Review

Rong Chen

Institute of Parallel and Distributed Systems

Shanghai Jiao Tong University

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

Syllabus

Distributed Data Store

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

Distributed Computation

- Data-parallel: MR/Dryad
- Graph-parallel processing
- Distributed partitioning
- Disk-based processing
- NUMA-aware processing
- RDMA-based Optimization

You know you have a distributed system when the crash of a computer you've never heard of stops you from getting any work done.

– Leslie Lamport

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:

Write x=5 | Read x (should be 5)

Time >

X ← X+1; Y ← Y+1 | Assert(X+Y == Const)

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**

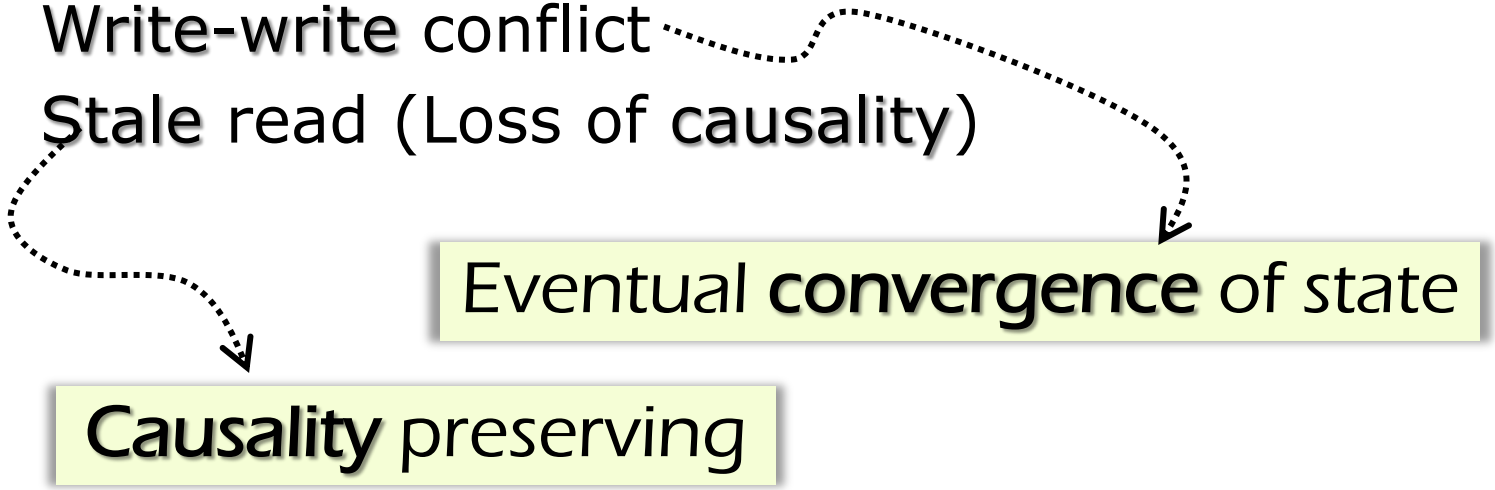
Eventual Consistency

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**)



ACID

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**

All-or-nothing Atomicity

All-or-nothing

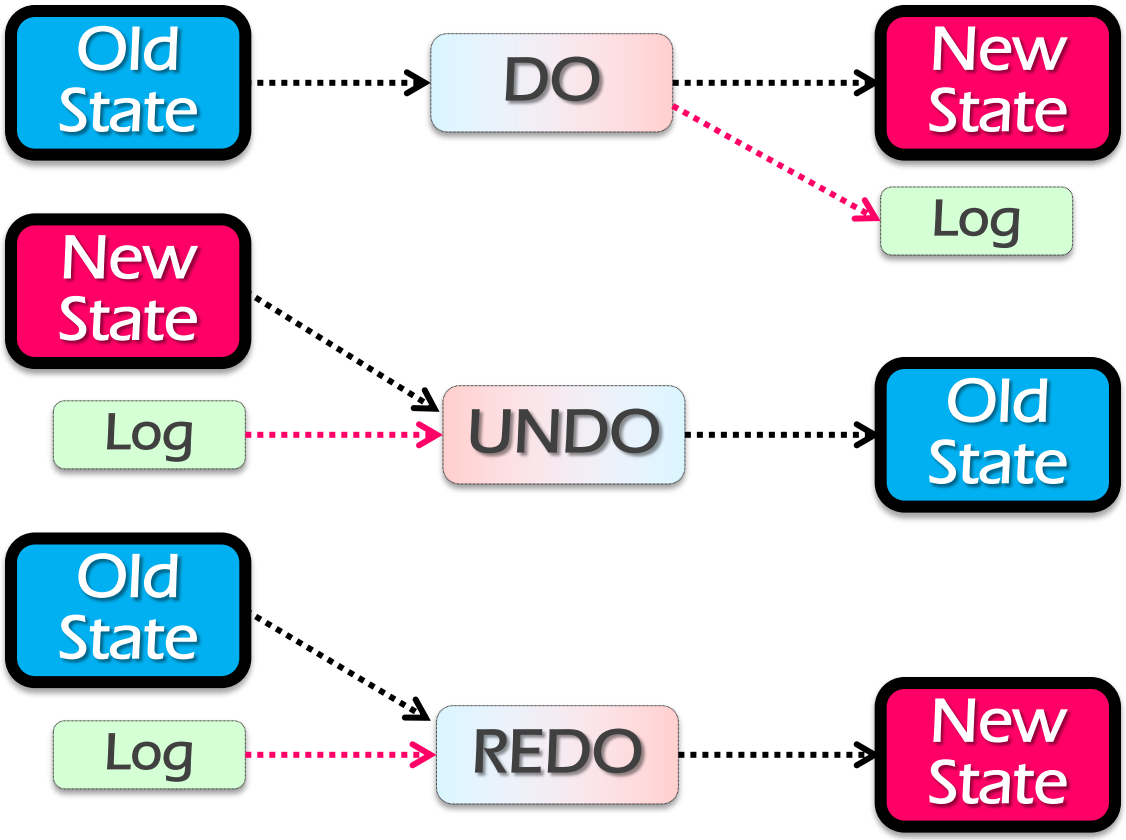
- A set of operations either all finish or none at all
- No intermediate state exist upon recovery

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

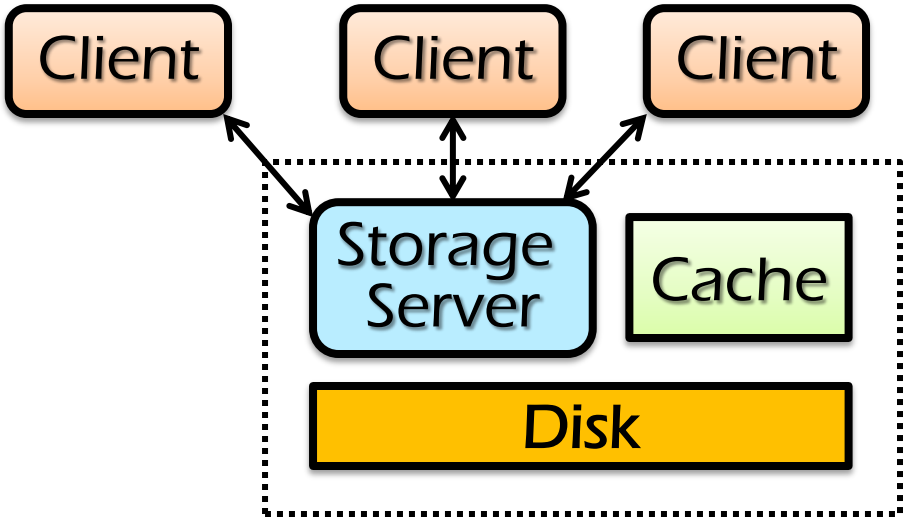
Why we need **checkpoint**

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

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 Concurrency Control

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)

