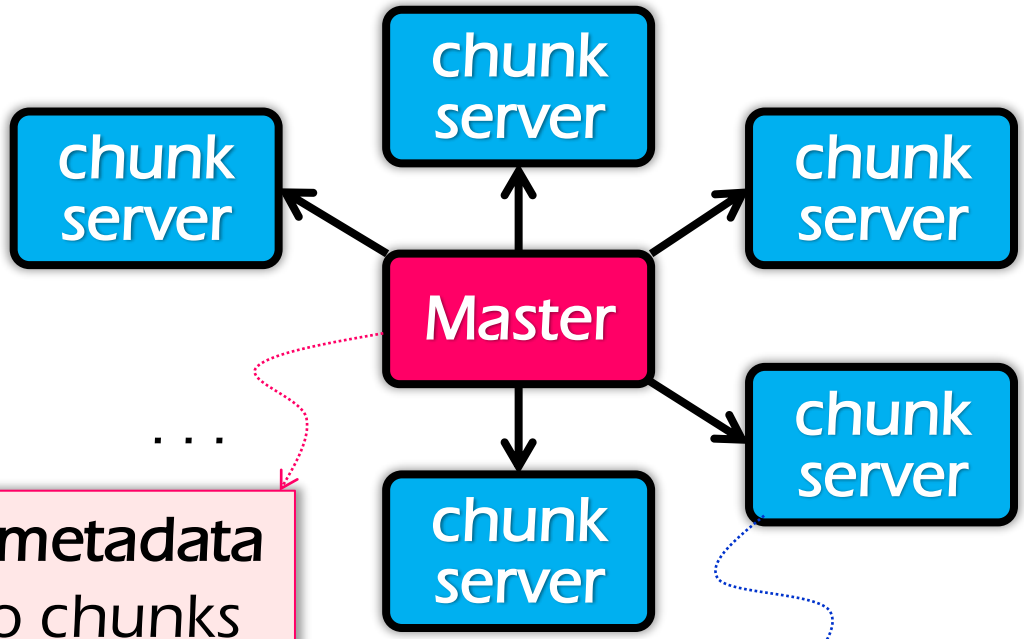
Designed for large data-intensive applications

Fault-tolerant; runs on commodity hardware

Delivers high performance to a large number of clients

GFS Servers

GFS cluster: *N* **Chunkservers** + *1* **Master**



File system **metadata**
Maps files to chunks

Data Storage: fixed-size chunks
Chunks replicated on several system

Distributed Computation

Assumption for Data-parallel

No dependency among data

Data can be split into equal-size chunks

Each process can work on a chunk

Master/worker approach

- Master
  1. Initialize array and splits it according to # of workers
  2. Send each worker the sub-array
  3. Receive the results from each worker
- Worker
  1. Receive a sub-array from master
  2. Perform processing
  3. Send results to master

