

## Announcements

- PA-1 due today (electronic submission)
- HW2 due on 11/12
- PA-2 is available online today
  - MapReduce programming on the cluster
- Quiz 1 regrading requests
  - Take to Gates 310, request in writing
  - Within a week from returning your quiz
- Guest lecture on Wednesday
  - Prof. Mendel Rosenblum on Virtual Machines

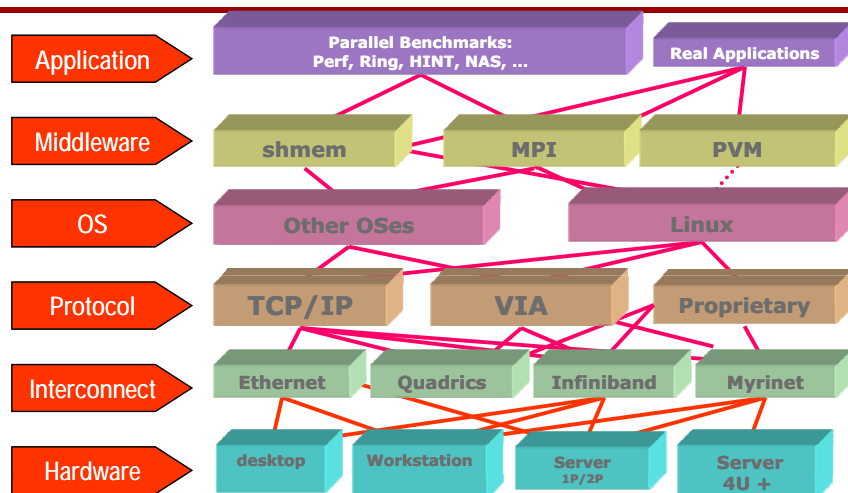
## Lecture 12:

### Clusters Software

Department of Electrical Engineering  
Stanford University

<http://eeclass.stanford.edu/ee282>

## Review: Cluster HW + SW



## Programming Clusters

- Choice #1: message-passing interface (MPI)
  - A library with basic communication elements (C, Fortran)
    - Point-to-point (send/receive) & collective communication (e.g., reduction)
    - Blocking & non blocking primitives
  - Highly portable, popular for scientific computing
- Choice #2: remote procedure calls (RPC)
  - A method to call a function/process on another machine
    - Typically implemented over networking (sockets)
    - Requires an RPC protocol for registering service, conventions, ...
  - Popular with commercial clusters & client/server applications
- Choice #3: MapReduce
  - A simple programming model that abstracts most complexity of programming clusters
  - Tradeoff: generality?

# MapReduce

- Programming model & runtime for processing large data-sets
  - E.g. Google's search algorithms
  - Goal: make it easy to use 1000s of CPUs and TBs of data
- Inspiration: functional programming languages
  - Programmer specifies only "what"
  - System determines "how"
    - Schedule parallelism, locality, communication, ...
- Ingredients:
  - Automatic parallelization and distribution
  - Fault-tolerance
  - I/O scheduling
  - Status and monitoring