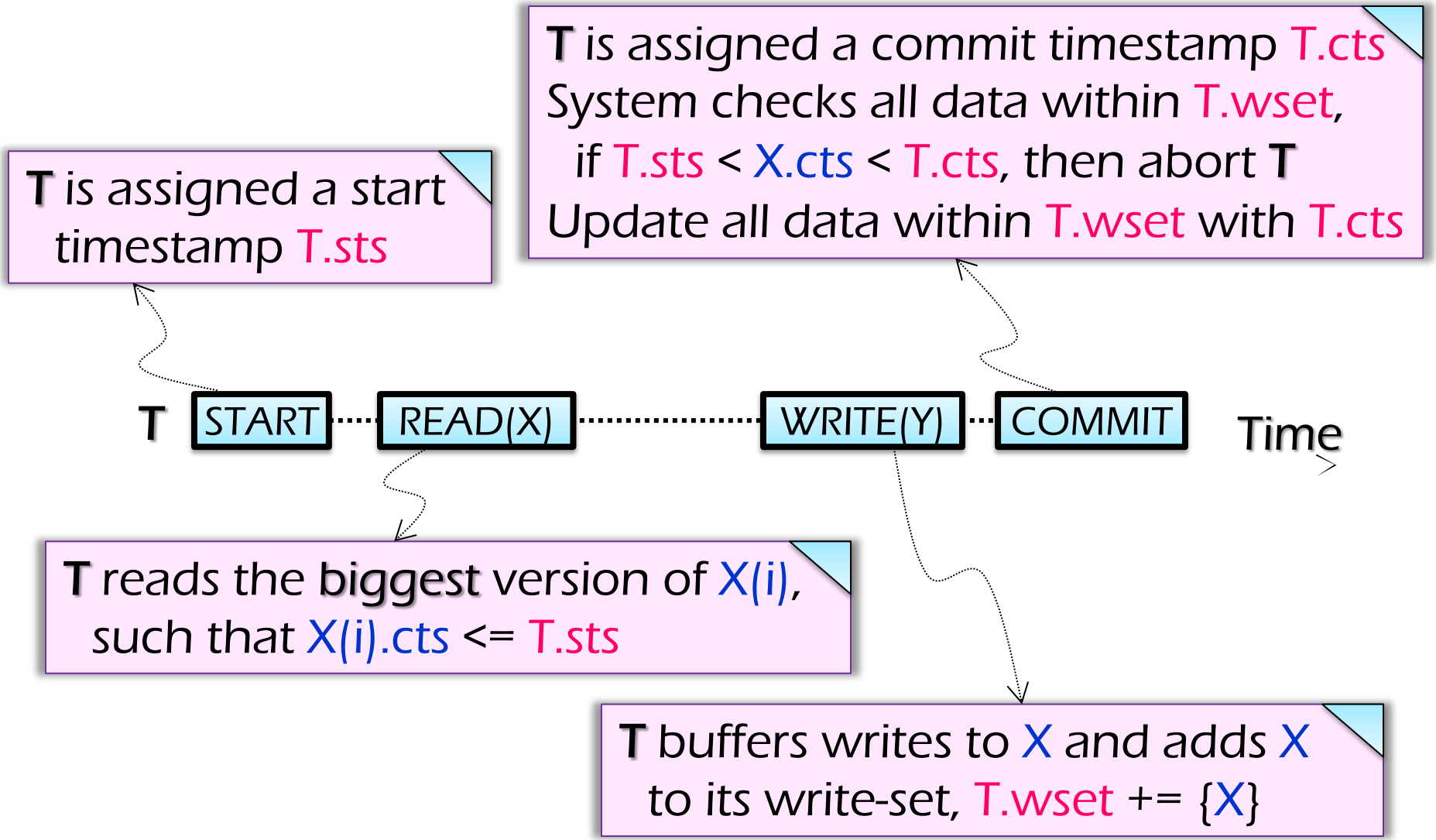
Snapshot Isolation

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



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**

Don't forget about their response after **REBOOT**

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**

Don't forget about its decision after **REBOOT**

RSM: Replicated State Machine

RSM is a general replication method

- e.g., apply RSM to lock service

RSM rules

- All replicas start in the same initial state
- Each replica apply ops in the same order
- All ops must be deterministic
- All replicas end up in the same state

RSM: Failure Handling

If primary fails, one backup acts as the **new** primary

Challenges

1. How to reliably detect primary failure?
2. How to ensure no two backups simultaneously become new primary?
3. How to preserve sequential consistency across primary changes?

A fault tolerant distributed consensus protocol

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

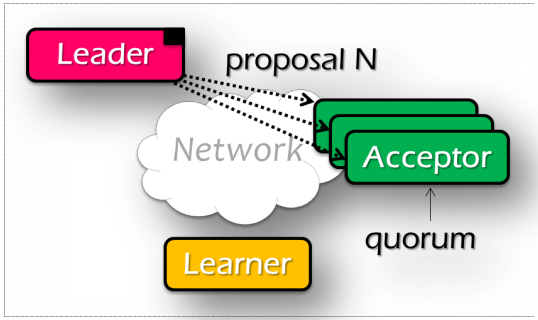
Leader choose $M_n > N_h$

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

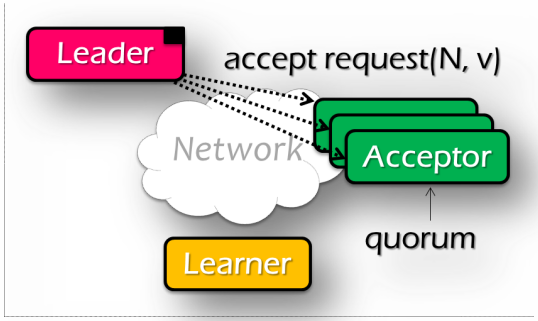
V = non-empty value (highest N_a received)
if V = null, then Leader can pick any **V**
send <accept, M_n, V> to all nodes