MapReduce Programming Model

Allows user to process *huge* amounts of data on *thousands* of nodes

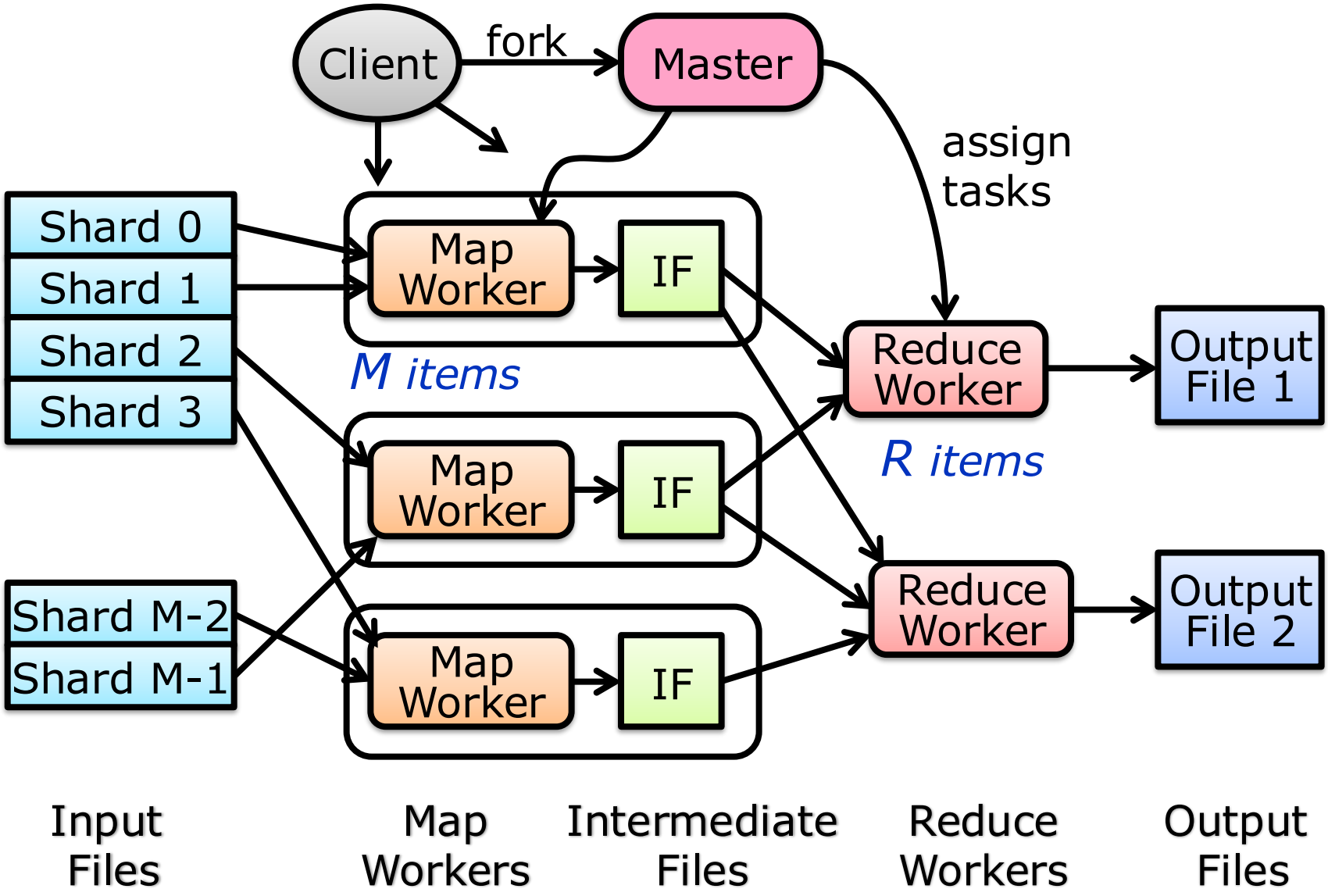
Framework for data-parallel computing

Programmers get **simple** API

Don't have to **worry** about handling

- ~~Parallelization~~
- ~~Data distribution~~
- ~~Load balancing~~
- ~~Fault tolerance~~

