# Google Cluster Computing Faculty Training Workshop

## Module I: Introduction to MapReduce

# Workshop Syllabus

- Seven lecture modules
  - Information about teaching the course
  - Technical info about Google tools & Hadoop
  - Example course lectures
- Four lab exercises
  - Assigned to students in UW course

# Overview

- University of Washington Curriculum
  - Teaching Methods
  - Reflections
  - Student Background
  - Course Staff Requirements
- Introductory Lecture Material

# UW: Course Summary

- Course title: “Problem Solving on Large Scale Clusters”
- Primary purpose: developing large-scale problem solving skills
- Format: 6 weeks of lectures + labs, 4 week project

# UW: Course Goals

- Think creatively about large-scale problems in a parallel fashion; design parallel solutions
- Manage large data sets under memory, bandwidth limitations
- Develop a foundation in parallel algorithms for large-scale data
- Identify and understand engineering trade-offs in real systems

# Lectures

- 2 hours, once per week
- Half formal lecture, half discussion
- Mostly covered systems & background
- Included group activities for reinforcement

# Classroom Activities

- Worksheets included pseudo-code programming, working through examples
  - Performed in groups of 2—3
- Small-group discussions about engineering and systems design
  - Groups of ~10
  - Course staff facilitated, but mostly open-ended

# Readings

- No textbook
- One academic paper per week
  - E.g., “Simplified Data Processing on Large Clusters”
  - Short homework covered comprehension
- Formed basis for discussion

# Lecture Schedule

- Introduction to Distributed Computing
- MapReduce: Theory and Implementation
- Networks and Distributed Reliability
- Real-World Distributed Systems
- Distributed File Systems
- Other Distributed Systems

# Intro to Distributed Computing

- What is distributed computing?
- Flynn’s Taxonomy
- Brief history of distributed computing
- Some background on synchronization and memory sharing

# MapReduce

- Brief refresher on functional programming
- MapReduce slides
  - More detailed version of module I
- Discussion on MapReduce

# Networking and Reliability

- Crash course in networking
- Distributed systems reliability
  - What is reliability?
  - How do distributed systems fail?
  - ACID, other metrics
- Discussion: Does MapReduce provide reliability?

# Real Systems

- Design and implementation of Nutch
- Tech talk from Googler on Google Maps

# Distributed File Systems

- Introduced GFS
- Discussed implementation of NFS and AndrewFS (AFS) for comparison

# Other Distributed Systems

- BOINC: Another platform
- Broader definition of distributed systems
  - DNS
  - One Laptop per Child project

# Labs

- Also 2 hours, once per week
- Focused on applications of distributed systems
- Four lab projects over six weeks

# Lab Schedule

- Introduction to Hadoop, Eclipse Setup, Word Count
- Inverted Index
- PageRank on Wikipedia
- Clustering on Netflix Prize Data

# Design Projects

- Final four weeks of quarter
- Teams of 1—3 students
- Students proposed topic, gathered data, developed software, and presented solution

# Example: Geozette


Image © Julia Schwartz

# Example: Galaxy Simulation

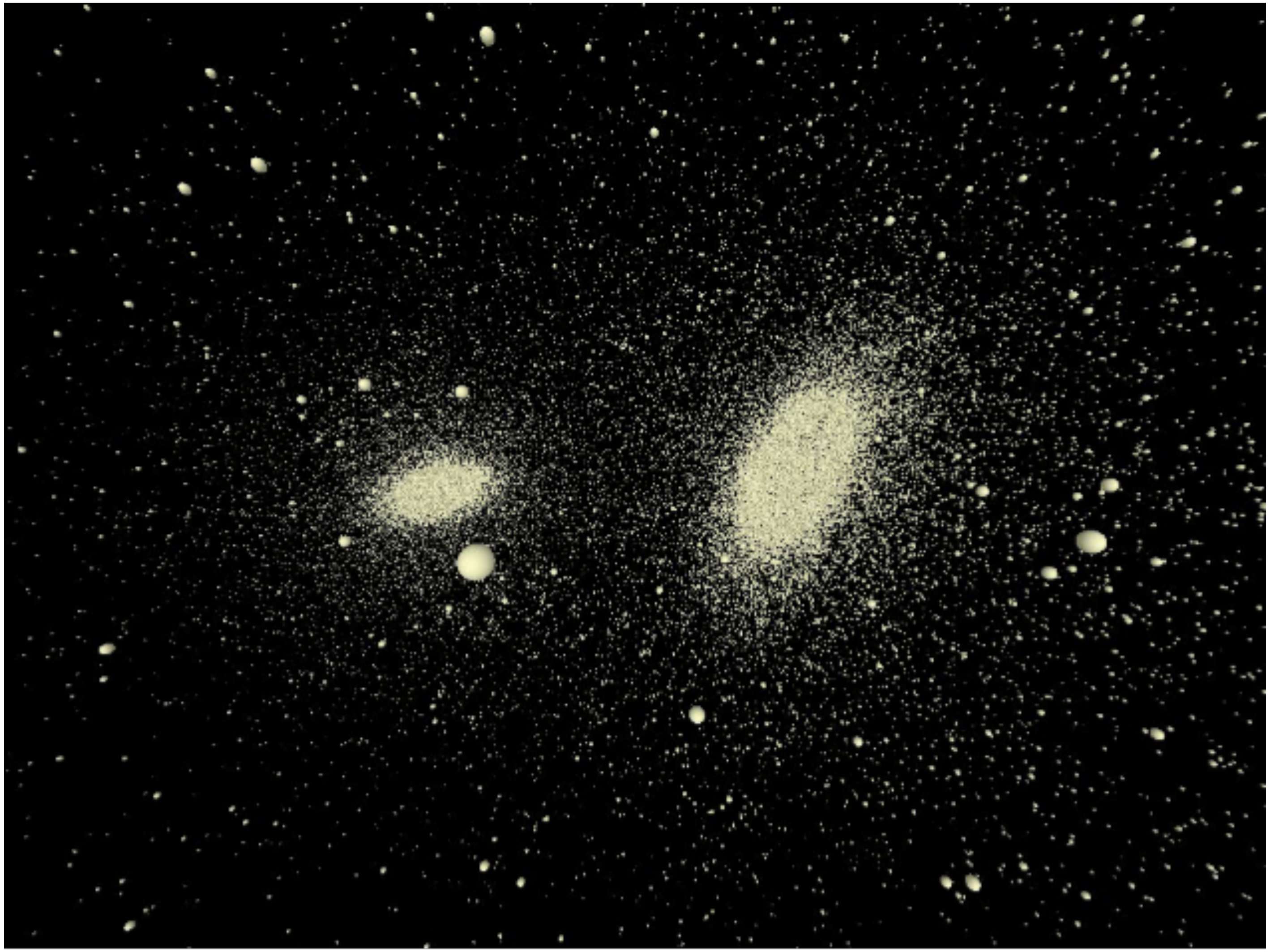


Image © Slava Chernyak, Mike Hoak



# Other Projects

- Bayesian Wikipedia spam filter
- Unsupervised synonym extraction
- Video collage rendering