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>



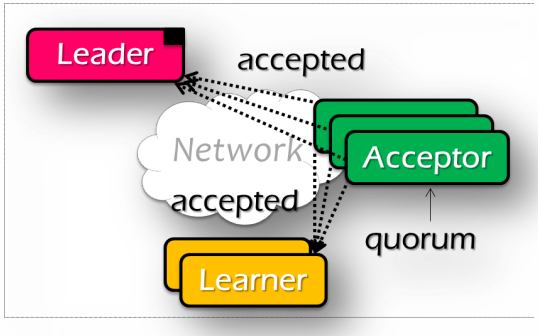
Paxos Pseudo-code

If Leader gets accept-ok from a majority

send <decide, V_a> to all nodes

If Leader fails to get majority accept-ok

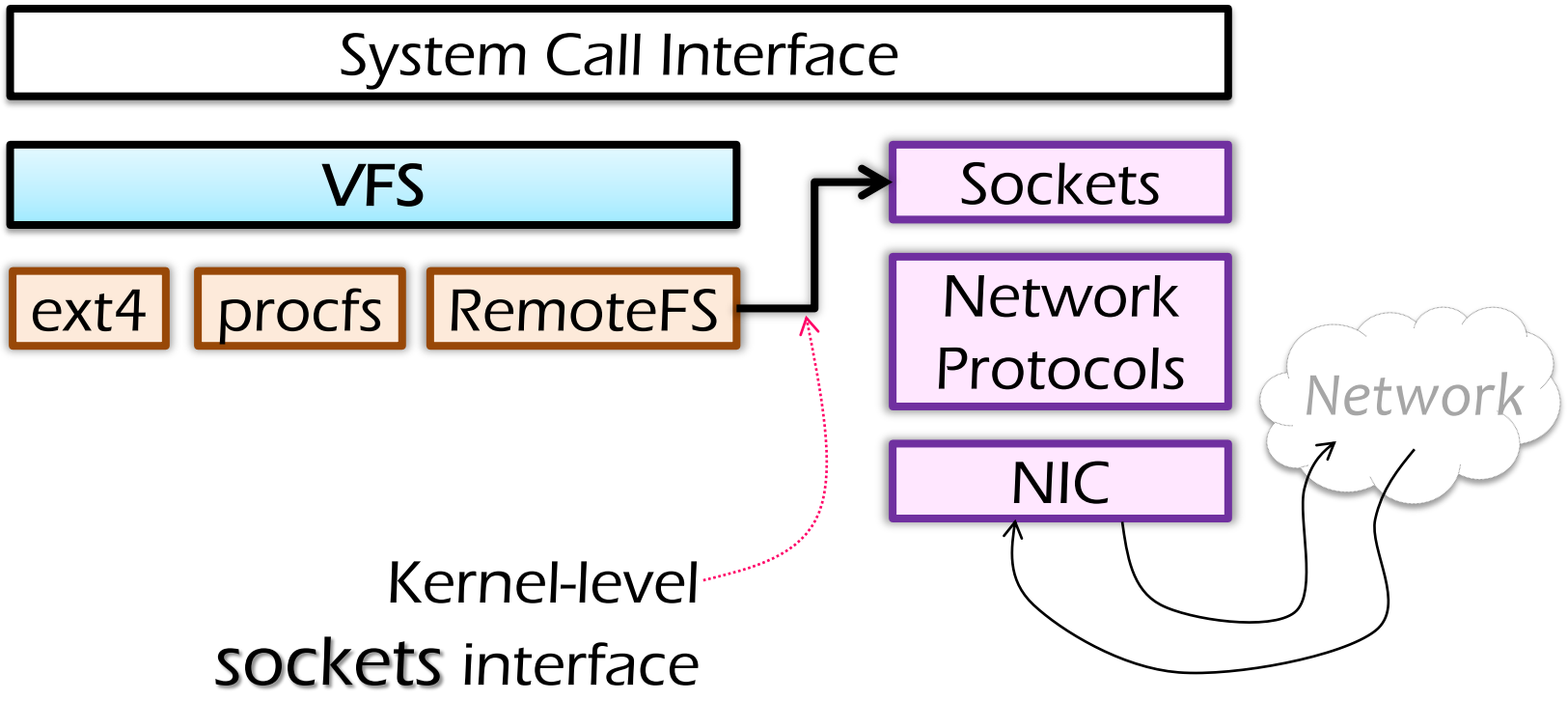
delay and restart Paxos



Distributed Storage

Accessing Remote Files

Implement the client module as a FS under VFS



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

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

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

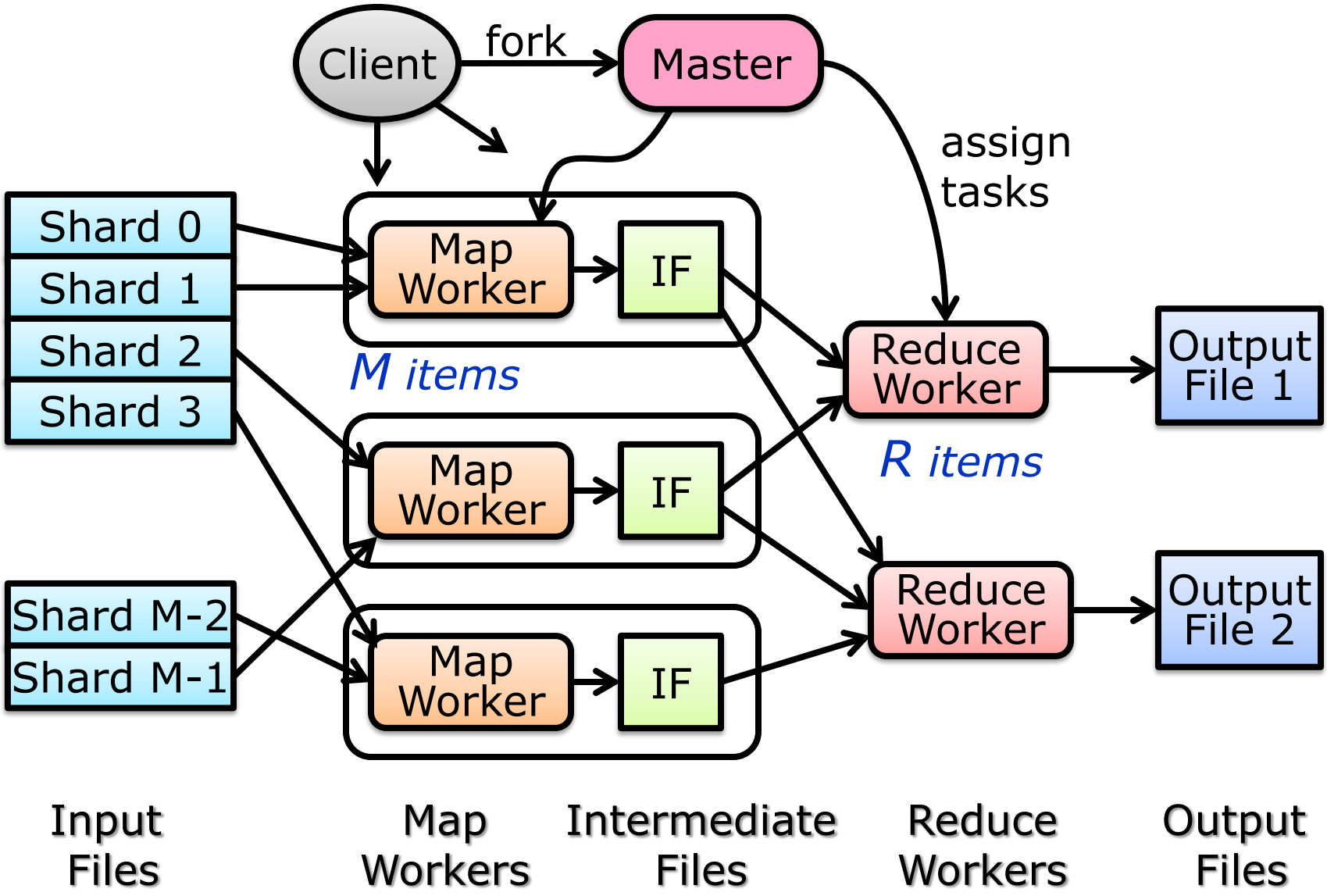
Framework for data-parallel computing

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

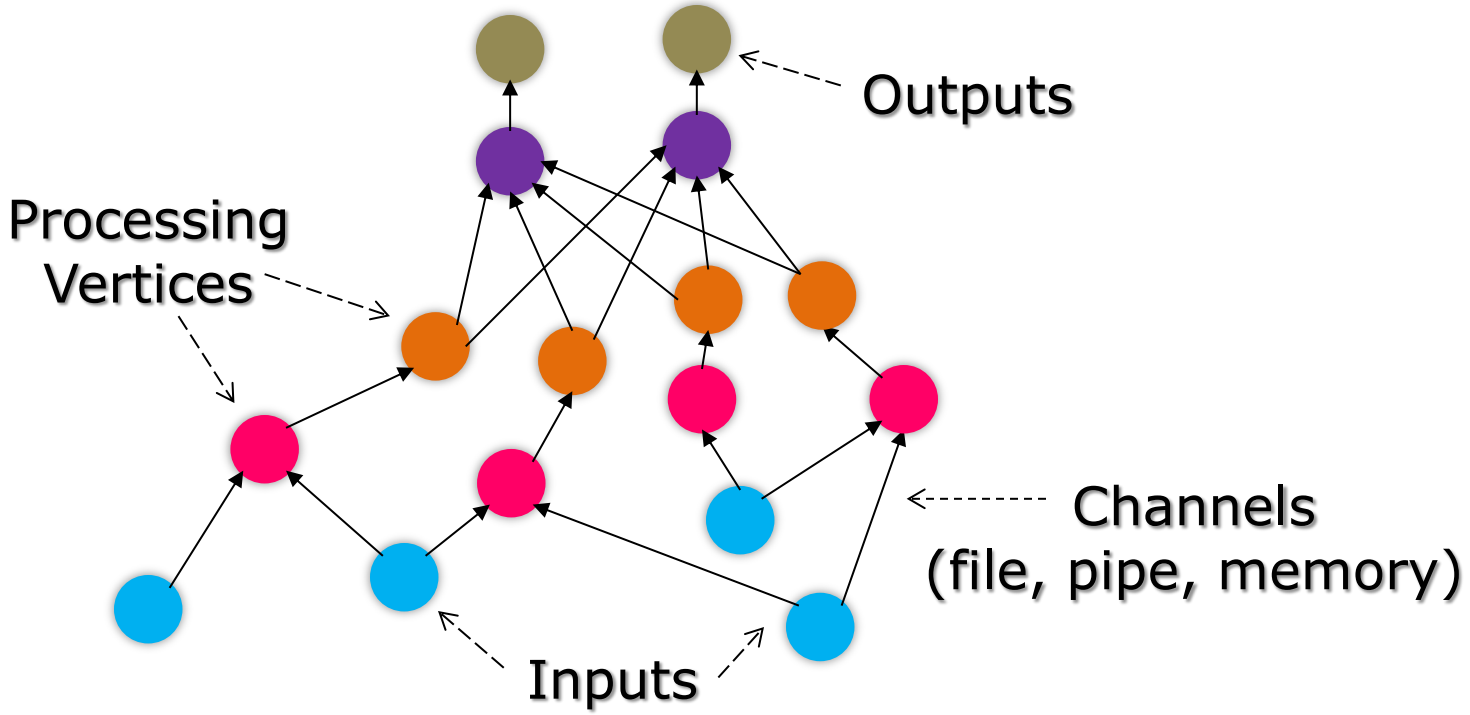
Straggler

. . .

