So, What Actually is a Cloud?

Dan Stanzione

Deputy Director, TACC

UT-Austin

Originally from Arizona State Cloud Computing Course, Spring 2009

(Jointly taught by Stanzione, Santanam, and Sannier)

ARIZONA STATE UNIVERSITY

IRA A. FULTON SCHOOL OF ENGINEERING

HIGH PERFORMANCE COMPUTING

You’ve heard about what clouds can *do*, and how they change the game. But what, technically speaking, are they?

- Terminology
- Some definitions of a cloud (from others)
- A working definition, and some history and background

Some Terminology:

- Cloud Computing
- Grid Computing
- Utility Computing
- Software As A Service (SaaS)
- On-demand Computing
- Distributed Computing
- Cluster Computing
- Parallel Computing
- High Performance Computing
- Virtual Computing (Virtualization)
- Web Services
- A little older:
  - Client-server computing
  - Thin clients
- *All these terms are batted around in trade publication, but what do they mean?*
  - *If someone asked you to design/deploy a cloud, a cluster, and a grid, what would you order, and what software would you run on it?*

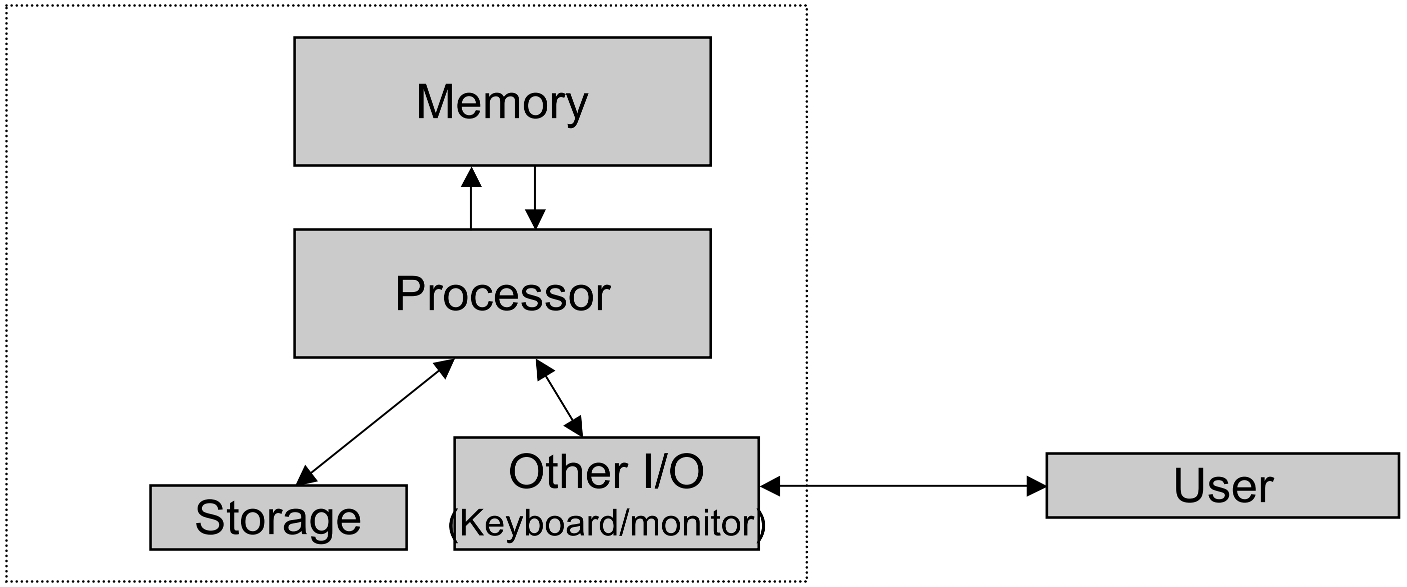
Unfortunately, there aren't very many universally accepted definitions to those terms

- Although I believe some are clearly wrong, and some are more "right" than others.
- So, we'll try and answer this question in a few ways:
  - By providing a framework and some background
  - By looking at definitions from others
  - Then by providing definitions of our own, from the more concrete to the slightly more fuzzy.