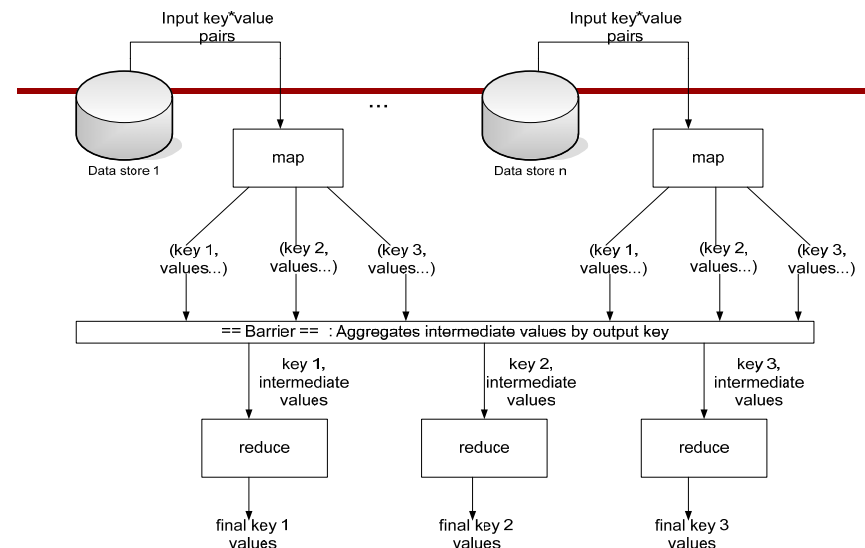
# MapReduce Programming Model

- Similar to functional languages (e.g. LISP) but with C syntax
  - High-level abstraction, hides most of the details, but scales well
  - Input & Output: each a set of key/value pairs
- Programmer specifies two functions:
  - `map (in_key, in_value) -> list(out_key, intermediate_value)`
    - Processes input key/value pair
    - Produces set of intermediate pairs
  - `reduce (out_key, list(intermediate_value)) -> list(out_value)`
    - Combines all intermediate values for a particular key
    - Produces a set of merged output values (usually just one)
  - User also specifies I/O locations and tuning parameters

# Example: Count word occurrences

```
map(String input_key, String input_value):
// input_key: document name
// input_value: document contents
for each word w in input_value:
    EmitIntermediate(w, "1");

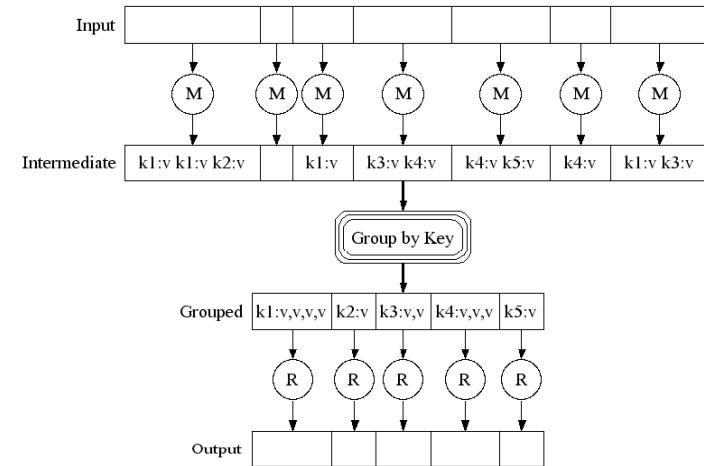
reduce(String output_key, Iterator intermediate_values):
// output_key: a word
// output_values: a list of counts
int result = 0;
for each v in intermediate_values:
    result += ParseInt(v);
Emit(AsString(result));
```



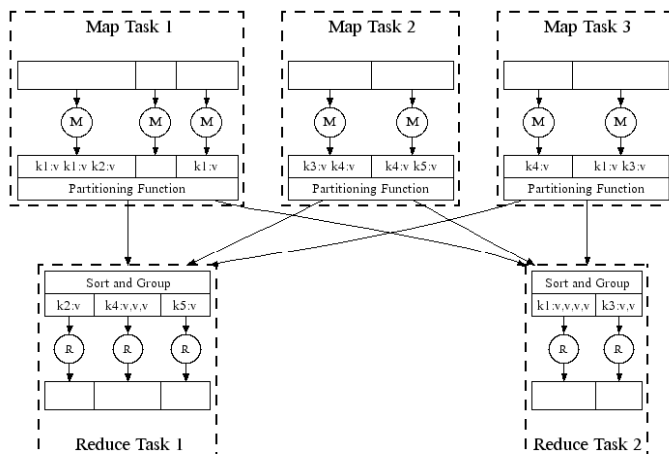
## Parallelism

- map() functions run in parallel, creating different intermediate values from different input data sets
- reduce() functions also run in parallel, each working on a different output key
- All values are processed independently
- Bottleneck: reduce phase can't start until map phase is completely finished.