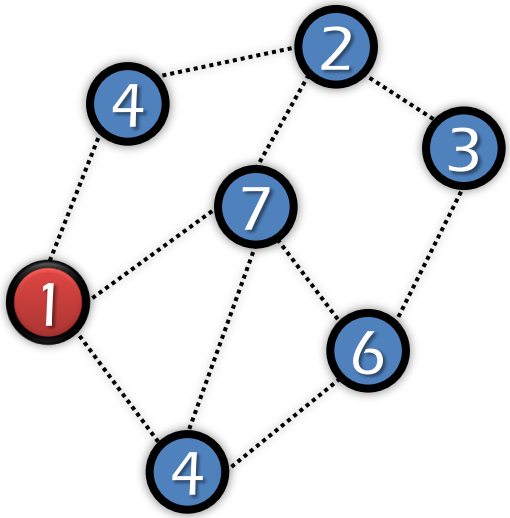
Dryad

Computations expressed as a graph

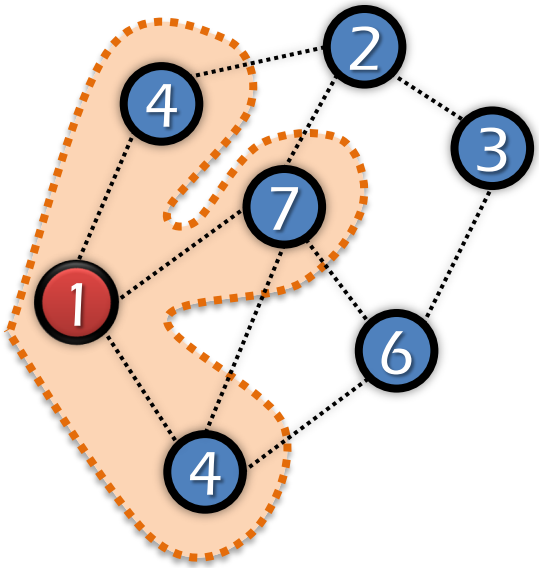
- Vertices are computations
- Edges are communication channels



Graph Parallel Algorithm



Dependency
Graph



Local
Updates

Iterative
Computation

Why not MapReduce?

Difficult to express dependent data in MR

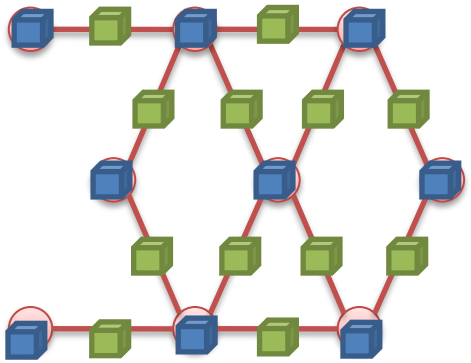
- Substantial data transformations
- User managed graph structure
- Costly data replication

MR runtime is not optimized for iteration

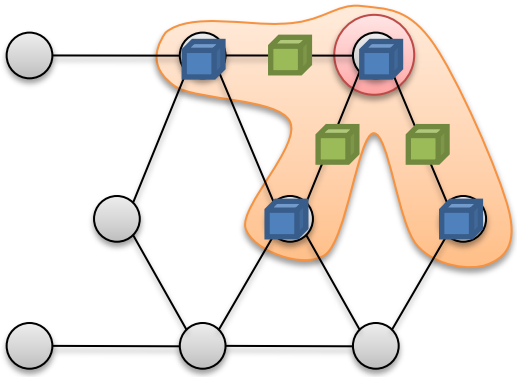
- Persistent I/O (e.g., GFS)

GraphLab Framework

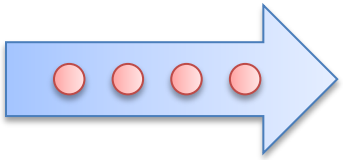
Graph Based
Data Representation



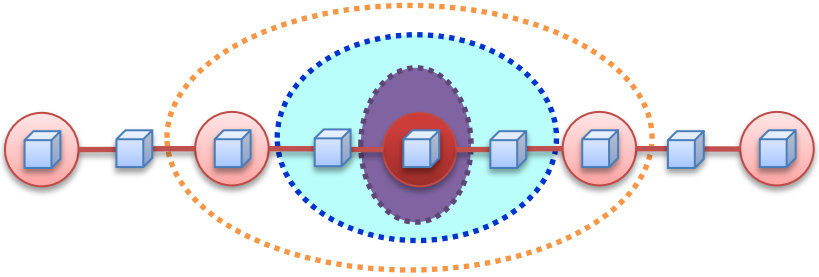
Update Functions
User Computation



Scheduler



Consistency Model

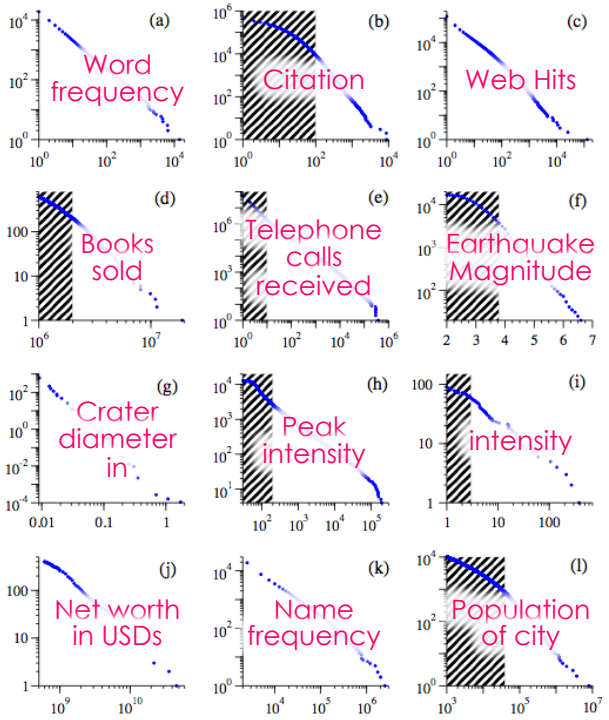


Natural Graphs are Skewed

Graphs in real-world

- *Power-law* degree distribution (aka Zipf's law or Pareto)

"most vertices have relatively few neighbors while a few have many neighbors" -- *PowerGraph*[OSDI'12]



Source: M. E. J. Newman, "Power laws, Pareto distributions and Zipf's law", 2005

Case: SINA *Weibo*(微博)

- 167 million active users[1]

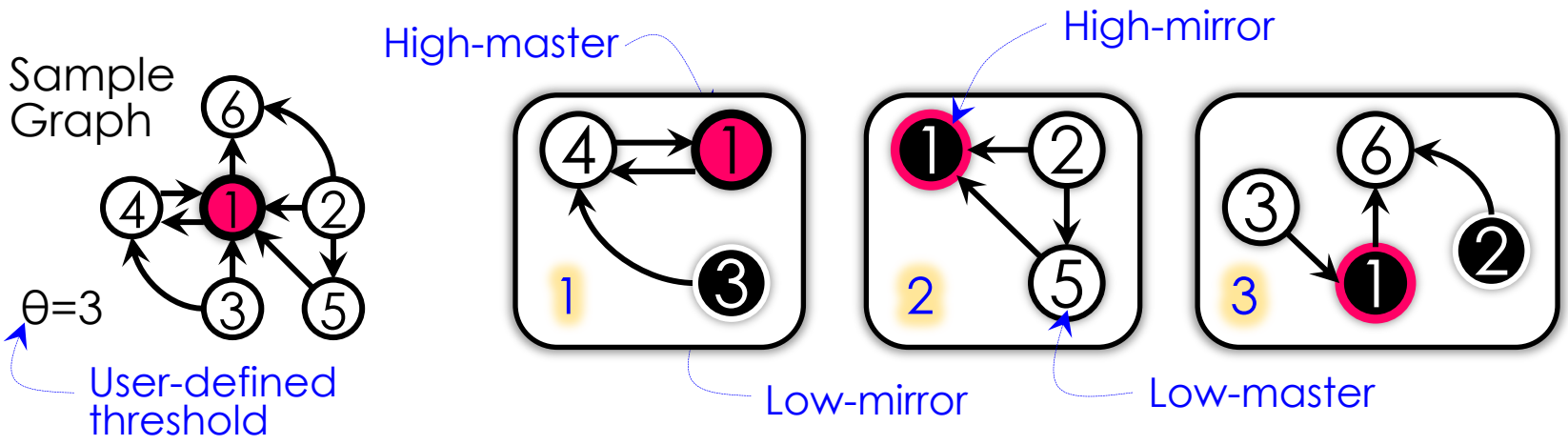
#Follower : 77,971,093 76,946,782 ··· 14,723,818 ····· 397 ····· 69 ·····

[1] http://www.chinainternetwatch.com/10735/weibo-q3-2014/

Graph Partitioning

Hybrid-cut: differentiate graph partitioning for low-degree and high-degree vertices