# Common Features

- Hadoop!
- Used publicly-available web APIs for data
- Many involved reading papers for algorithms and translating into MapReduce framework

# Course Staff

- Instructor (me!)
- Two undergrad teaching assistants
  - Helped facilitate discussions, directed labs
- One student sys admin
  - Worked only about three hours/week

# Preparation

- Teaching assistants had taken previous iteration of course in winter
- Lectures retooled based on feedback from that quarter
  - Added reasonably large amount of background material
- Ran & solved all labs in advance

# The Course: What Worked

- Discussions
  - Often covered broad range of subjects
- Hands-on lab projects
- “Active learning” in classroom
- Independent design projects

# Things to Improve: Coverage

- Algorithms were not reinforced during lecture
  - Students requested much more time be spent on “how to parallelize an iterative algorithm”
- Background material was very fast-paced

# Things to Improve: Projects

- Labs could have used a moderated/scripted discussion component
  - Just “jumping in” to the code proved difficult
  - No time was devoted to Hadoop itself in lecture
  - Clustering lab should be split in two
- Design projects could have used more time

# Conclusions

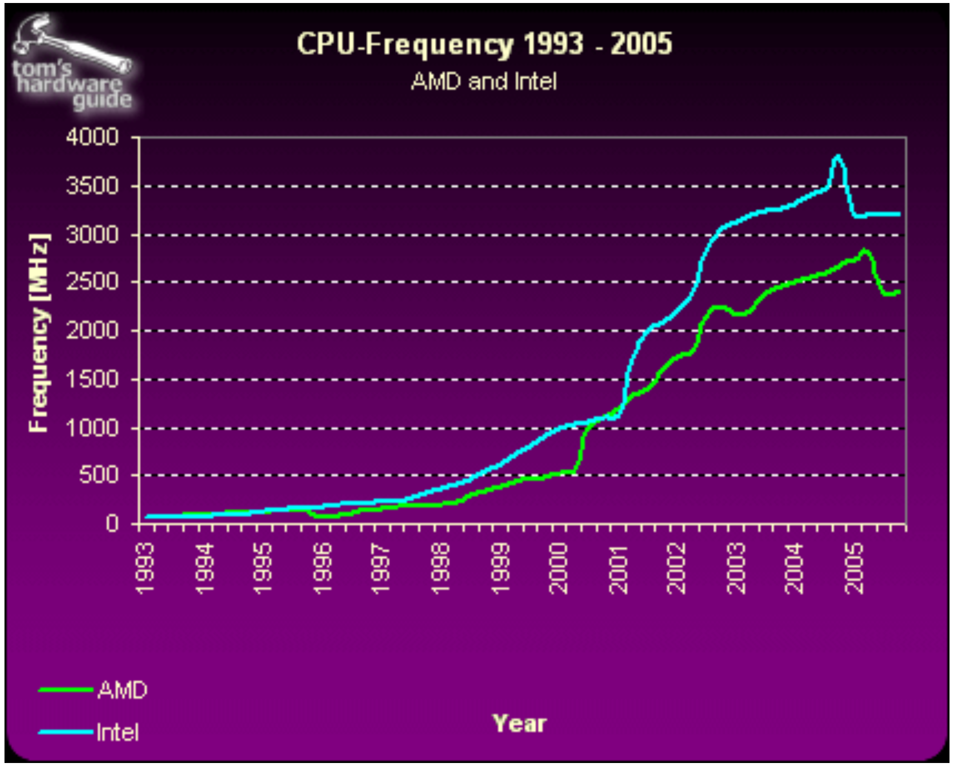
- Solid basis for future coursework
  - Needs additional background (e.g., algorithms)
  - Full semester requires additional material (e.g., distributed systems, web systems course)
- Hadoop-based systems exciting to students & can teach important CS

# Introductory Distributed Systems Material

# Overview

- Introduction
- Models of computation
- A brief history lesson
- Connecting distributed modules
- Failure & reliability

# Computer Speedup




Moore's Law: "*The density of transistors on a chip doubles every 18 months, for the same cost*" (1965)



Image: Tom's Hardware

# Scope of Problems

- What can you do with 1 computer?
- What can you do with 100 computers?
- What can you do with an entire data center?

# Distributed Problems

- Rendering multiple frames of high-quality animation

Image: DreamWorks Animation

# Distributed Problems

- Simulating several hundred or thousand characters

*Happy Feet* © Kingdom Feature Productions; *Lord of the Rings* © New Line Cinema

# Distributed Problems

- Indexing the web (Google)
- Simulating an Internet-sized network for networking experiments (PlanetLab)
- Speeding up content delivery (Akamai)

*What is the key attribute that all these examples have in common?*

# Distributed Problems

- All involve **separable** computation
- Many involve data that necessarily must be stored in **multiple locations**.

- For a problem to be distributable, different components of the problem should be able to be handled **independently**.

# Taking A Step Back

- Before we talk more about distributed computing… what does it mean to design "a computer?"

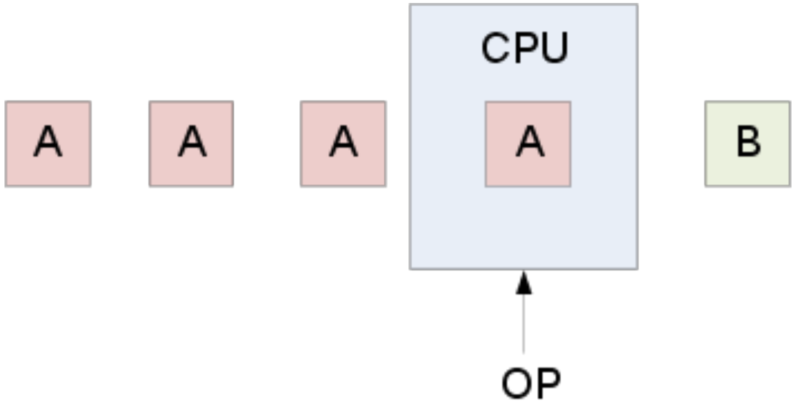
- How would a distributed or parallel system look different from a single-CPU machine?