The Complete Picture



Issues

Locality

Fault tolerance

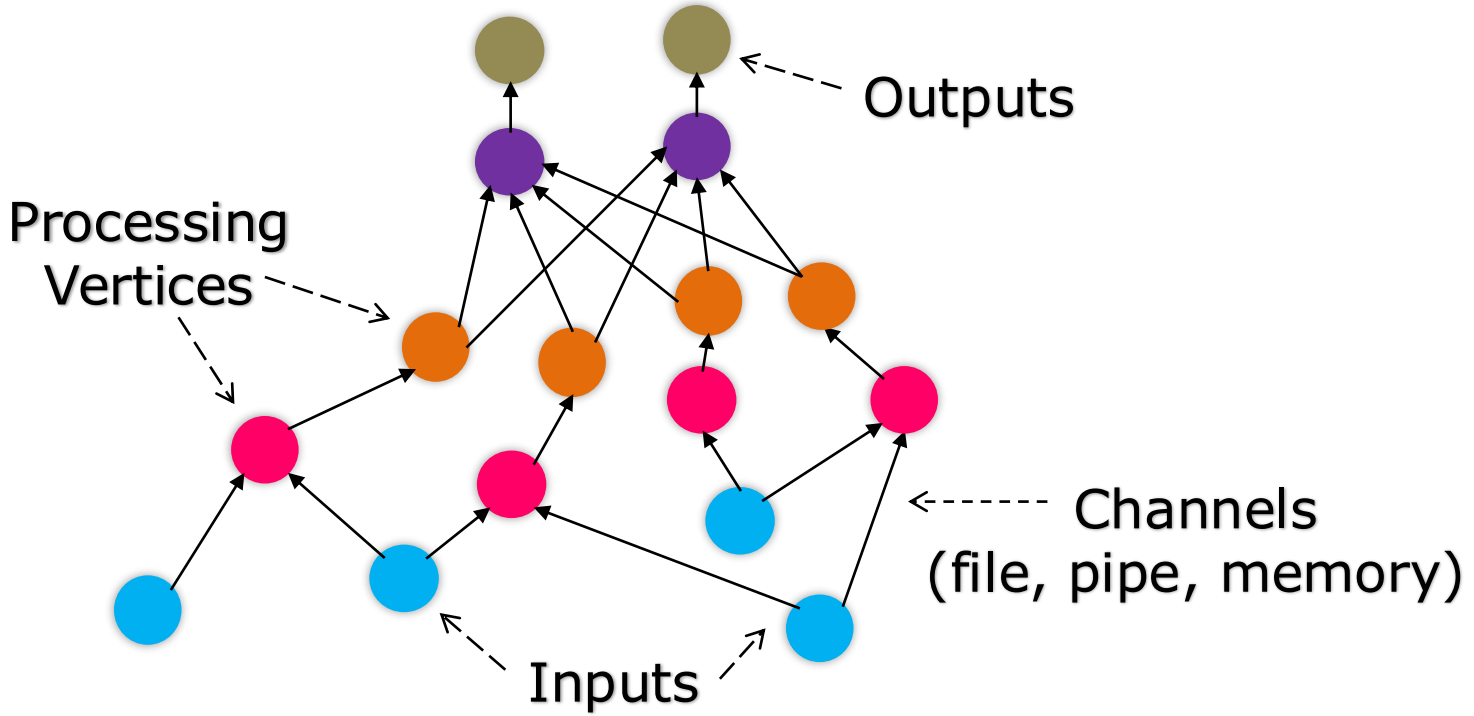
Straggler

. . .

Dryad: DAG-based Computation

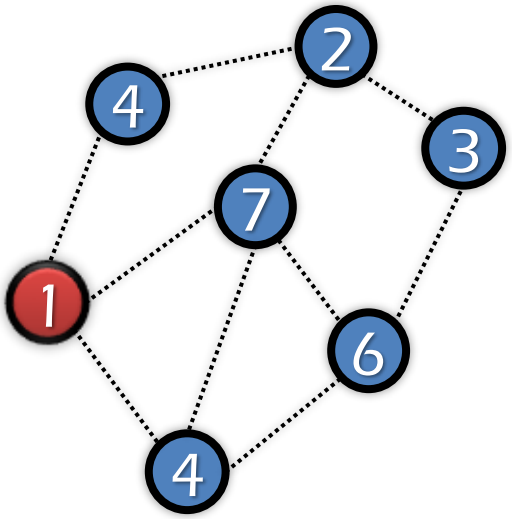
Computations expressed as a graph

- Vertices are computations
- Edges are communication channels

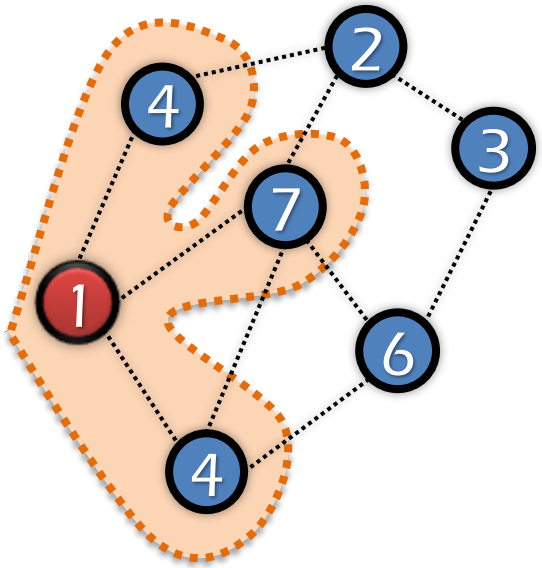


Graph-parallel Computation

Graph Parallel Algorithm



Dependency
Graph



Local
Updates

Iterative
Computation

Why not MapReduce?

