Agenda for Today

- MapReduce
- Project 1
- Replicated State Machine
Example Scenario

- Genome data from over 10 million users
  - O(100 MB) of data per user

- Goal: Analyze data to identify genes that show susceptibility to Parkinson’s disease
Other Example Scenarios

- Ranking web pages
  - $O(100 \text{ trillion})$ web pages

- Selecting ads to show
  - Clickstreams of over two billion users
Lots of Data!

Petabytes or even exabytes of data

- Impossible to store data on one server
- Will take forever to process on one server

Need distributed storage and processing
Desirable Properties of Soln.

- **Scalable**
  - Performance grows with # of machines

- **Fault-tolerant**
  - Can make progress despite machine failures

- **Simple**
  - Minimize expertise required of programmer

- **Widely applicable**
  - Should not restrict kinds of processing feasible
Distributed Data Processing

● Strawman solution:
  ◆ Partition data across servers
  ◆ Have every server process local data
  ◆ Gather and concatenate outputs

● Why won’t this work?

● Remember use cases:
  ◆ Identify genes indicating susceptibility to Parkinson’s
  ◆ Rank web pages based on their popularity
Distributed Data Processing

- Challenge: Inter-data dependencies

- To say if a gene is correlated with Parkinson’s
  - Need data from all users with that gene

- To compute popularity of a web page
  - Need content of all pages that link to it

- Implications:
  - No way to partition data
  - Must join outputs across partitions
MapReduce

- Distributed data processing paradigm introduced by Google in 2004
- Popularized by open-source Hadoop framework

- MapReduce represents
  - A programming interface for data processing jobs
    - Map and Reduce functions
  - A distributed execution framework
    - Scalable and fault-tolerant
MapReduce Execution

Partition → Map → Coalesce → Reduce

- (k₁, v₁), (k₂, v₂), ..., (kₙ, vₙ)
- (k₁, v₁)
- (kᵢ, vᵢ), (kⱼ, vⱼ)
- (a, b), (w, p), (y, r), (c, d), (y, z)
- (a, b), (a, q), (a, s)
- (k¹, v¹), (k², v²), (k³, v³), (k⁴, v⁴)
MapReduce: Word count

map(key, value): //filename, file contents
    for each word w in value:
        EmitIntermediate(w, "1");

reduce(key, list(values)): //word, counts
    int result = 0;
    for each v in values:
        result += ParseInt(v);
    Emit(AsString(result));
MapReduce: PageRank

- Compute rank for web page P as average rank of pages that link to P

- Initialize rank for every web page to 1

- Map(a web page W, W’s contents)
  - Output?

- Reduce(?, ?)
  - Output?
MapReduce: PageRank

- Compute rank for web page P as average rank of pages that link to P
- Initialize rank for every web page to 1
- Map(a web page W, W’s contents)
  - For every web page P that W links to, output (P, W)
- Reduce(web page P, {set of pages that link to P})
  - Output rank for P as average rank of pages that link to P
- Run repeatedly until ranks converge
MapReduce Execution

Partition → Map → Coalesce → Reduce

When can a Reduce task begin executing?
Synchronization Barrier

Partition: 
- \((k_1, v_1)\)
- \((k_2, v_2)\)
- \(\vdots\)
- \((k_n, v_n)\)

Map: 
- \((k_1, v_1)\)
- \((w, p)\)
- \((a, b)\)
- \((a, q)\)
- \((y, r)\)
- \((c, d)\)
- \((c, t)\)

Coalesce: 
- \((a, b)\)
- \((a, q)\)
- \((y, r)\)
- \((c, d)\)
- \((c, t)\)

Reduce: 
- \((k^1, v^1)\)
- \((k^2, v^2)\)
- \((k^3, v^3)\)
- \((k^4, v^4)\)
Fault Tolerance via Master

The diagram illustrates the process of fault tolerance in a distributed system, emphasizing the role of the master node. The master node initiates processes and distributes tasks among worker nodes. Key steps include:

1. The master forks a user program to initiate the process.
2. The master assigns maps and reduces tasks to worker nodes.
3. Worker nodes read input files.
4. Workers perform local writes during the map phase.
5. Workers perform remote reads during the reduce phase.
6. Workers write to output files.