A Framework for Talking About Computing

- Unlike most scientific disciplines, computer science lacks firm taxonomies for discussing computer systems.
  - Speaking as a computer scientist, I put this in the list of things that differentiate "computer science" from "real science"… can't simply fall back to first principles; there are no Maxwell's Equations of Computer Science, nothing has a latin name.
  - As a result, almost every discussion of computer systems, even among most faculty, is hopelessly confused.
  - Because computing also has a trade press and a market, things are much, much worse. Terms are rapidly corrupted, bent, misused and horribly abused.
    - There is no Dr. Dobb's Journal of Chemical Engineering or Molecular Biology, and no Microsoft of High Energy Physics offering the ***myFusion*** ™ *home thermonuclear reactor with free bonus Bose-Einstein Condensates Beta!.*

Computing 1.0 – **John Von Neumann, June 30th, 1945**



- The dotted line box defined the modern notion of a “computer”
- The connection between memory and processor is known as the “Von Neumann Bottleneck”; more on that later

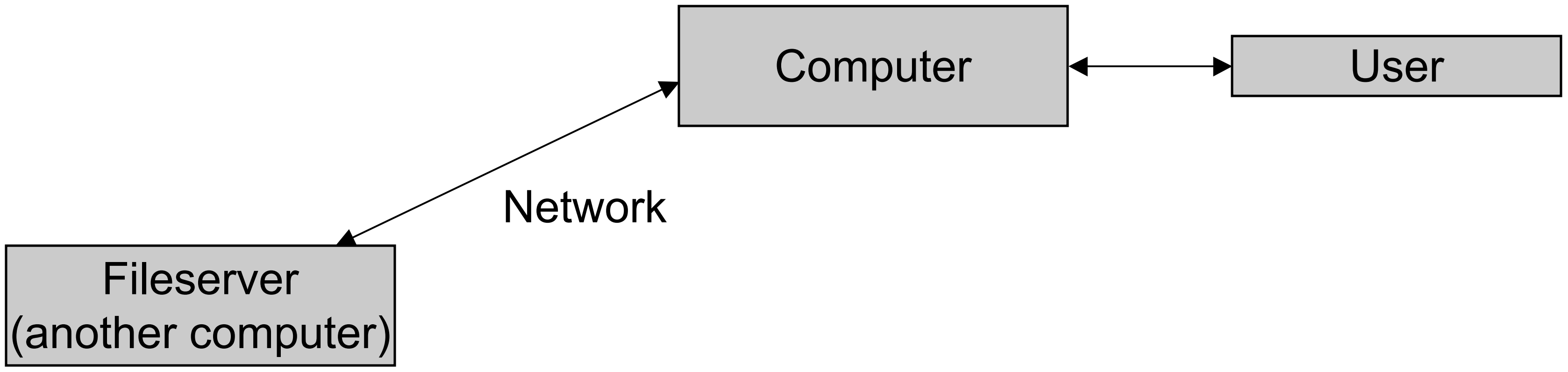
Computing 2.0

a.k.a.

Hey, maybe there is more than one computer in the world!

a.k.a

Maybe Computers shouldn't just talk to people, they should talk to each other!