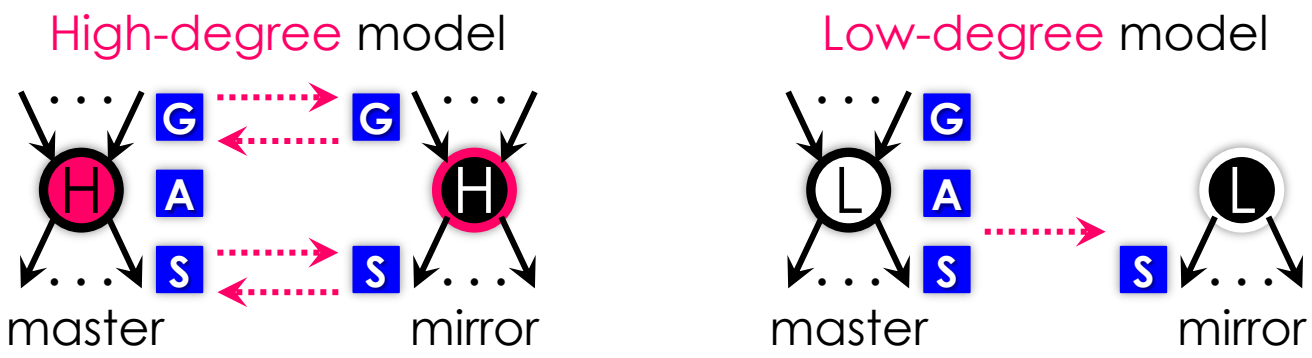
- Low-cut (inspired by *edge-cut*): reduce mirrors and exploit locality for low-degree vertices
- High-cut (inspired by *vertex-cut*): provide balance and restrict the impact of high-degree vertices
- Efficiently combine low-cut and high-cut



Graph Computation

Hybrid-model: differentiate the graph computation model to high-degree and low-degree vertices

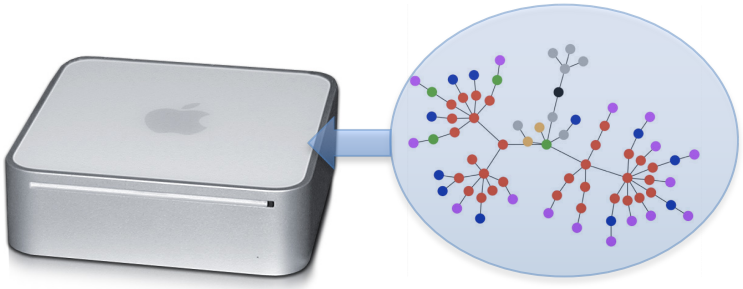
- High-degree model: decompose workload of high-degree vertices for load balance
- Low-degree model: minimize the communication cost for low-degree vertices
- Seamlessly run all of existing graph algorithms



Research Goal

Compute on graphs with billions of edges, in *a reasonable time,* on a single PC.

- *Reasonable* = close to numbers previously reported for distributed systems in the literature.

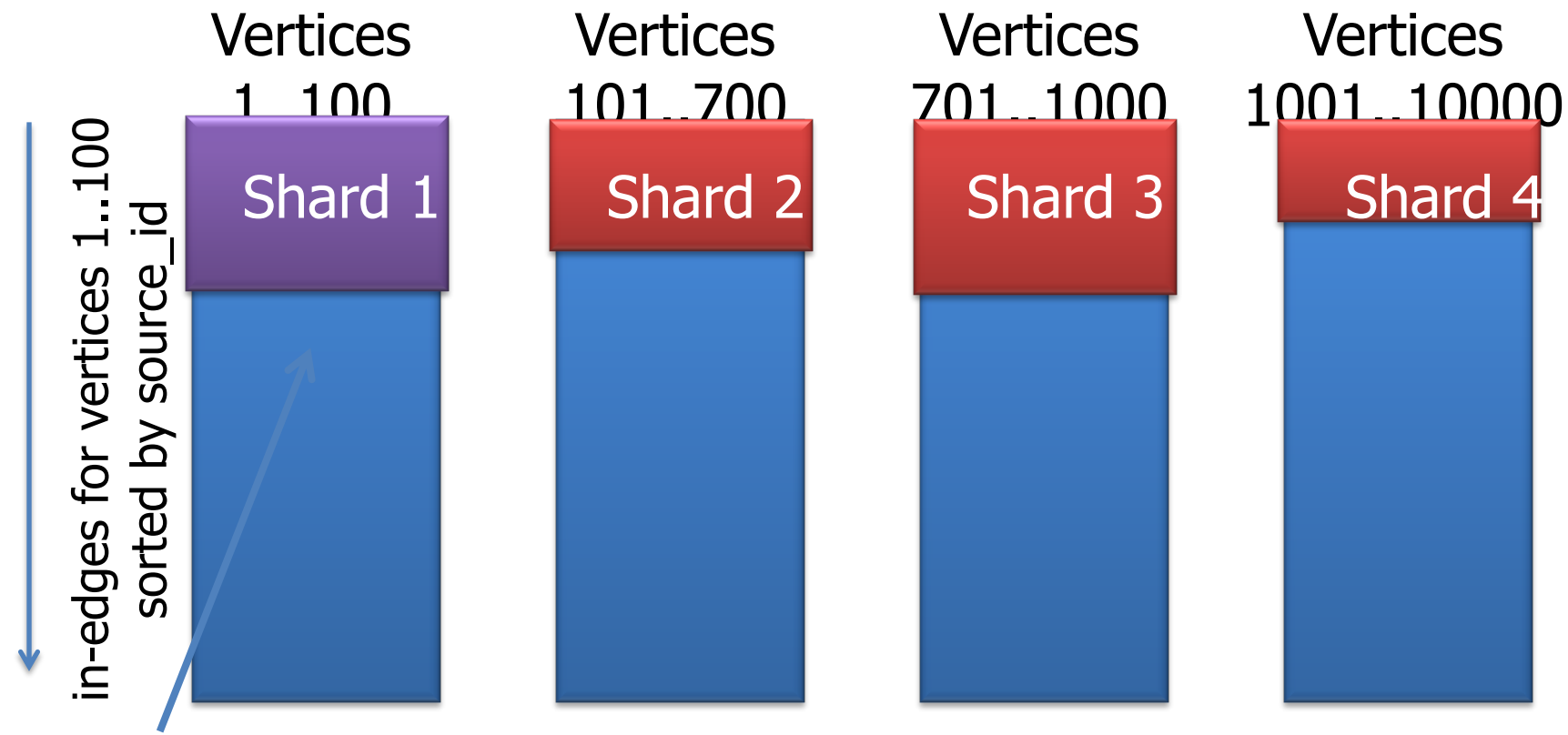


Experiment PC: Mac Mini (2012)