The diagram also categorizes the process into phase: input files, map phase, intermediate files (on local disks), reduce phase, and output files.
Project 1

- **Part 1:** Map and Reduce for word count
- **Parts 2 and 3:** Modify Master to enable distributed processing and deal with failures

**Goals:**
- Familiarize with Go
- Reason about failures

**Reminders:**
- Declare GitHub ID and group on course web page
- Sign up for Piazza
Handling Master Failure

- If Master fails, can we start up a new Master and resume execution of job?
  - No!
    - Lost all state about execution status of tasks

- Must replicate Master state to tolerate failures
Replicating MapReduce Master

Master M1
Map tasks:
1 2 3 4

Task 3 assigned to W1

Done with task 1
Run task 3

Worker W1

Master M2
Map tasks:
1 2 3 4

Done with task 2
Run task 3

Worker W2
Replicating Bank Database

- One copy of account in SF, another copy in NY
- Clients send updates to both copies

Inconsistent replicas!

“Deposit $100”
$1,000
$1,100
$1,111

“Pay 1% interest”
$1,000
$1,010
$1,110
Synchronizing Replicas

- How to ensure replicas are in sync?
  - Apply updates in same order at all replicas

- Model every replica as a state machine
  - Given identical initial states, applying updates in same sequence results in same final state

\[ \text{Replicated state machine (RSM)} \]
Replicated State Machine

- Application oblivious to replication

- Replication of state enables higher availability
- Replicas talk amongst each other to sync
Replicated State Machine

- Replicate over time
  - Start a new replica after one goes down
  - Crash failures

- Replicate over space
  - Run multiple replicas simultaneously
  - Fail-stop failures
Implementing RSM

- Order updates based on (time of receipt, replica ID)

- Challenge: Clocks not in sync across replicas
Clock Syncing: Challenges

- Leverage GPS broadcasts?
  - Time from GPS accurate to about 1 microsecond
  - Power hungry and does not work indoors

- Correct for skew based on reference clock?
  - Clock drift
  - Unbounded network delay

Client

Server

Time of day?

2:50 PM
Cristian’s algorithm

1. Client sends a request packet, timestamped with its local clock $T_1$

2. Server timestamps its receipt of the request $T_2$ with its local clock

3. Server sends a response packet with its local clock $T_3$ and $T_2$

4. Client locally timestamps its receipt of the server’s response $T_4$

How can client use these timestamps to synchronize its local clock to server’s clock?
Cristian’s algorithm

Goal: At $T_4$, client sets clock $\leftarrow T_3 + \delta_{\text{resp}}$

Assume: $\delta_{\text{req}} \approx \delta_{\text{resp}}$

- Client estimates $\delta_{\text{req}} = (T_2 - T_1)$

Client sets clock $\leftarrow T_3 + \delta_{\text{req}}$
Cristian’s algorithm

Goal: At $T_4$, client sets clock $\leftarrow T_3 + \delta_{\text{resp}}$

- Client only knows $\delta$, not $\delta_{\text{resp}}$
- Client measures round trip time $\delta$
  \[ \delta = \delta_{\text{req}} + \delta_{\text{resp}} = (T_4 - T_1) - (T_3 - T_2) \]
- Assume: $\delta_{\text{req}} \approx \delta_{\text{resp}}$
- Client sets clock $\leftarrow T_3 + \frac{1}{2}\delta$
Further refinements

- Limitation of Cristian algorithm
  - Relies on timeserver availability

- NTP
  - Distributed ecosystem for time synchronization
  - Tolerates failures of network and time servers
  - Leverages heterogeneous clocks
Next Time ...

- Ordering events without synchronizing physical clocks
- Go to discussion section tomorrow for tutorial on Go