# Flynn's Taxonomy

- Four categories of computer architectures
- Broke down serial/parallel in terms of instructions and data

| | |
|---|---|
| SISD | SIMD |
| MISD | MIMD |

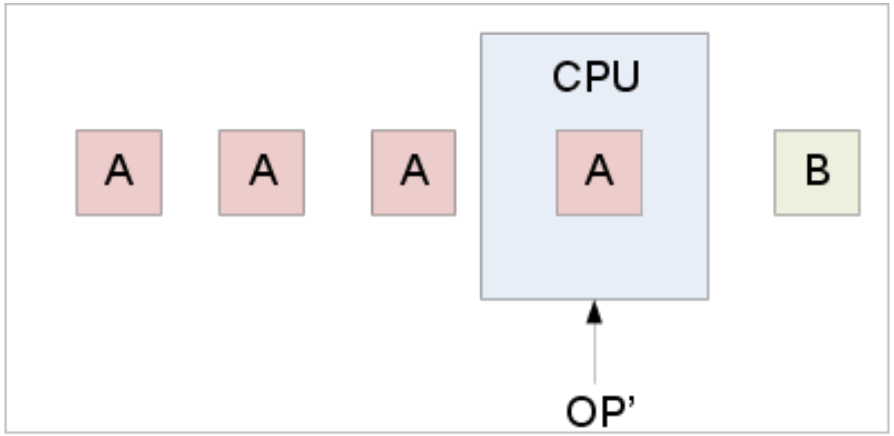
# SISD

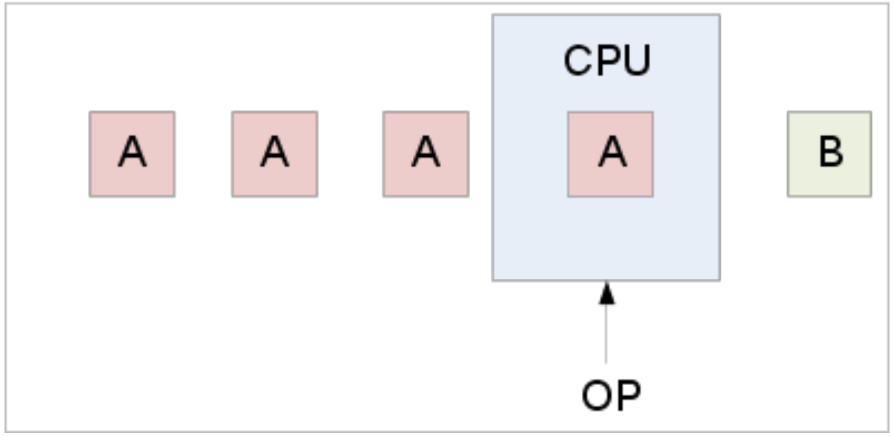



Single instruction, single data element

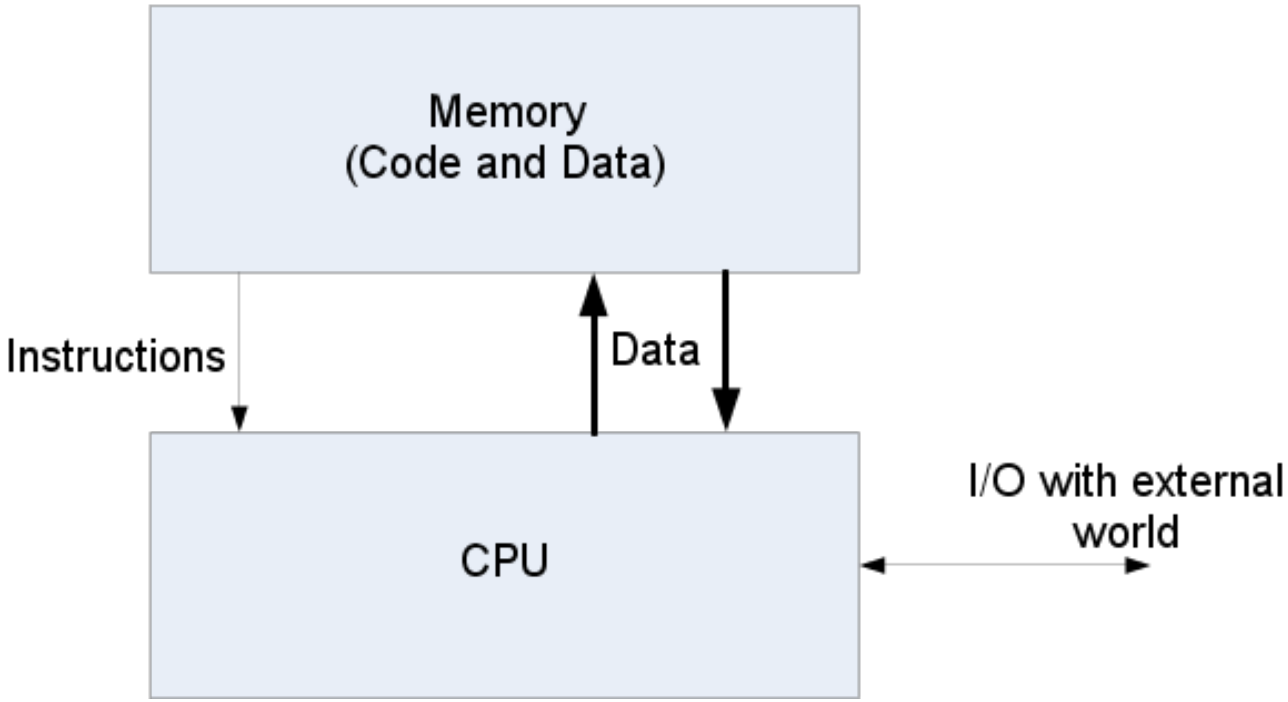
# MIMD

Multiple instructions, multiple data elements



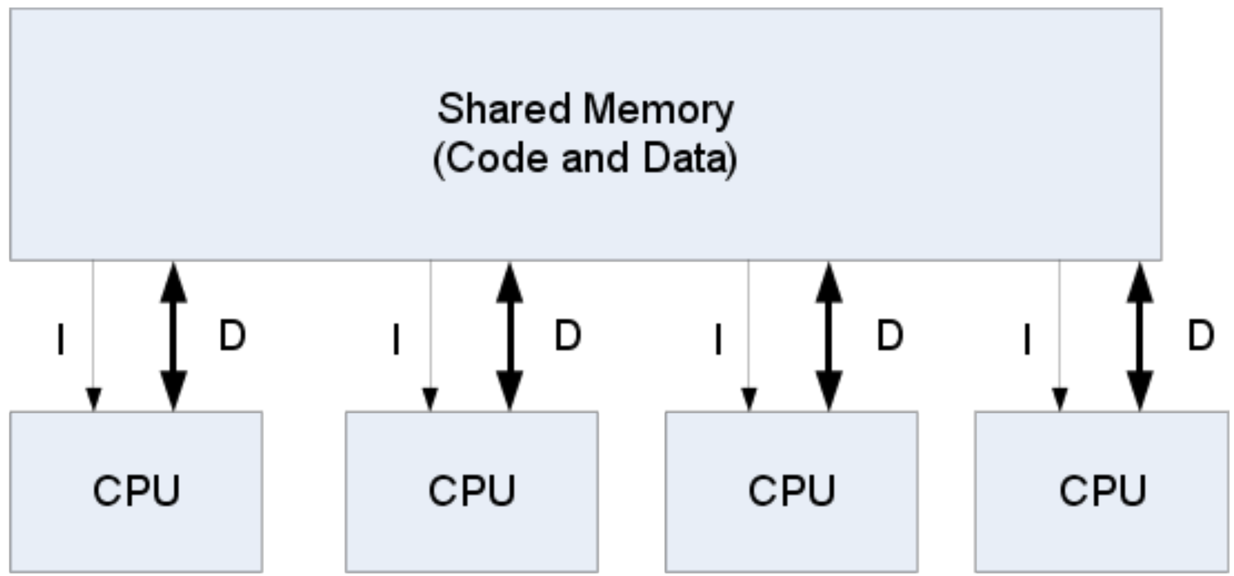

# Models of Computing




The Von Neumann architecture a.k.a. RAM model

… How do we extend this to parallel computing?