## Parallel Execution Model



## Parallel Execution Model: Granularity



## Google's MapReduce Implementation

- Runs on Google clusters:
  - 100s/1000s of 2-CPU x86 machines, 2-4 GB of memory, local-based storage (GFS), limited bandwidth
- Implemented as a C++ library linked to user programs (on top of RPC)
- Scheduling/runtime system (aka master)
  - Assign tasks to machines: typically map tasks > machines
    - Often use 200,000 map/5,000 reduce tasks w/ 2000 machines
  - Minimizes time for fault recovery
  - Can pipeline shuffling with map execution
  - Better dynamic load balancing
- Other MapReduce implementations
  - Hadoop: an open-source, Java-based MapReduce framework
  - Phoenix: an open-source MapReduce framework for multi-core

## Locality

---

- Master program divides up tasks based on location of data: tries to have map() tasks on same machine as physical file data, or at least same rack
- map() task inputs are divided into 64 MB blocks: same size as Google File System chunks

## Fault Tolerance & Optimizations (1)

---

- In a machine with 1000s of nodes, failures will be common
- On worker failure:
  - Detect failure via periodic heartbeats
  - Re-execute completed and in-progress *map* tasks
  - Re-execute in progress *reduce* tasks
  - Task completion committed through master
- Not dealing with master failures so far...
- Optimization for fault-tolerance and load-balancing
  - Slow workers significantly lengthen completion time
    - Due to other jobs on machines, disk with errors, caching issues, ...
    - Other jobs consuming resources on machine
  - Solution: Near end of phase, spawn backup copies of tasks
  - Whichever one finishes first "wins"

## Fault Tolerance & Optimizations (2)

---

- Scheduling for locality
  - Asks GFS for locations of replicas of input file blocks
  - Map tasks schedule to node with copy of input data
    - Effect: Thousands of machines read input at local disk speed
- Skipping "bad" records in input
  - Map/Reduce functions sometimes fail for particular inputs
    - Best solution is to debug & fix, but not always possible
  - Solution
    - On seg fault, notify master about the record that fails
    - If master sees 2 failures for same record, it notifies workers to skip
  - Effect: Can work around bugs in third-party libraries
- Other
  - Compression of intermediate data

## MapReduce Implementation

---

## Underlying Technologies (1)

- Ethernet – IEEE 802.3 standard
  - The common physical layer of the networking stack
  - Low-level signaling and wiring to connect two nodes
- IP – Internet protocol
  - Defines an addressing scheme for computers
  - Encapsulates internal data in a “packet”
  - Includes enough info for the data to tell routers where to send it
  - Does not provide reliability

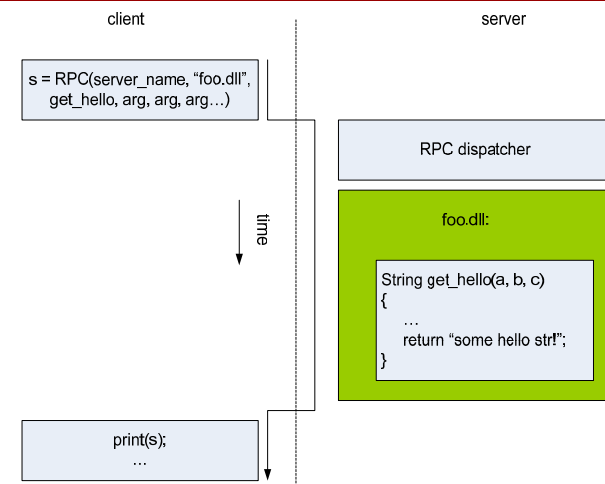
## Underlying Technologies (2)