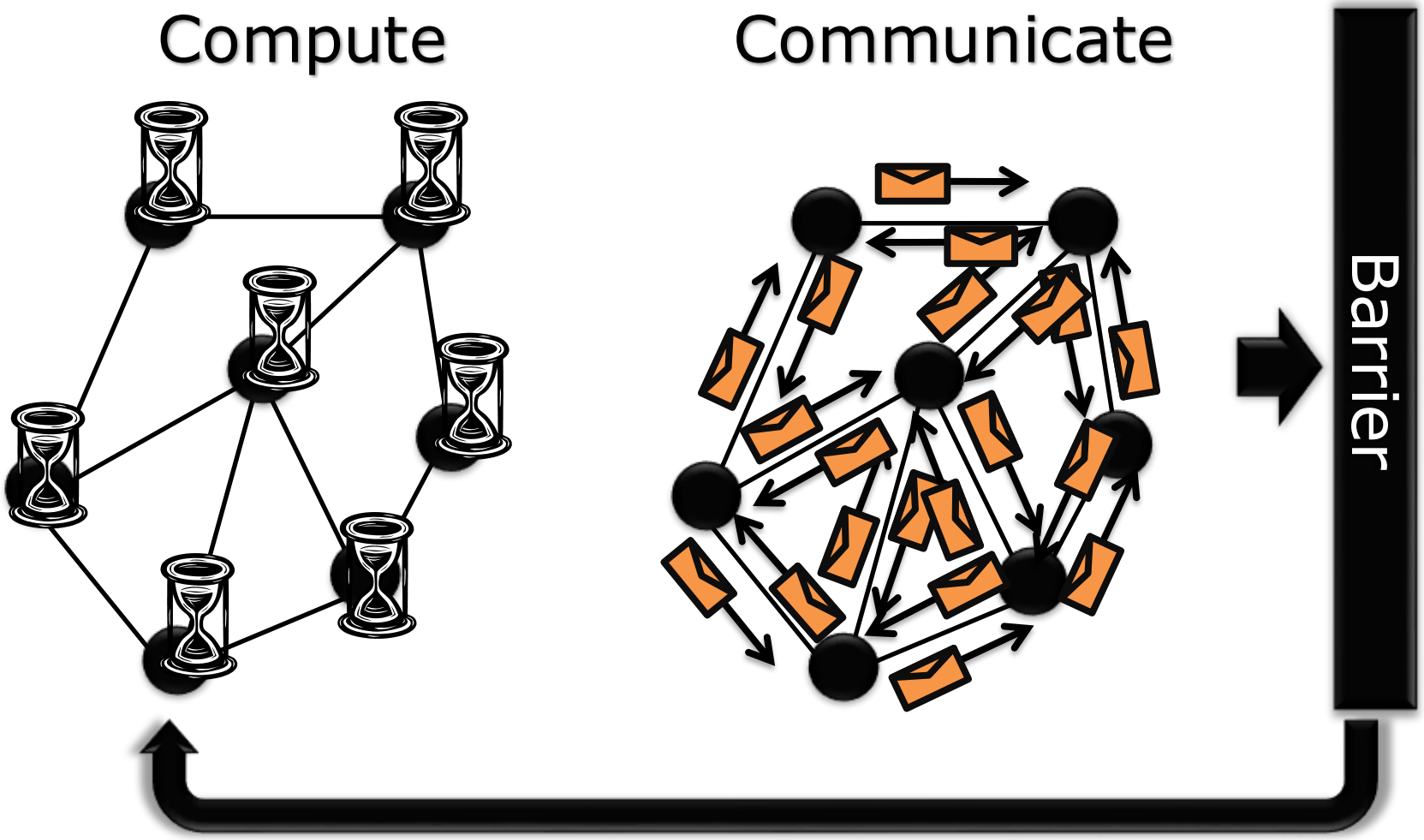
Difficult to express dependent data in MR

- Substantial data transformations
- User managed graph structure
- Costly data replication

MR runtime is not optimized for iteration

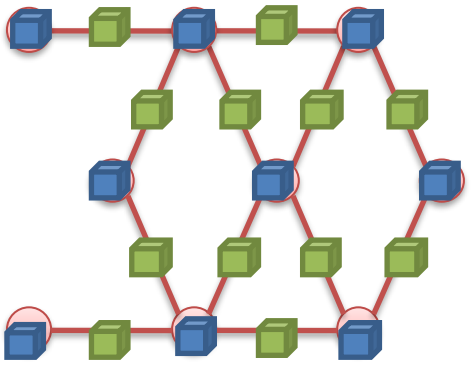
- Persistent I/O (e.g., GFS)