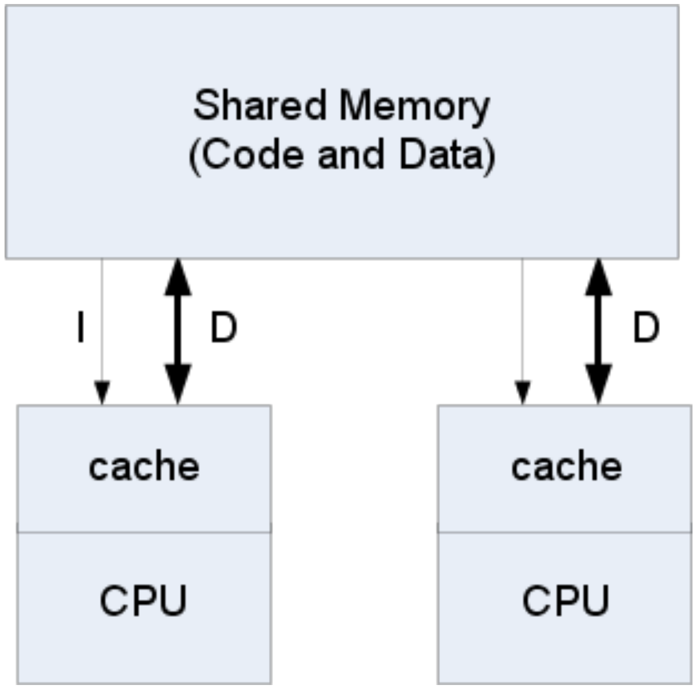
# A First Try: PRAM




Parallel Random Access Machine model:

- N processors connected to shared memory
- All memory addresses reachable in unit time by any CPU
- All processors execute one instruction per tick in “lock step”

# … Does not even cover Core2Duo




Although there were some early attempts…

# Early Parallel Computing

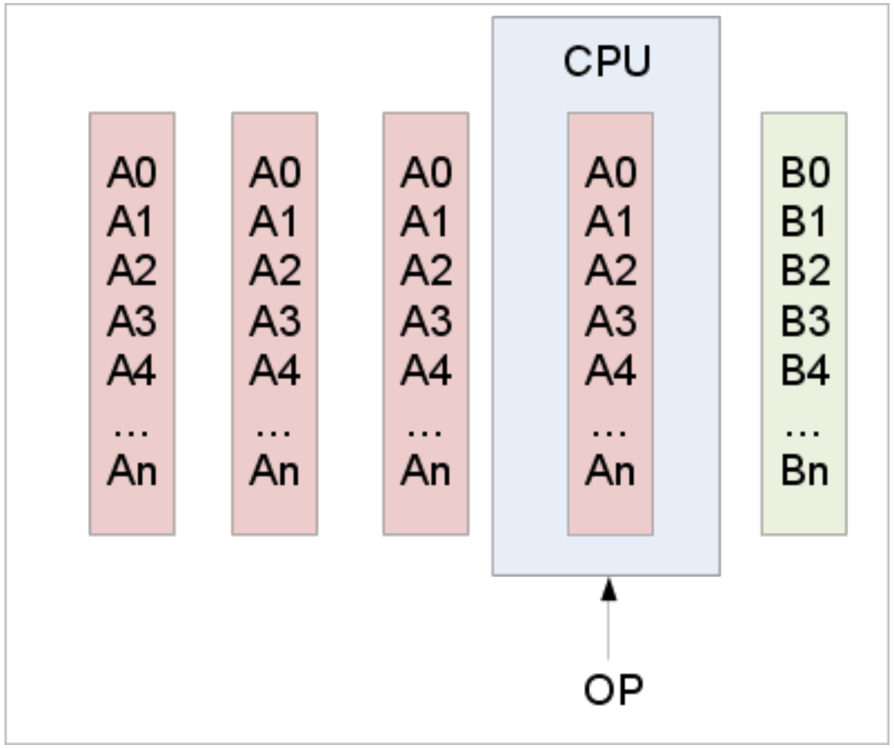
- CDC 6600: Out-of-order execution (1964)
- CDC 7600: Pipelining
- CDC 8600: Multi-core! 4 7600's in one box
  - Provided lock-step execution of CPUs
  - NB: Memory speed at the time exceeded CPU speed
  - … Also never actually made it to production

# Vector Processing

- Cray 1 (1976) allowed programmers to apply operations to large chunks of data at a time

SIMD architecture:

Single instruction, multiple data




# Loop Compilation

```
for (i = 0; i < N; i++) {
a[i] = b[i] + c[i];
}
```

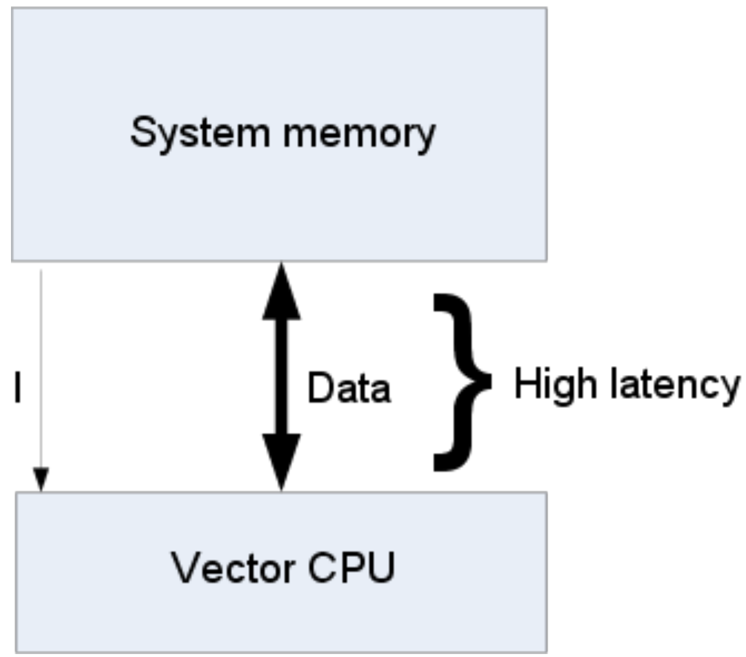
```
top:
compare i, N
jge exit
load_offset $1, b, i
load_offset $2, c, i
add $3, $1, $2
store_offset $3, a, i
inc i
j top
exit:
```