- TCP – Transmission control protocol
  - Builds on top of IP
  - Introduces the concept of a connection
  - Provides reliability and ordering of communication
    - End to end flow control and congestion control
- Sockets
  - The basic networking API for programmers
    - A set of C function calls
  - Provides a two-way “pipe” abstraction between two applications
  - Client creates a socket, and connects to the server, who receives a socket representing the other side
  - Often, but not necessarily, implemented on top of TCP/IP

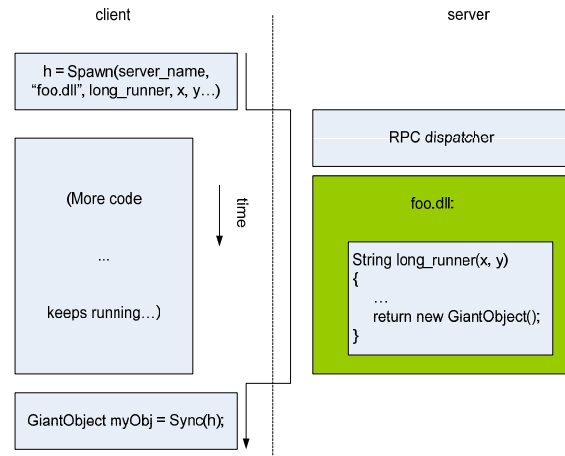
## Remote Procedure Calls (RPCs)

- RPC: remote procedure calls
  - Aka: Remote Method Invocation (RMI)
  - A function call to another machine (or process)
  - Implemented over networking (sockets)
- Using RPC, one can implement a MapReduce system
  - To communicate between master nodes and worker nodes
- RPC protocols
  - RPC defines a communication protocol
    - To register/findservices, pass arguments, specify output buffers, ...
  - Give protocol, server/client programs can be developed independently

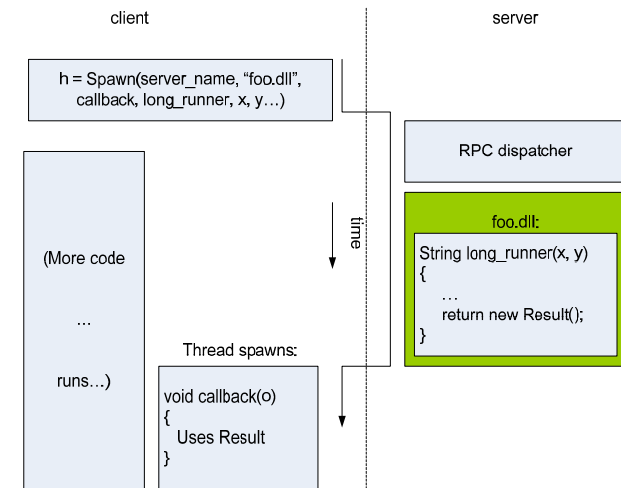
## Synchronous RPC



## Asynchronous RPC



## Asynchronous RPC 2: Callbacks



## Distributed Filesystems

- Support access to files on remote servers
- Must support concurrency
  - Make varying guarantees about locking, who “wins” with concurrent writes, etc...
  - Must gracefully handle dropped connections
- Can offer support for replication and local caching
- Different implementations sit in different places on complexity/feature scale

## Examples of DFS

- NFS – network file system
  - DFS that presents a standard UNIX FS interface
    - Network drives are mounted into local directory hierarchy
  - Implemented over UDP (not TCP)
  - Original implementations were stateless; now clients can lock
  - Clients can also cache data (requires consistency protocol)
- GFS – Google file system
  - Files stored as chunks (64MB)
  - Reliability through replication (3 servers/chunk)
  - Single master to coordinate access, keep metadata
  - No data caching (assumption large datasets/streaming)