BSP Model

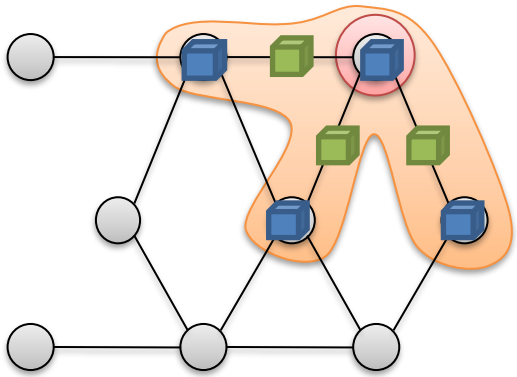


GraphLab Framework

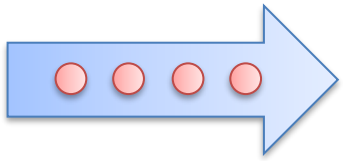
Graph Based
Data Representation



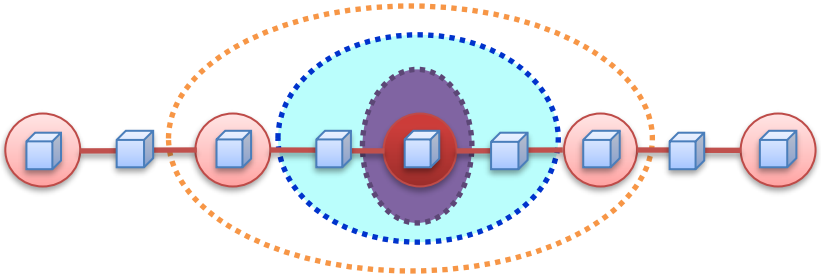
Update Functions
User Computation



Scheduler

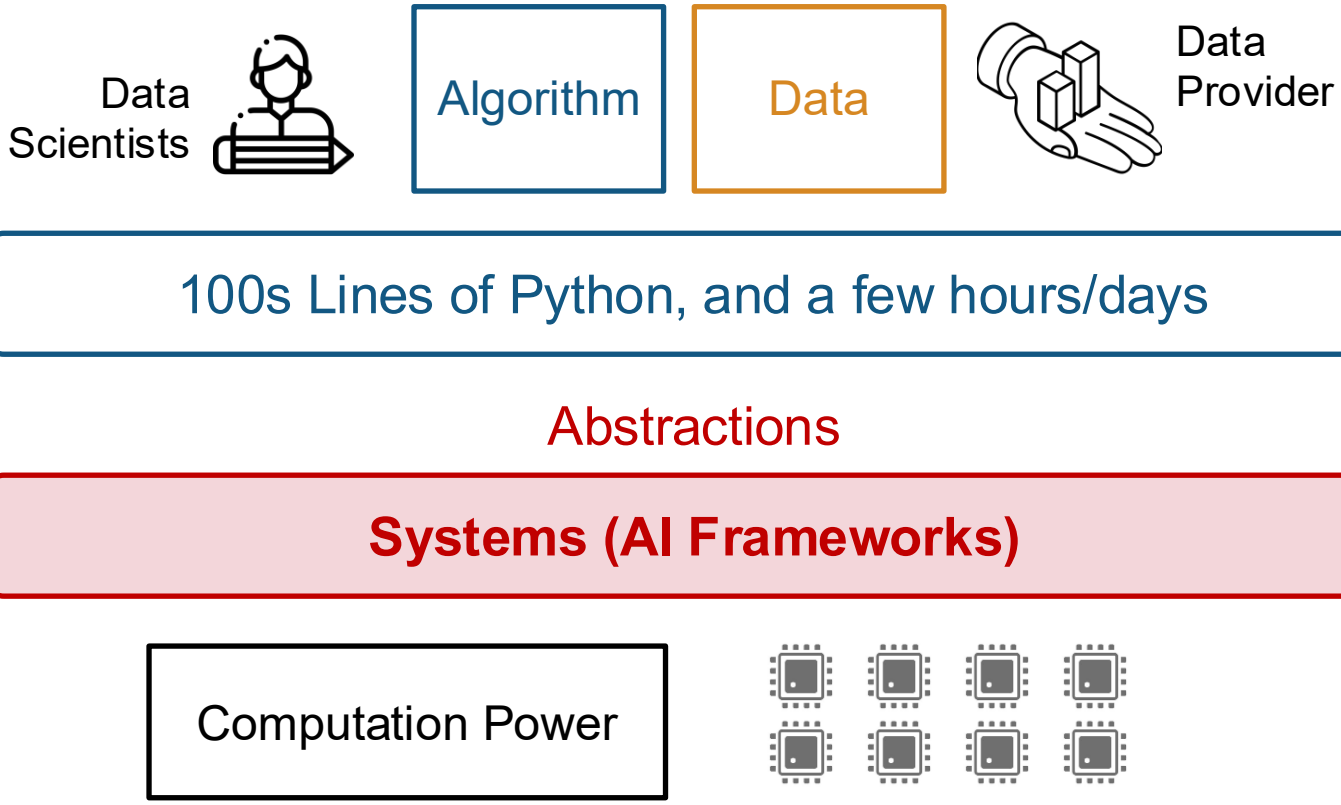


Consistency Model



Parallelizing AI systems

AI Systems



Credit: “Machine Learning Systems”, 15-884, CMU

Parallelizing AI Systems

Similar characteristics in AI systems

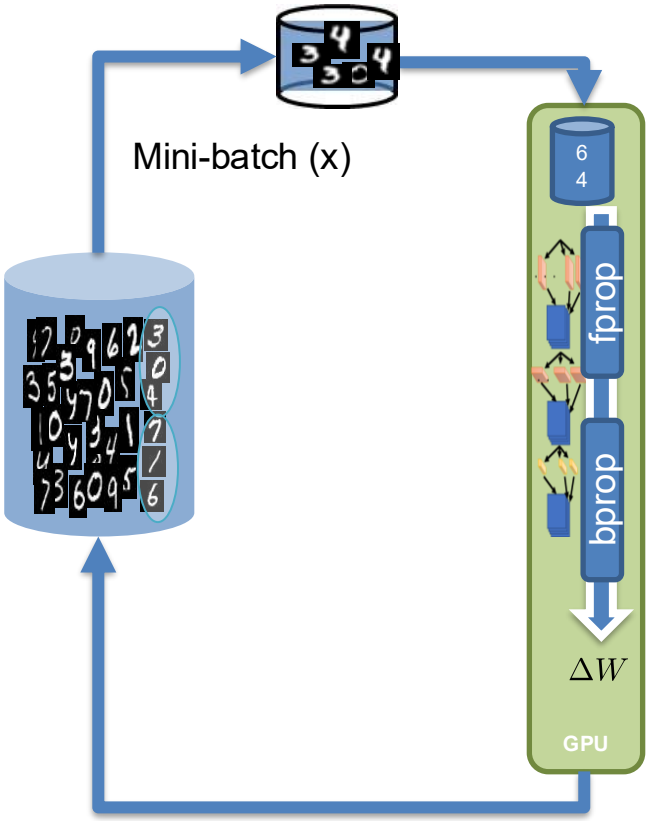
- Massive data and resources
- Data dependency and update synchronization
- Iterative convergence

New characteristics in AI systems

- **Compute-intensive**: large models (parameter scale: ~10^9)
- **Execution flow**: back-propagation, selective (MoE, sparse-activation)