# Vector Compilation

```
for (i = 0; i < N; i++) {
a[i] = b[i] + c[i];
}
```

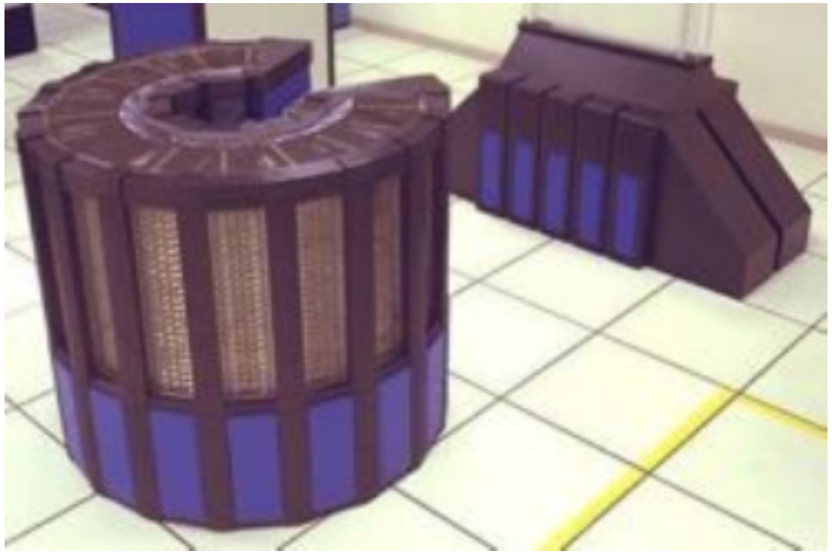
```
load_vector $1, b, N
load_vector $2, c, N
add $3, $1, $2
store_vector $3, a, N
```

# Vector Memory Operations



# 1975-85

- Parallel computing was favored in the early years
- Primarily vector-based at first
- Gradually more thread-based parallelism was introduced

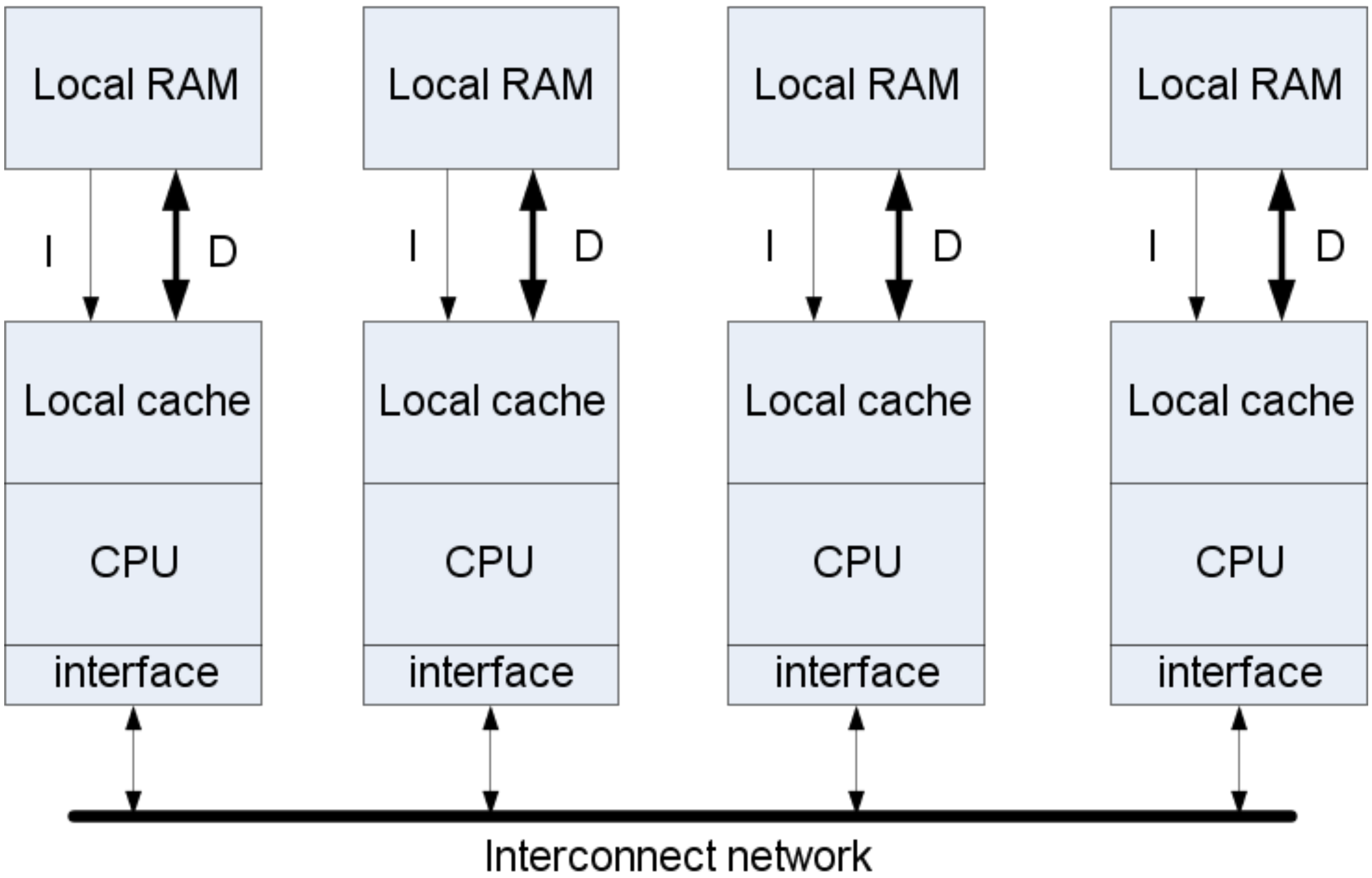


Cray 2 supercomputer (Wikipedia)

# System Organization

- Having one big memory would make it a huge bottleneck
  - Eliminates all of the parallelism
- The PRAM model does not work
  - Lock-step execution too restrictive
  - Does not accurately model memory

# CTA: Memory is Distributed



# Interconnect Networks

- Bottleneck in the CTA is transferring values from one local memory to another
- Interconnect network design very important; several options are available
- Design constraint: How to minimize interconnect network usage?

