 Topics covered in this lecture

- Thread safety wrap-up
  - Synchronizers and summary
- Map Reduce

Semaphores

- Counting semaphores control the number of activities that can:
  - Access a certain resource
  - Perform a given action
- Used to implement resource pools or impose bounds on a collection

Frequently asked questions from the previous class survey

- ConcurrentHashMap
  - Does the lock operate over a consecutive space?
  - During resize operations can elements be added/removed?
- Latches:
  - Why not use a counter object, that is guarded by synchronous methods?

SYNCHRONIZERS

- Semaphores
  - Manage a set of virtual permits
    - Initial number passed to the constructor
    - Activities acquire and release permits
  - If no permits are available:
    - acquire blocks until one is available
    - The release method returns a permit to the semaphore
Semaphores are useful for implementing resource pools

- Block if the pool is empty
- Unblock if the pool is non-empty
- Initialize a semaphore to the pool size
- Acquire a permit before trying to fetch a resource from pool
- Release the permit after putting the resource back in pool
- Acquire blocks until the pool is non-empty

Binary semaphores

- Semaphore with an initial count of 1
- Can be used as a mutex with non-reentrant locking semantics
  - Whoever holds the sole permit holds the mutex

```
public BoundedHashSet<T> {
    private final Set<T> set;
    private final Semaphore sem;
    public BoundedHashSet(int bound) {
        this.set = Collections.synchronizedSet(new HashSet<T>());
        sem = new Semaphore(bound);
    }
    public boolean add(T o) throws InterruptedException{
        sem.acquire();
        boolean wasAdded = false;
        try {
            wasAdded = set.add(o);
            return wasAdded;
        } finally {
            if (!wasAdded) sem.release();
        }
    }
    public boolean remove(Object o) {
        boolean wasRemoved= set.remove(o);
        if (wasRemoved) 
            sem.release();
        return wasRemoved;
    }
}
```

Barriers

- Barriers are similar to latches in that they block a group of threads till an event has occurred
- All threads must come together at barrier point at the same time to proceed
- Latches wait for events, barriers wait for other threads

```
Barriers and dinner …

- Family rendezvous protocol
  - Everyone meet at Panera @ 6:00 pm;
    - Once you get there, stay there ... till everyone shows up
    - Then we'll figure out what we do next
```

```
Barriers

- Often used in simulations where work to calculate one step can be done in parallel
  - But all work associated with a given step must complete before advancing to the next step
  - All threads complete step \( k \), before moving on to step \( k+1 \)
```
**CyclicBarrier**

- Allows a fixed number of parties to rendezvous at a fixed point
- Useful in parallel iterative algorithms
  - Break problem into fixed number of independent subproblems
- Creation of a CyclicBarrier
  - Runnable cyclicBarrierAction = ...;
  - new CyclicBarrier(2, cyclicBarrierAction);

**Using Cyclic Barriers**

```java
class Solver {
    final int N;
    final CyclicBarrier barrier;

class Worker implements Runnable {
    int myRow;
    Worker(int row) { myRow = row; }
    public void run() {
        while (!done()) {
            processRow(myRow);
            try {
                barrier.await();
            } catch (BrokenBarrierException ex) {
                ...
            }
        }
    }
}

public Solver(float[][] matrix) {
    data = matrix; N = matrix.length;
    barrier = new CyclicBarrier(N, new Runnable() {
        public void run() {
            mergeRows(...);
        }
    });
    for (int i = 0; i < N; ++i)
        new Thread(new Worker(i)).start(); // DO NOT START THREAD in constructor.
    waitUntilDone();
}
```

**Exchanger**

- Another type of barrier
- Two-party barrier
- Parties exchange data at the barrier point
- Useful when asymmetric activities are performed
  - Producer-consumer problem
- When 2 threads exchange objects via Exchanger
  - Safe publication of objects to other party

**Thread Safety: Summary [1/4]**

- It’s all about mutable, shared state
  - The less mutable state there is, the easier it is to ensure thread-safety
- Make fields final unless they need to be mutable
- Immutable objects are automatically thread-safe
- Encapsulation makes it practical to manage complexity

**Thread Safety: Summary [2/4]**

- Guard each mutable variable with a lock
- Guard all variables in an invariant with the same lock
- Hold locks for the duration of compound actions
**Thread Safety: Summary** [3/4]

- Program that access mutable variables from multiple threads without synchronization?
  - Broken program
- Include thread-safety in the design process
  - Document if your class is not thread-safe
- Document your synchronization policy

**Thread Safety: Summary** [4/4]

- Rather than scattering access to shared state throughout your programs and attempting ad hoc reasoning about interleaved access
  - Structure program to facilitate reasoning about concurrency
  - Use a set of standard synchronization primitives to control access to shared state

---

**MapReduce: What we will look at**

- Contrast with other systems
- MapReduce Paper
- How to express programs using Hadoop MapReduce
- HDFS
- MapReduce Runtime
- Why?