PSW: Loading Sub-graph

1. Load
2. Compute
3. Write

Load subgraph for vertices 1..100

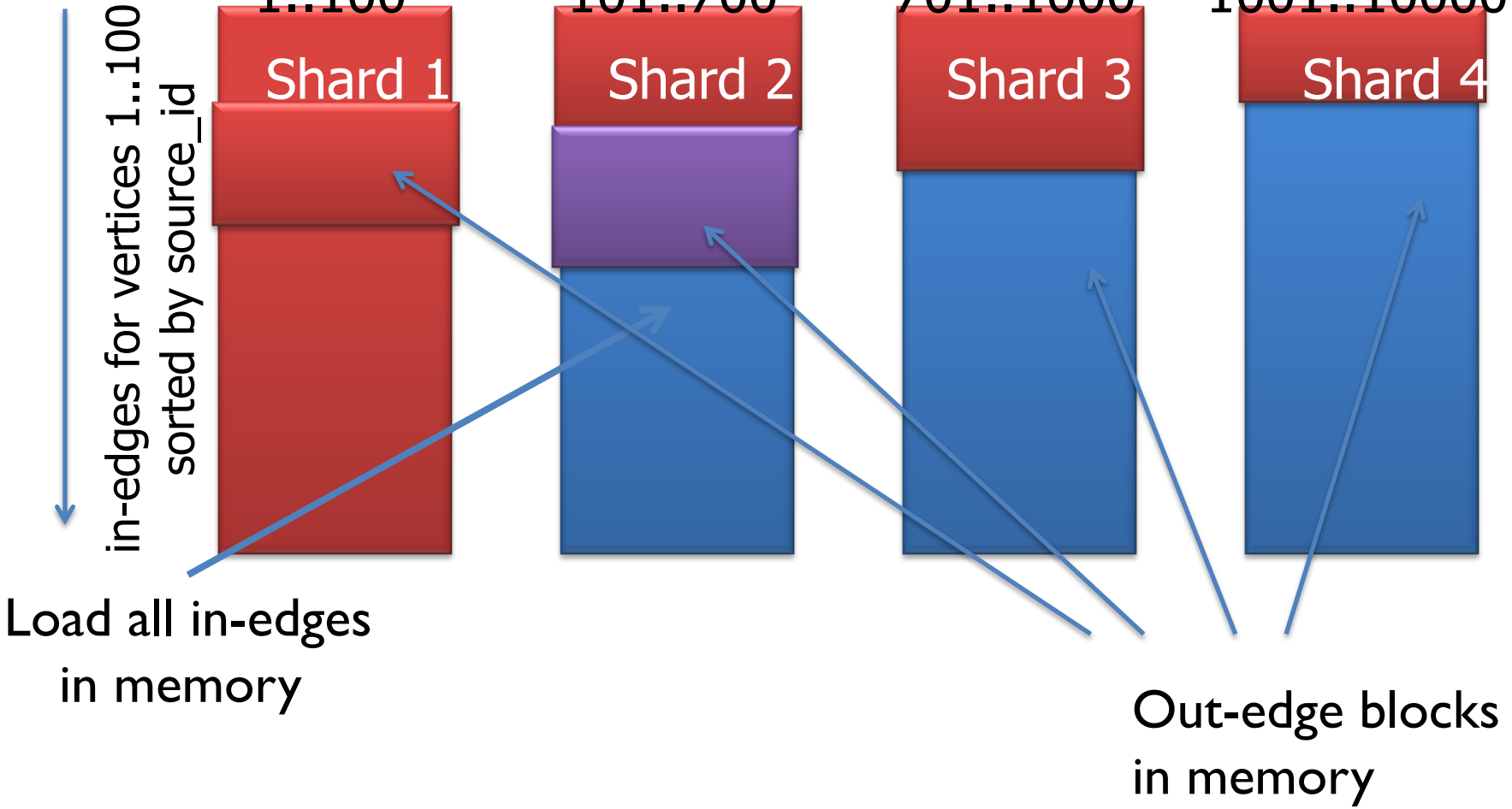


Load all in-edges in memory

What about out-edges?
Arranged in sequence in other shards

PSW: Loading Sub-graph

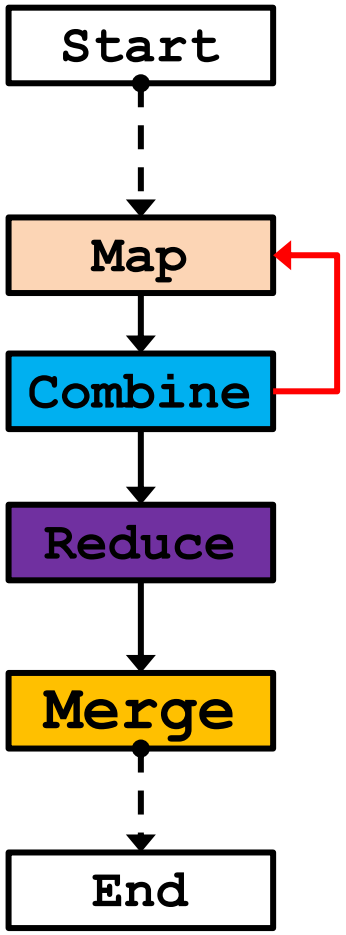
Load subgraph for vertices 101..700



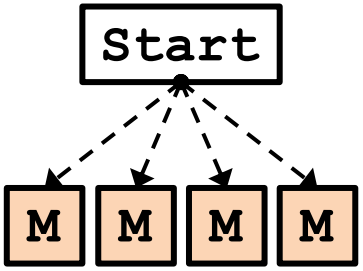
Tiled-MapReduce

Extensions to MapReduce Model

1. Replace the **Map** phase with a loop of **Map** and **Reduce** phases
2. Process one sub-job in each iteration
3. Rename the **Reduce** phase within loop to the **Combine** phase
4. Modify the **Reduce** phase to process the partial results of all iterations