# A Brief History… 1985-95

- “Massively parallel architectures” start rising in prominence
- Message Passing Interface (MPI) and other libraries developed
- Bandwidth was a big problem
  - For external interconnect networks in particular

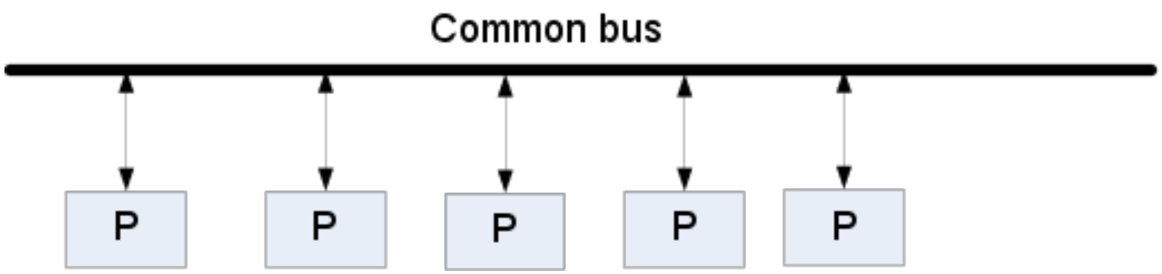
# A Brief History… 1995-Today

- Cluster/grid architecture increasingly dominant
- Special node machines eschewed in favor of COTS technologies
- Web-wide cluster software
- Companies like Google take this to the extreme (10,000 node clusters)

# More About Interconnects

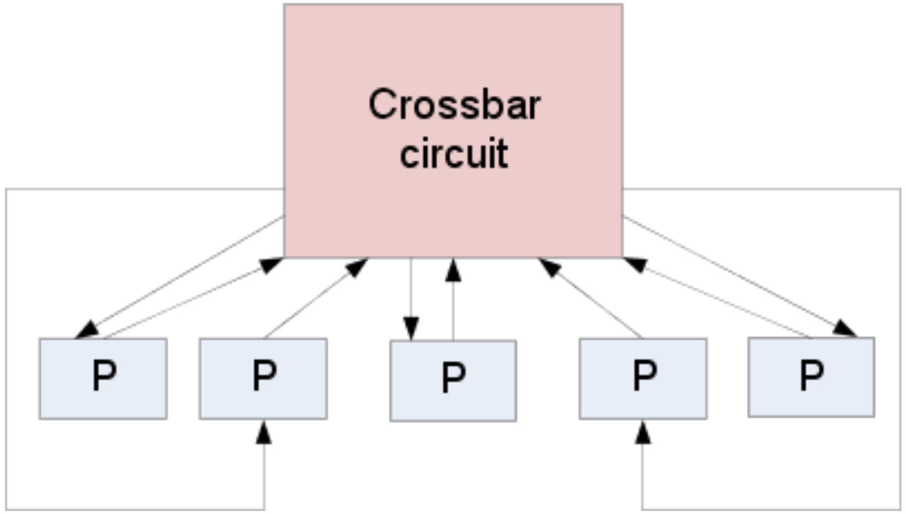
- Several types of interconnect possible
  - Bus
  - Crossbar
  - Torus
  - Tree

# Interconnect Bus



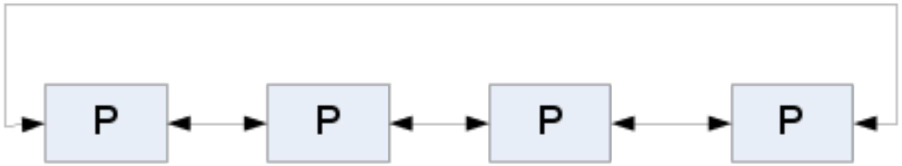

- Simplest possible layout
- Not realistically practical
    - Too much contention
    - Little better than “one big memory”

# Crossbar

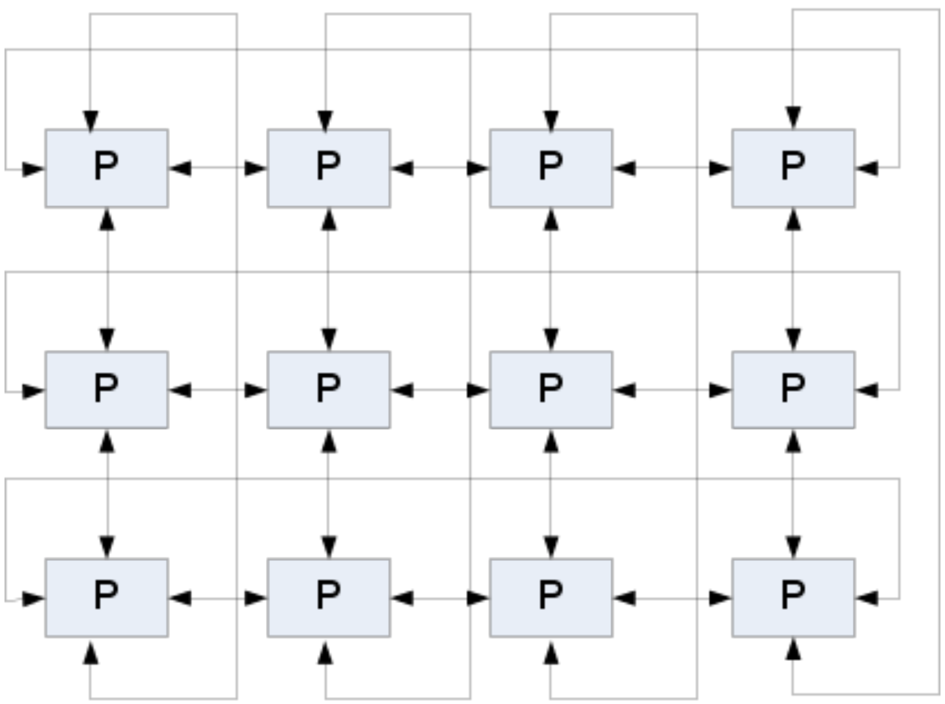



- All processors have “input” and “output” lines
- Crossbar connects any input to any output
- Allows for very low contention, but lots of wires, complexity
    - Will not scale to many nodes

# Toroidal networks



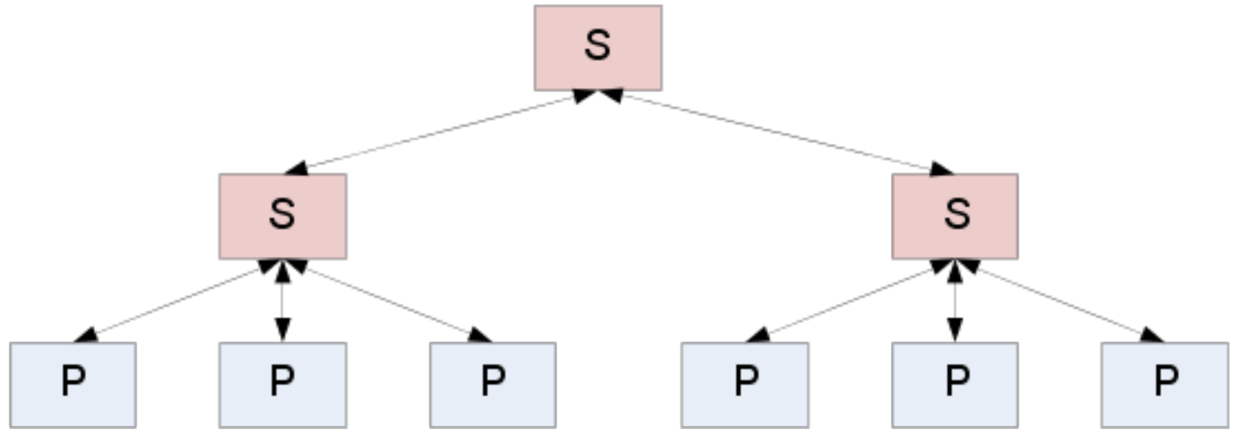

A 1-dimensional torus




A 2-dimensional torus

- Nodes are connected to their logical neighbors
- Node-node transfer may include intermediaries
- Reasonable trade-off for space/scalability