- **New resources**: GPU and accelerators, NVLink/NVSwitch

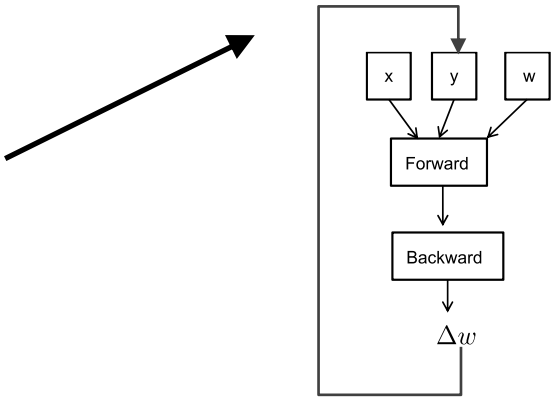
Overview of Model Training in Action

In every iteration of SGD we load a random mini-batch of training data and compute gradient



$$\min_w \mathcal{J}(w) = \frac{1}{N}\sum_{i=1}^{N} cost(w, x_i)$$

$$w^1 = w^0 \underbrace{- \frac{\alpha}{B}\sum_{i=1}^{B} \frac{\partial \mathcal{J}(w^0)}{\partial w}}_{\Delta w}$$



Parallelization Opportunities

Data Parallelism: *Distribute the processing of data to multiple PEs.*

$$w^1 = w^0 - \frac{\alpha}{B} \sum_{i=1}^{B} \frac{\partial \mathcal{J}(w^0)}{\partial w}$$

Model Parallelism: *Break the model and distribute processing of every layer to multiple PEs*

$$w^1 = w^0 - \frac{\alpha}{B} \sum_{i=1}^{B} \frac{\partial \mathcal{J}(w^0)}{\partial w}$$