Input Data Memory Reuse



Processors

Worker Threads

Disk

Input

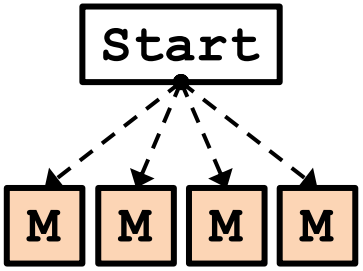
Main Memory

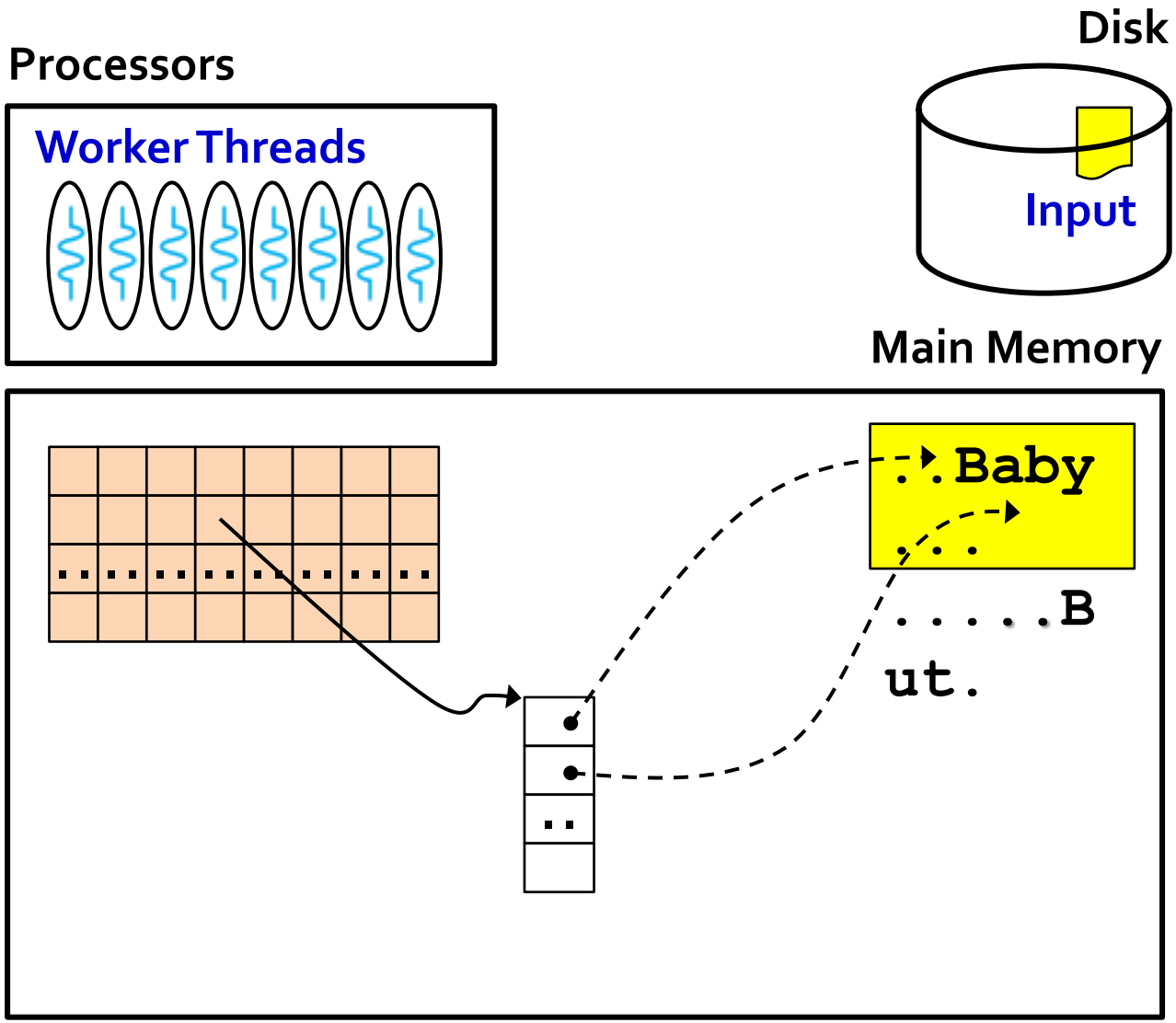
Input Buffer

Load

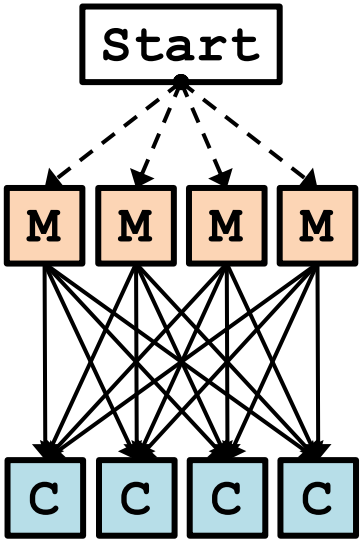
acquire

Input Data Reuse





Input Data Reuse



Processors

Worker Threads

Disk

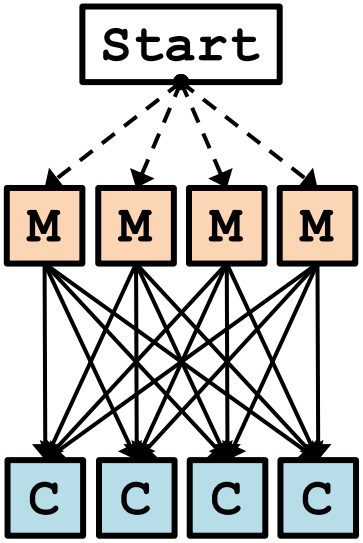
Input

Main Memory

Baby

New Buffer

Input Data Reuse



Processors

Worker Threads

Disk

Input

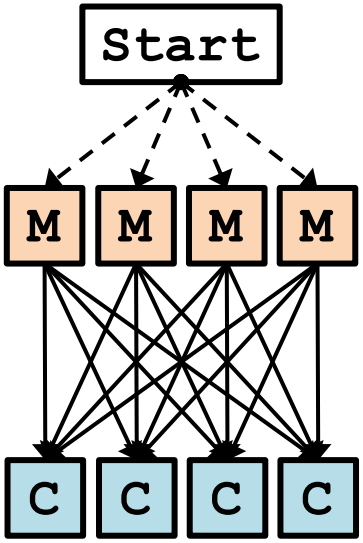
Main Memory

..Baby
...
.....B
ut.

New Buffer

Bab
y
But
...

Input Data Reuse



Processors

Worker Threads

Disk

Input

Main Memory

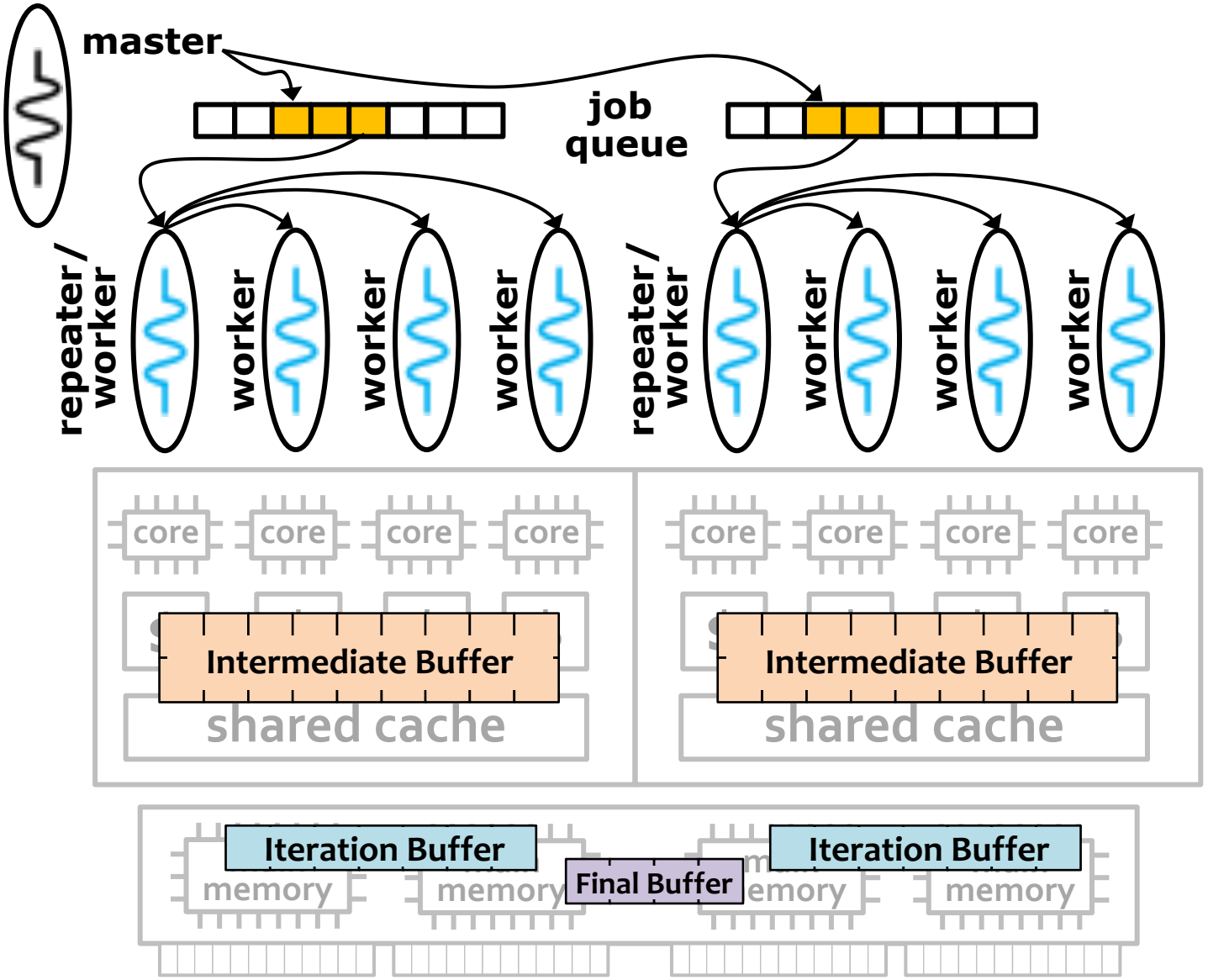
Free

release

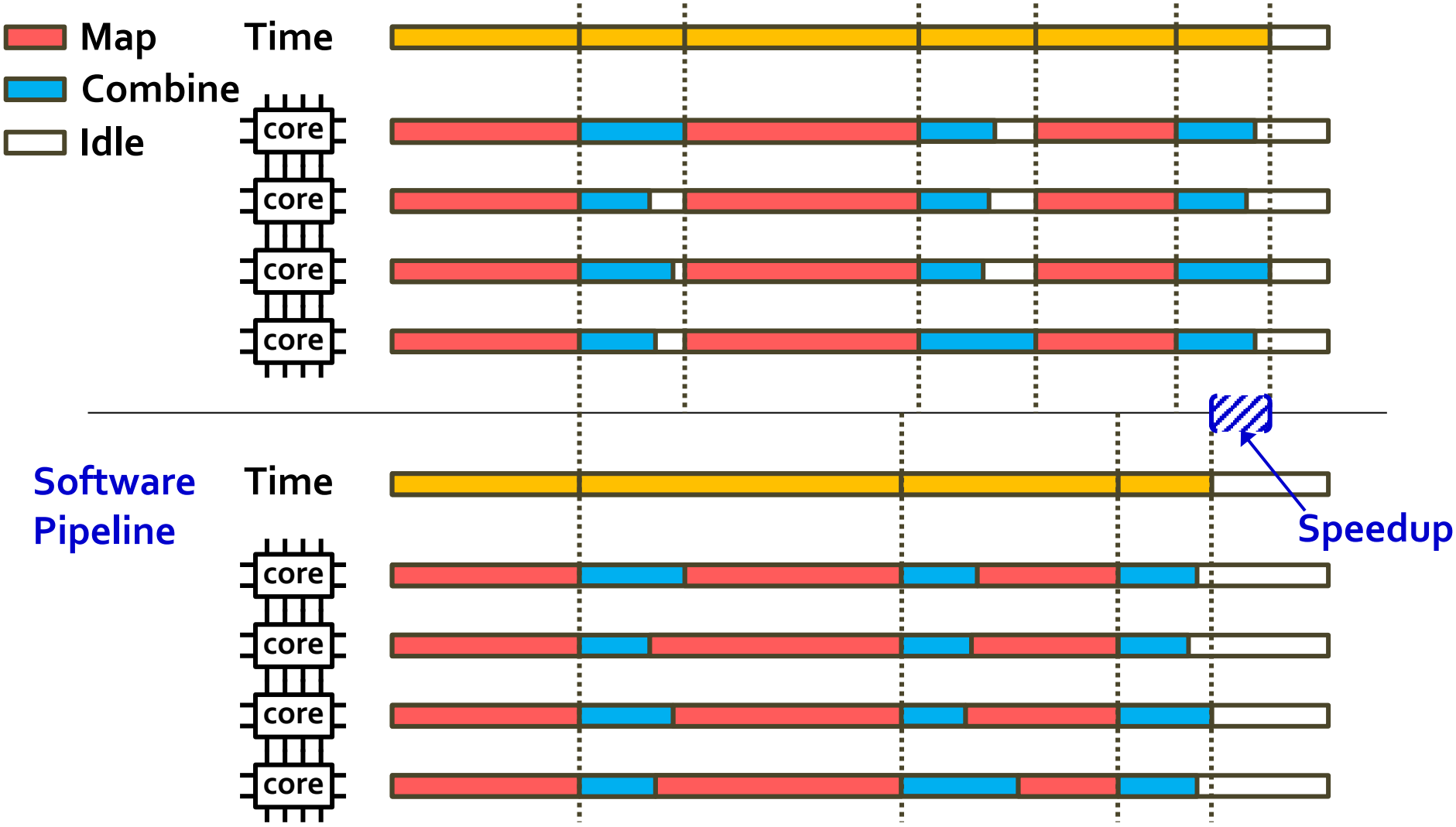
Bab
y
But

. . .