# Tree




- Switch nodes transfer data “up” or “down” the tree
- Hierarchical design keeps “short” transfers fast, incremental cost to longer transfers
- Aggregate bandwidth demands often very large at top
- Most natural layout for most cluster networks today

# Parallel vs. Distributed

- Parallel computing can mean:
  - Vector processing of data (SIMD)
  - Multiple CPUs in a single computer (MIMD)
- Distributed computing is multiple CPUs across many computers (MIMD)

# What is Different in Distributed?

- Higher inter-CPU communication latency
  - Individual nodes need to act more autonomously
- Different nodes can be heterogeneous (by function, location…)
- System reliability is much harder to maintain

“A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable”

-- Leslie Lamport

# Reliability Demands

- Support partial failure
  - Total system must support graceful decline in application performance rather than a full halt

# Reliability Demands

- Data Recoverability
  - If components fail, their workload must be picked up by still-functioning units

# Reliability Demands

- Individual Recoverability
  - Nodes that fail and restart must be able to rejoin the group activity without a full group restart

# Reliability Demands

- Consistency
  - Concurrent operations or partial internal failures should not cause externally visible nondeterminism

# Reliability Demands

- Scalability
  - Adding increased load to a system should not cause outright failure, but a graceful decline
  - Increasing resources should support a proportional increase in load capacity

# Reliability Demands

- Security
  - The entire system should be impervious to unauthorized access
  - Requires considering many more attack vectors than single-machine systems

Ken Arnold, CORBA designer:

> "Failure is the defining difference between distributed and local programming"

# Component Failure

- Individual nodes simply stop

# Data Failure

- Packets omitted by overtaxed router
- Or dropped by full receive-buffer in kernel
- Corrupt data retrieved from disk or net

# Network Failure

- External & internal links can die
  - Some can be routed around in ring or mesh topology
  - Star topology may cause individual nodes to appear to halt
  - Tree topology may cause “split”
  - Messages may be sent multiple times or not at all or in corrupted form…

# Timing Failure

- Temporal properties may be violated
  - Lack of “heartbeat” message may be interpreted as component halt
  - Clock skew between nodes may confuse version-aware data readers

# Byzantine Failure

- Difficult-to-reason-about circumstances arise
  - Commands sent to foreign node are not confirmed: What can we reason about the state of the system?

# Malicious Failure

- Malicious (or maybe naïve) operator injects invalid or harmful commands into system

# Correlated Failures

- Multiple CPUs/hard drives from same manufacturer lot may fail together
- Power outage at one data center may cause demand overload at failover center

# Preparing for Failure

- Distributed systems must be robust to these failure conditions
- But there are lots of pitfalls…

# The Eight Design Fallacies

- The network is reliable.
- Latency is zero.
- Bandwidth is infinite.
- The network is secure.
- Topology doesn't change.
- There is one administrator.
- Transport cost is zero.
- The network is homogeneous.

-- Peter Deutsch and James Gosling, Sun Microsystems

# Dealing With Component Failure

- Use heartbeats to monitor component availability
- “Buddy” or “Parent” node is aware of desired computation and can restart it elsewhere if needed
- Individual storage nodes should not be the sole owner of data
  - Pitfall: How do you keep replicas consistent?

# Dealing With Data Failure

- Data should be check-summed and verified at several points
  - Never trust another machine to do your data validation!
- Sequence identifiers can be used to ensure commands, packets are not lost

# Dealing With Network Failure

- Have well-defined split policy
  - Networks should routinely self-discover topology
  - Well-defined arbitration/leader election protocols determine authoritative components
    - Inactive components should gracefully clean up and wait for network rejoin

# Dealing With Other Failures

- Individual application-specific problems can be difficult to envision
- Make as few assumptions about foreign machines as possible
- Design for security at each step

# TPS: Definition

- A system that handles *transactions* coming from several sources concurrently
- Transactions are “events that generate and modify data stored in an information system for later retrieval”*

* http://en.wikipedia.org/wiki/Transaction_Processing_System

# Key Features of TPS: ACID

- “ACID” is the acronym for the features a TPS must support:

- **Atomicity** – A set of changes must all succeed or all fail
- **Consistency** – Changes to data must leave the data in a valid state when the full change set is applied
- **Isolation** – The effects of a transaction must not be visible until the entire transaction is complete
- **Durability** – After a transaction has been committed successfully, the state change must be permanent.

# Atomicity & Durability

What happens if we write half of a transaction to disk and the power goes out?

# Logging: The Undo Buffer

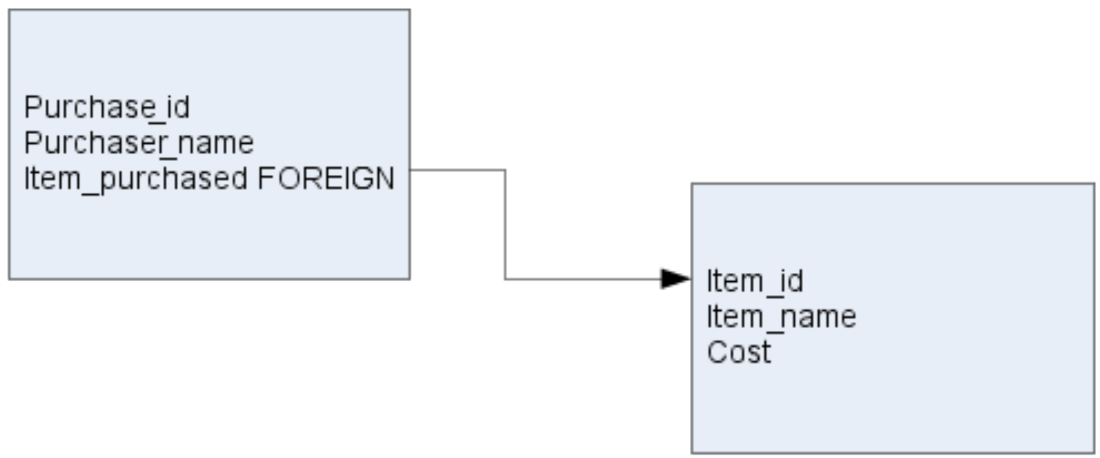
1. Database writes to log the current values of all cells it is going to overwrite
2. Database overwrites cells with new values
3. Database marks log entry as committed