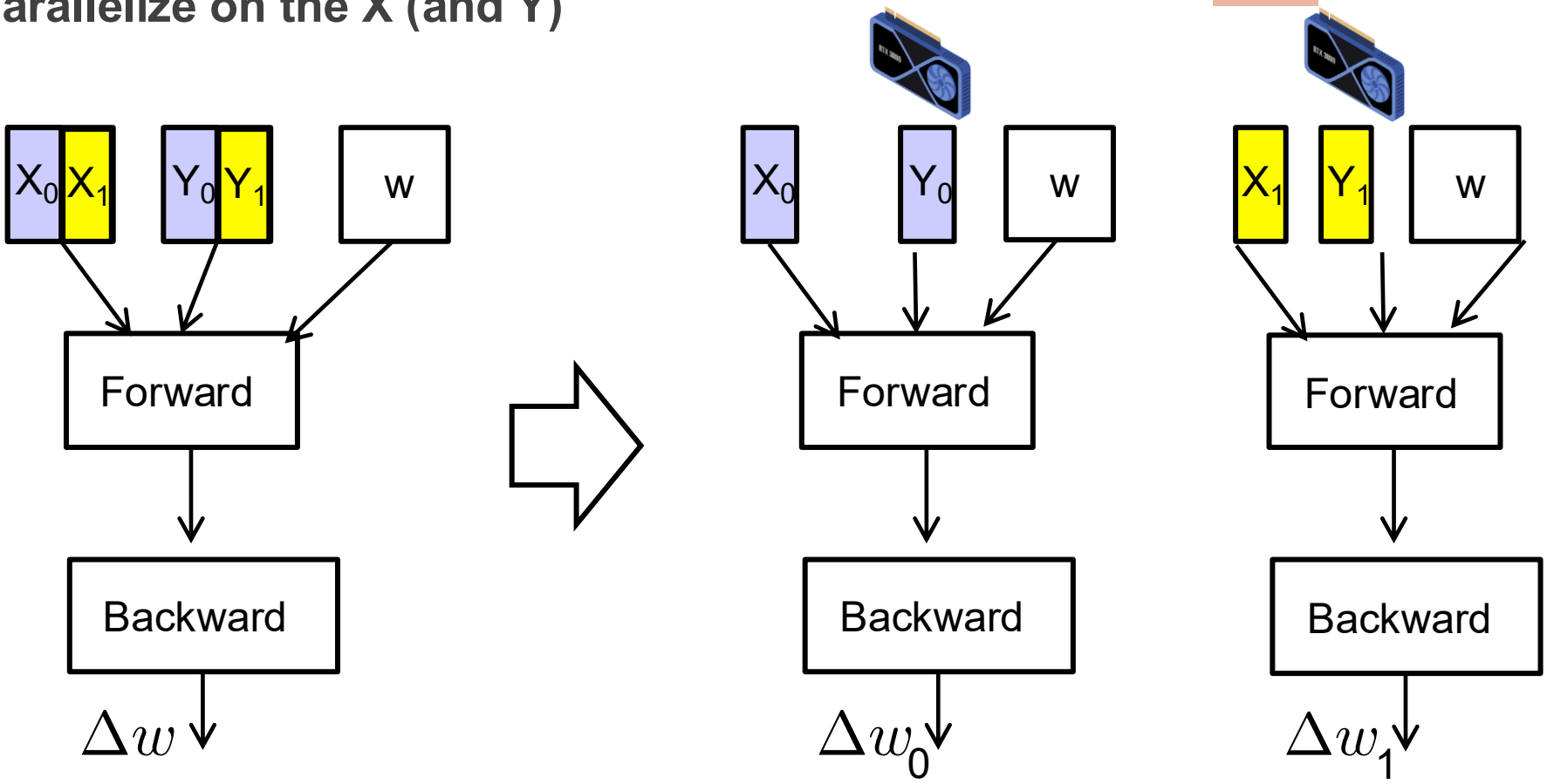
For either approach it is also possible to use ***synchronous*** *or* ***asynchronous*** *updates*

$$w^1 = w^0 - \frac{\alpha}{B} \sum_{i=1}^{B} \frac{\partial \mathcal{J}(w^0)}{\partial w}$$

Data Parallelism

$$w^1 = w^0 - \frac{\alpha}{B} \sum_{i=1}^{B} \frac{\partial \mathcal{J}(w^0)}{\partial w}$$

Parallelize on the X (and Y)



Summary: Data Parallelism

Data Parallelism: Distribute the processing of data to multiple PEs

Pros	Cons
Scalability (Large-scale Data)	Synchronization Overhead
Easy Implementation	Model Duplication

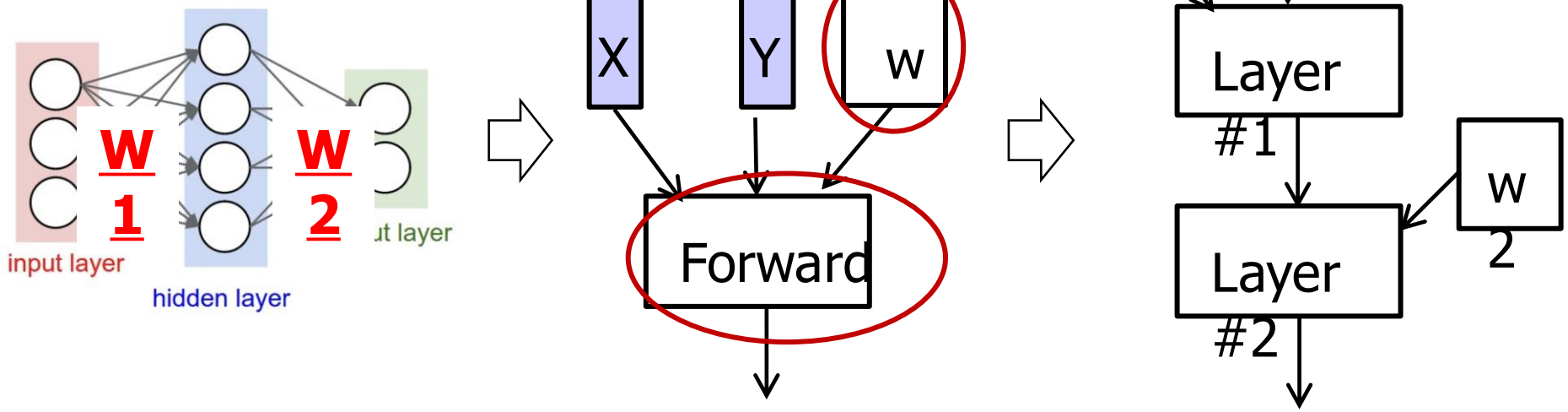
Drawback: model (parameters) are replicated on all PEs