---

**Cloud Computing**

- The volume of data that we produce has increased dramatically
  - IDC (International Data Corporation) estimates
    - 180 EB (10^14) in 2006
    - 1.8 ZB (10^21) in 2011
    - Roughly a disk drive per person!
    - 40 ZB by 2020
Some of the sources of this deluge
- New York Stock Exchange
  - 1 TB of new trade data every day
- Facebook
  - ~10^9 photos
- Internet Archive
  - Stores 2 PB of data … growing at 20 TB per month
- LHC produces 1.5 PB per year

Amount of data generated by machines will outpace what people produce
- Machine logs
- RFID readers
- Sensor networks
- Instruments
- Vehicle GPS traces
- IoT
  - 20-35 billion IoT devices are expected to be online in 2020

Hard disk capacities, seek rates, and transfer times
- 1990
  - 1 GB HDDs with a transfer speed of 4.4 MB/sec
- Now
  - 1 TB hard drives are common
  - But the transfer speed is just 100 MB/sec
  - Writing is even slower!

Data transfers can be improved by using multiple disks
- What if we use 100 disk drives?
  - Each holding 1/100th of the data
- We could have cumulative transfer speeds of up to 100 x 100 MB/sec or 10 GB/sec
- But isn’t using 1/100th of disk wasteful?
  - Not if you store a 100 different datasets on these disks
  - Provide shared access to the disks

But there’s more than just reading and writing from multiple disks in parallel
- Cope with hardware failures
  - As the number of components increase, so does the probability of failure
- Analysis tasks need to be able to combine data
  - Dataset is dispersed over multiple disks

What MapReduce provides …
- Programming model that abstracts the problem from disk reads and writes
- Transform the problem into computations over sets of keys and values
- Supports distributed processing on large datasets over a cluster of computers
But why not use databases with lots of disks? [1/2]

- Another trend in disk drives
  - Seek time is improving much slower than transfer rates
- If data access pattern is dominated by seeks?
  - It takes longer to read or write large portions of the dataset than streaming through it
  - Streaming through dataset operates at transfer speed

But why not use databases with lots of disks? [2/2]

- Updating a small proportion of records in the dataset
  - Traditional B-Tree works well
- For updating a majority of the dataset
  - B-Tree is less efficient than MapReduce which uses Sort/Merge to rebuild the dataset

MapReduce should be seen as being complementary to databases

- MapReduce is good for problems that access the entire dataset
  - Particularly ad hoc analysis
  - Write once, read many times
- RDBMS is good for point queries or updates
  - Dataset has been indexed for low-latency retrieval and update times
  - Read and write many times

Grid Computing/HPC systems

- Distribute work across a cluster of machines that access a shared file system
- Works well for predominantly compute-intensive jobs
  - Problem when access to large data volumes is needed
    - Network bandwidth is a bottleneck and compute nodes become idle

MapReduce tries to collocate data with the compute node

- Data Locality
  - Data access is fast since it is local
  - Conserves network bandwidth
- Implementations go to great lengths to conserve it
  - Model network topology

MPI (Message Passing Interface) gives great control to the programmer

- MPI requires explicit handling of the mechanics of data flow
  - In MapReduce, the mechanics of data flow is implicit
- MapReduce spares programmers from having to think about failures
  - Detect failures and schedule replacements on healthy machines
  - Done with a shared-nothing architecture
  - MPI programs have to deal with checkpointing and recovery
    - More control but difficult to write
Volunteer computing
- SETI@home
- Volunteers donate cycles not bandwidth
- MapReduce
  - Runs jobs lasting minutes or hours on trusted, dedicated machines with high-bandwidth interconnects
- Volunteer computing
  - Perpetual computations on untrusted machines
    - Highly variable connection speeds and no data locality

Volunteer computing
- SETI@home
- Volunteers donate cycles not bandwidth
- MapReduce
  - Runs jobs lasting minutes or hours on trusted, dedicated machines with high-bandwidth interconnects
- Volunteer computing
  - Perpetual computations on untrusted machines
    - Highly variable connection speeds and no data locality

Source of raw data at Google
- Crawled data
- Log of the web requests

Several computations work on this raw data to compute derived data
- Inverted indices
- Representation of the graph structure of web documents
- Pages crawled per host
- Most frequent queries in a day ...

Most computations are conceptually straightforward
- But data is large
- Computations must be scalable
  - Distributed across thousands of machines
  - To complete in a reasonable amount of time

Complexity of managing distributed computations can ...
- Obscure simplicity of original computation
- Contributing factors:
  - How to parallelize the computation
  - Distribute the data
  - Handle failures
MapReduce was developed to cope with this complexity:

- Express simple computations
- Hide messy details of:
  1. Parallelization
  2. Data distribution
  3. Fault tolerance
  4. Load balancing

MapReduce:

- Programming model
  - Associated implementation for:
  1. Processing & Generating large data sets

Programming model:

- Computation takes a set of input key/value pairs
- Produces a set of output key/value pairs
- Express the computation as two functions:
  - Map
  - Reduce

Map:

- Takes an input pair
- Produces a set of intermediate key/value pairs

Mappers:

- If map operations are independent of each other they can be performed in parallel
  - Shared nothing
- This is usually the case

MapReduce library:

- Groups all intermediate values with the same intermediate key
- Passes them to the Reduce function
Reduce function
- Accepts intermediate key I and
- Set of values for that key
- Merge these values together to get
- Smaller set of value

Counting number occurrences of each word in a large collection of documents

```java
map (String key, String value)
   //key: document name
   //value: document contents
   for each word w in value
      EmitIntermediate(w, "1")
```

```java
reduce (String key, Iterator values)
   //key: a word
   //value: a list of counts
   int result = 0;
   for each v in values
      result += ParseInt(v);
   Emit(AsString(result));
```

The contents of this slide set are based on the following references

- Jeffrey Dean, Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. OSDI 2004: 137-150