NUCA/NUMA-Aware Scheduler

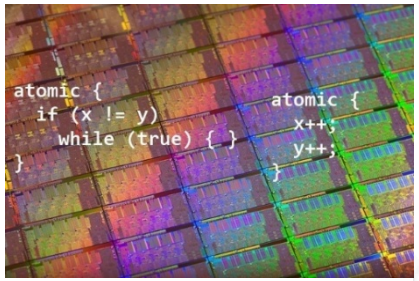


Software Pipeline

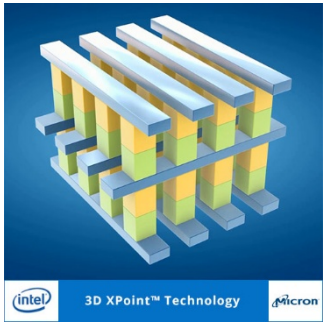


Hardware Platform

HTM

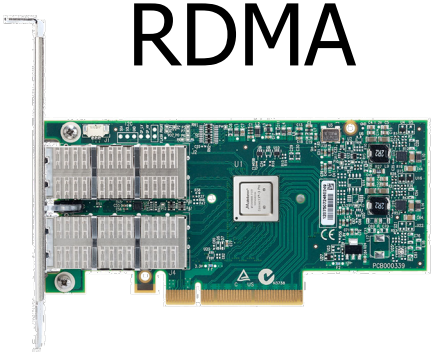


NVM



GPU

RDMA



A few rack-scale machines

Opportunities with HTM & RDMA

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

> *a non-transactional code will unconditionally abort a transaction when their accesses conflict*

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

> *one-sided RDMA operations are cache-coherent with local accesses*

HTM Strong Atomicity + **RDMA Strong Consistency** ⇒ **RDMA ops will abort conflicting HTM TX**

DrTM-KV's Design

- ✓ Simple hash structure to fully leverage HTM
- ✓ Decouple race detection from memory store
  - Use HTM to detect races
  - Use one-sided RDMA ops for remote read & write
- ✓ Location-based and fully transparent cache

	Pilaf	FaRM	DrTM-kV
Hashing	Cuckoo	Hopscotch	Chaining
Race Detection	Checksum	Versioning	HTM
Remote Read	One-sided RDMA		One-sided RMDA
Remote Write	Messaging		
Caching	No		Yes

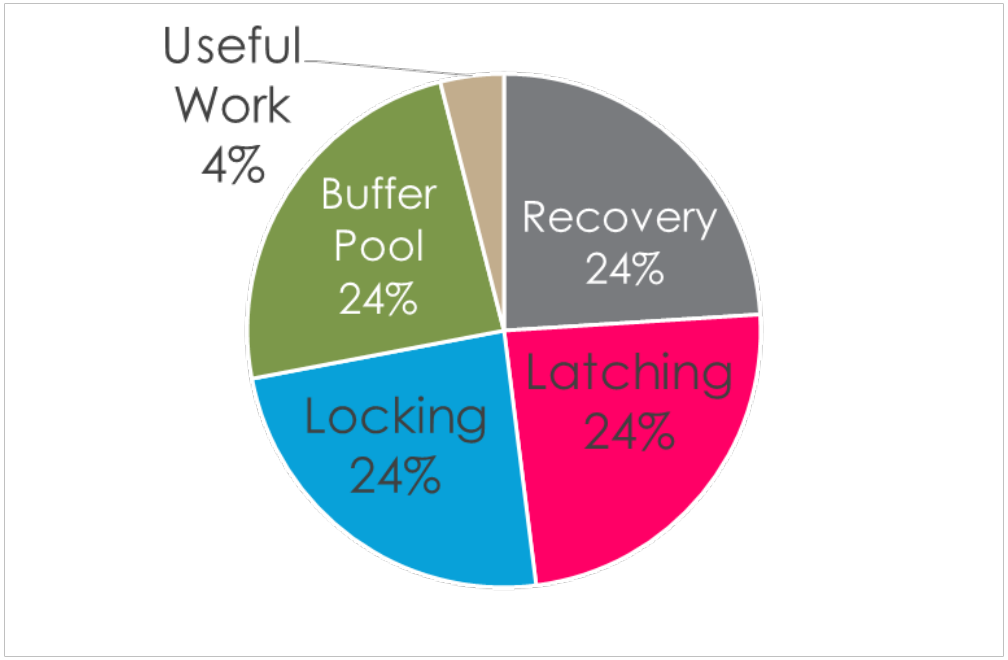
→ Simple

→ Efficient

Why (Distributed) TXs are Slow?

Only **4% of wall-clock time** spent on useful data processing, while the rest is occupied with **buffer pools**, **locking**, **latching**, **recovery**.[1]

-- Michael *Stonebraker*

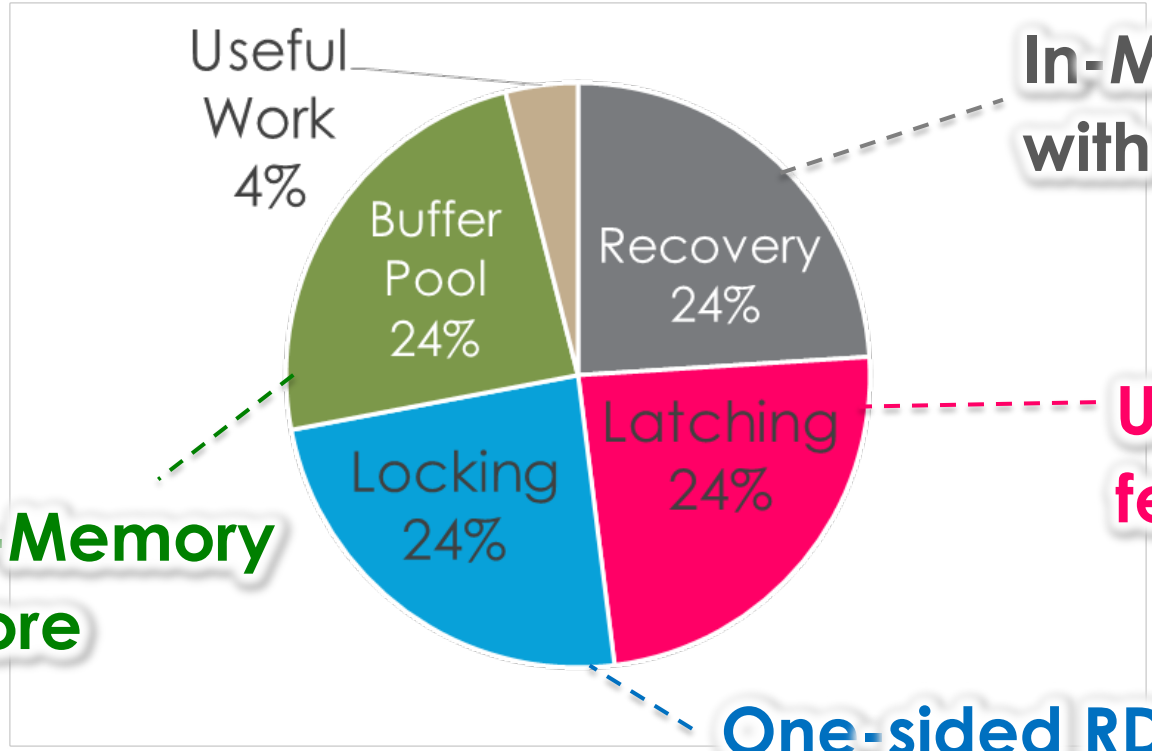


[1] "The Traditional RDBMS Wisdom is All Wrong"

DrTM: Fast In-memory Transaction Processing using RDMA and HTM

Use HTM's ACI properties for local TX execution

Use one-sided RDMA to glue multiple HTM TXs



In-Memory Logging with NVM

Use HTM's ACI features

In-Memory Store

One-sided RDMA Ops

Summary

In-memory computing demands high throughput and low latency

RDMA: helps bridge the gap from incommensurate scaling for in-memory computing

- Distributed key/value store
- In-memory transactions

Achieving orders-of-magnitude lower latency & higher throughput than prior state-of-the-art centralized and distributed systems

THANKS