## MPI Summary

---

## MPI: Message Passing Interface

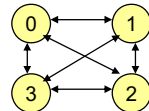
---

- Software standard for distributed memory parallel programs
  - Library of routines: communication, utilities
  - MPI is a specification, not a specific implementation
  - Portable & fast
  - Parallel model: MPMD with explicit distributed memory
- MPI covers
  - Point-to-point communication (send/receive)
  - Collective communication
  - Support for library development
- MPI does not cover
  - Fault tolerance
  - Parallel/distributed operating system
- MPI tutorials: <http://www-unix.mcs.anl.gov/mpi/tutorial/index.html>
- MPI standard: <http://www-unix.mcs.anl.gov/mpi/>

## MPI Application Environment

---

- The elements of an application are
  - N processes, numbered 0 through N-1
  - Communication paths between them
- MPI\_COMM\_WORLD
  - The set of processes plus the communication paths
  - MPI\_COMM\_SIZE(): the number of processes (N)
  - MPI\_COMM\_RANK(): the specific number of a process
- Compiling & running (not standard)
  - mpicc -o hello hello.c
  - mpirun -np 4 ./hello
    - 4 processes but not necessarily 4 processors...



## MPI\_Send(): sending a message

---

- The first important MPI function:
  - `MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)`
- Buf: address of send buffer (where the message is)
- Count: number of elements
- Datatype: data type of send buffer elements
- Dest: process id of destination process
- Tag: message tag
- Comm: communicator (group of communicating processes)

## MPI\_Recv(): receiving a message

- The other important MPI function:
  - `MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)`
- Buf: address of receive buffer
- Count: size of receive buffer in elements
- Datatype: data type of receive buffer elements
- Source: source process id or `MPI_ANY_SOURCE`
- Tag: message tag or `MPI_ANY_TAG`
- Status: status object (indicates sender and tag)
  - Receiver can recover source, length, error, etc
- Note: sender & receiver must match
  - Count and datatype
  - Tag and communicator

## MPI Program Template

```
main(int argc, char *argv[])
{
    MPI_Init (&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    /* Data distribution */ ...
    /* Computation & Communication*/ ...
    /* Result gathering */ ...
    MPI_Finalize();
}
```

## Example: Array Increment (1/2)

```
#include <stdio.h>
#include "mpi.h"
int main(int argc, int argv ) {
    int rank, size, i;
    MPI_Status status;
    int A[100];
    MPI_INIT( &argc, &argv );
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );
    MPI_Comm_size( MPI_COMM_WORLD, &size );
    if (rank == 0) {
        read_array(A, "input.file", 100); /* read array A*/
        MPI_send(A+25, 25, MPI_INT, 1, 0, MPI_COMM_WORLD);
        MPI_send(A+50, 25, MPI_INT, 2, 0, MPI_COMM_WORLD);
        MPI_send(A+75, 25, MPI_INT, 3, 0, MPI_COMM_WORLD);
    } else {
        MPI_Recv(A, 25, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);
    }
}
```

## Example: Array Increment (1/2)

```
/* computation without any communication */
for (i=0; i<25 i++) A[i]++;

if (rank != 0) MPI_send(A, 25, MPI_INT, 0, 1, MPI_COMM_WORLD);
else {
    MPI_Recv(A+25, 25, MPI_INT, 1, 1, MPI_COMM_WORLD, &status);
    MPI_Recv(A+50, 25, MPI_INT, 2, 1, MPI_COMM_WORLD, &status);
    MPI_Recv(A+75, 25, MPI_INT, 3, 1, MPI_COMM_WORLD, &status);
    write_array(A, "output.file", 100); /* write array A*/
}

MPI_Finalize( );
return 0;
}
```

## Other MPI Features

---

- Collective communication
  - Barrier, broadcast, gather, scatter, all-to-all, reductions
- Non blocking send/receives
- Communication modes (synchronous, buffered, ready,...)
- Watch out for
  - Deadlock and livelocks on send/receive
    - Order and send/receive semantics matter
  - Overhead of communication
    - Try to overlap it with computation