- If db crashes during (2), we use the log to roll back the tables to prior state

# Consistency: Data Types

- Data entered in databases have rigorous data types associated with them, and explicit ranges
- Does not protect against all errors (entering a date in the past is still a valid date, etc), but eliminates tedious programmer concerns

# Consistency: Foreign Keys

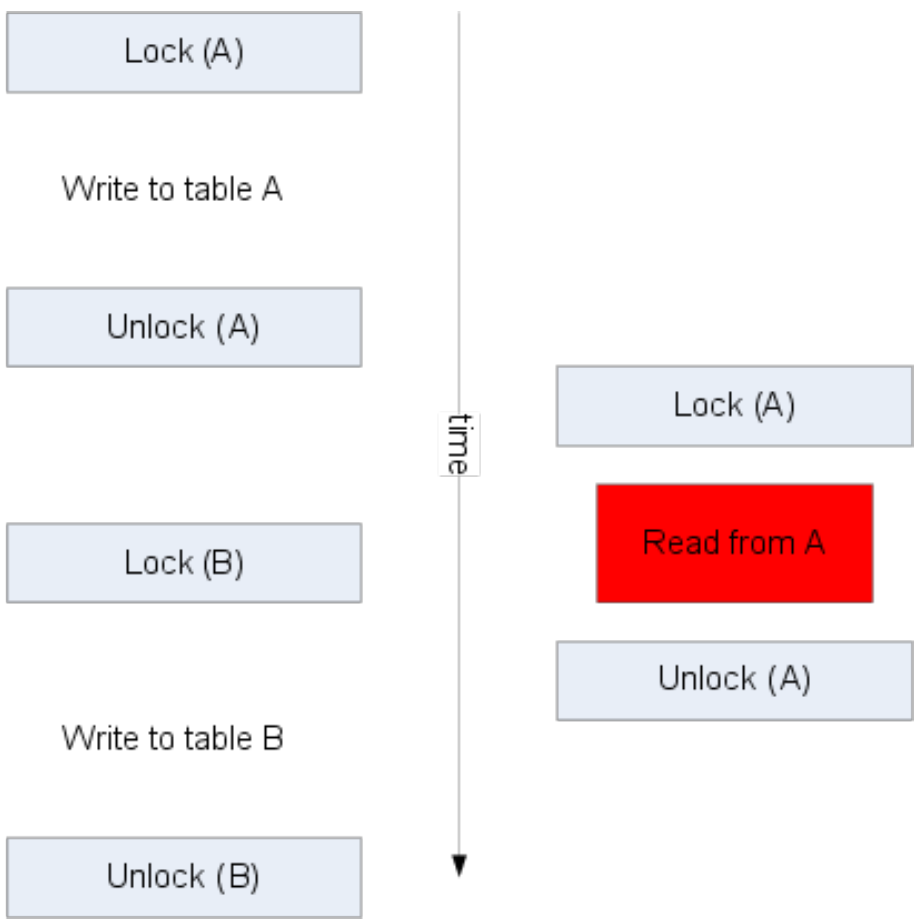



- Database designers declare that fields are indices into the keys of another table
- Database ensures that target key exists before allowing value in source field

# Isolation

- Using *mutual-exclusion locks*, we can prevent other processes from reading data we are in the process of writing
- When a database is prepared to commit a set of changes, it locks any records it is going to update before making the changes

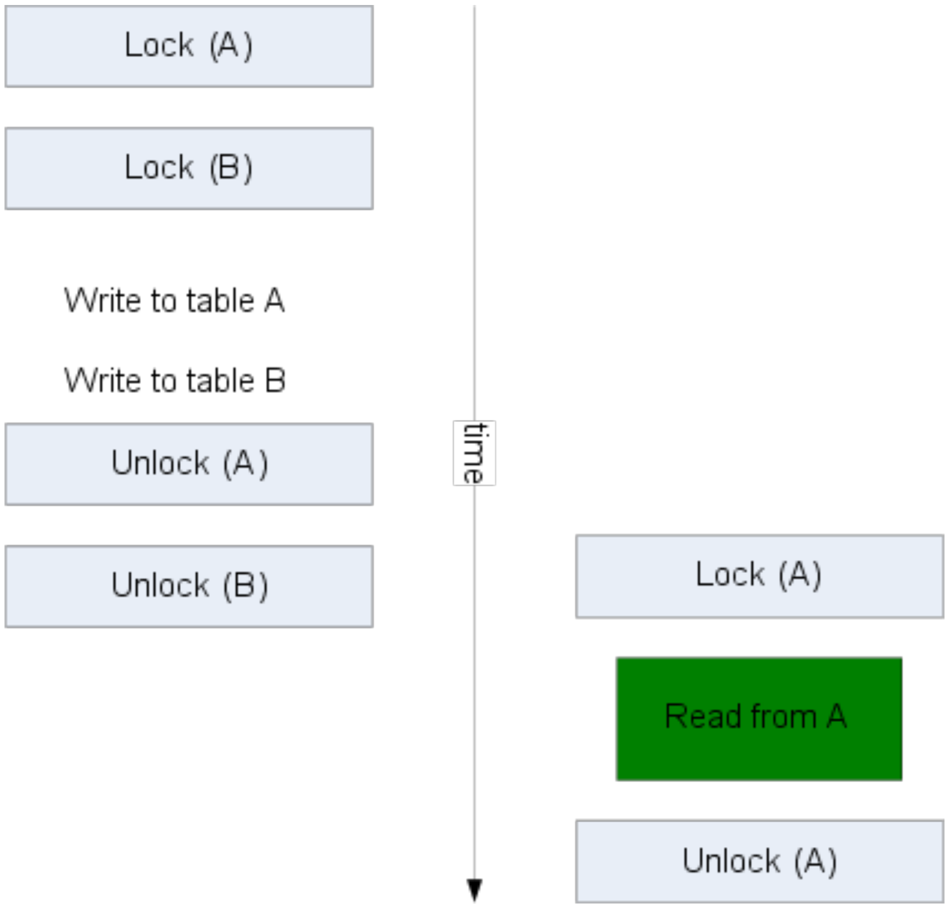
# Faulty Locking




- Locking alone does not ensure isolation!
- Changes to table A are visible before changes to table B – this is not an isolated transaction

# Two-Phase Locking

- After a transaction has released any locks, it may not acquire any new locks
- Effect: The lock set owned by a transaction has a "growing" phase and a "shrinking" phase



# Relationship to Distributed Comp

- At the heart of a TPS is usually a large database server
- Several distributed clients may connect to this server at points in time
- Database may be spread across multiple servers, but must still maintain ACID

# Conclusions

- Parallel systems evolved toward current distributed systems usage
- Hard to avoid failure
  - Determine what is reasonable to plan for
  - Keep protocols as simple as possible
  - Be mindful of common pitfalls