Target: larger dataset but smaller models

Model Parallelism

Partition model parameters, typically done in two ways:

- **Partition on the W**: tensor parallelism
- **Partition on the layer**: pipeline parallelism



Summary: Model Parallelism

Two forms: Pipeline Parallelism + Tensor Parallelism

Pros	Cons
Memory Efficiency	Difficult to Implement

Tensor parallelism: partition the parameters of a layer (or a set of layers)

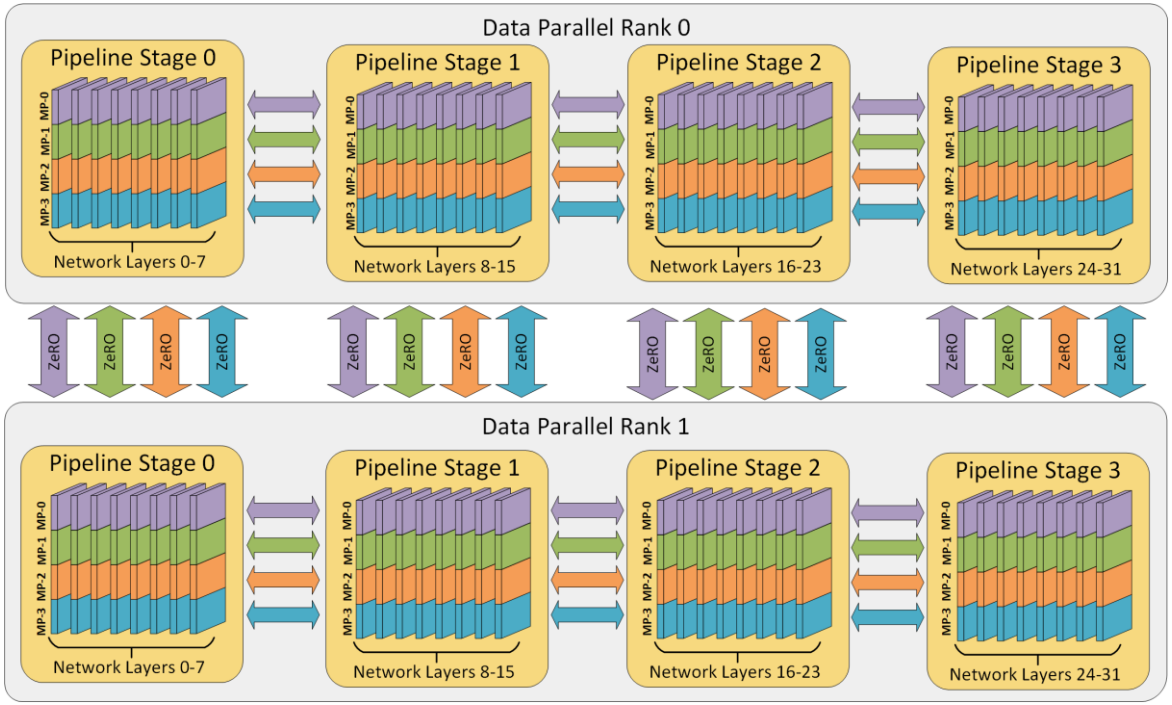
- Pros: better support for large models
- Cons: high communication cost

Pipeline parallelism: partition the computation graph by layers

- Pros: reduced communication
- Cons: bubbles

Put It All Together

It is practical/common to use DP, TP, and PP together (3D or PTD parallelism) to scale to thousands of GPUs



Credit: “System for Machine Learning”, CS229s, Stanford

Final Exam

2026.6.22（下周一）

13:10~15:10

东上院 **107**

半开卷

THANKS