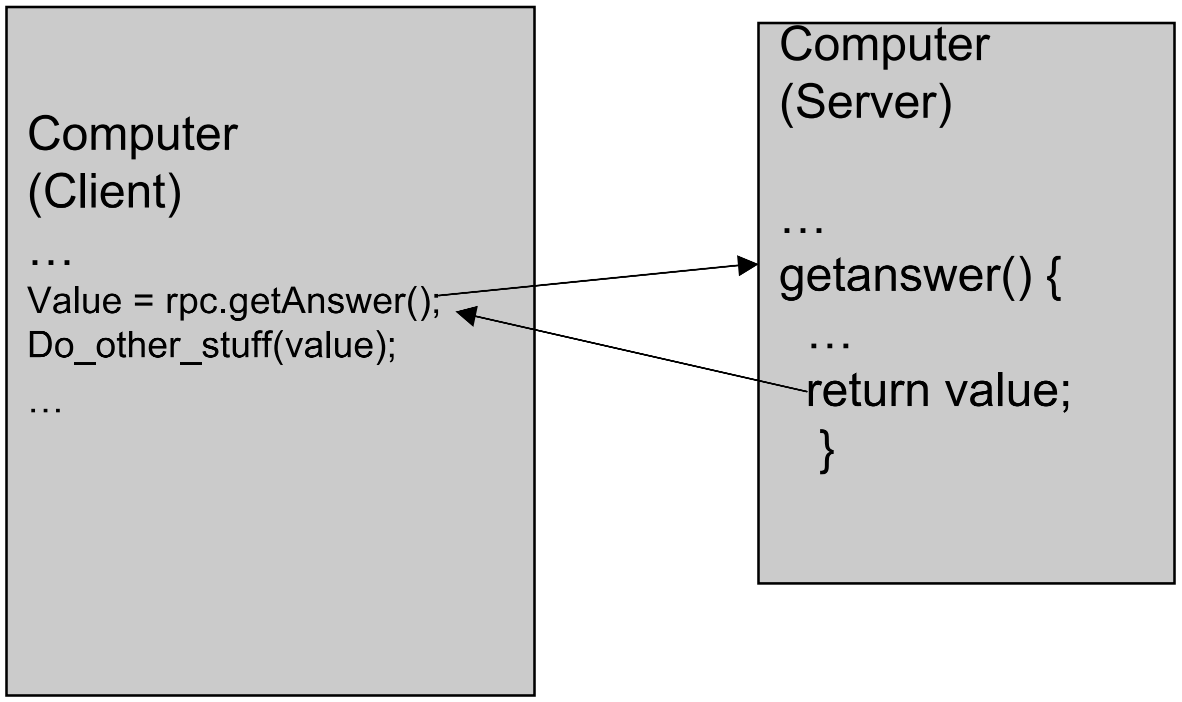
To solve my problem, I might need *more than one* computer to do something. In this model, one computer provides a service to another (early on, this usually meant making files available).

One might argue this is where the concept of a reliable computer system ended once and for all…

Computing 2.1 **Distributed Computing Environments**

- Computing 2.0 kicked off the concept of having multiple computers interact to perform tasks, or groups of tasks, for users.
- This notion rapidly got extended from fileservers to the concept of the *remote procedure call*, which is truly the parent concept of all modern distributed computing
- The basic idea of RPC is that code running on one computer makes a call to, and receives a response from, code running on another computer. The *client-server* architecture largely grew from this concept.
- This seems simple, but was a great leap forward in programming model; one crucial, non-obvious side effect was the introduction of *concurrency*; with more than one computer, several things can happen at the same time.

Computing 2.1 **Distributed Computing Environments**



The Fork() in the Road

- From about the time distributed computing environments existed, computer systems forked into two (-ish) camps, both dealing with limitations of the computer system as we know it.
  - In the *technical* computing community (science and engineering simulation), the basic problem was that the processors were too slow to solve enough problems.
  - In the *enterprise* computing community (business processes, offices and classrooms, etc.) the problem was that the servers couldn’t satisfy enough clients, and couldn’t do it reliably enough.
- Both attacked the problem through concurrency.

The Fork() in the Road

- The S&E computing community began down the road of **supercomputing**, and eventually settled on **parallel computing** as the answer, resulting in today's **high performance computing** systems
- The enterprise computing community began down the path of failover and redundancy, resulting in today's massively parallel **utility computing** systems, of which the ultimate evolution may be the **cloud**.
  - While they went different paths, they (sort of) ended up in the same place, but with very different worldviews.
  - As a result, there are a few key technical differences between the modern cloud and supercomputer; despite their many similarities, they solve very different problems.

Fork #1: Scientific Computing

- The fundamental problem: Simulations are too big, and either take too long or don't fit on the machine.
  - Solution attempt #1: The supercomputer, 70s and 80s style.
    - Solve the problem at the circuit level: build faster processors
    - Ran into barriers of engineering cost, and economies of scale.
  - Solution attempt #2:
    - If one processor delivers X performance, wouldn't 2 processors deliver 2X performance? (Well, no, but it's a compelling idea and we can come close, often).
  - **Parallel Computing** is the core concept of all modern high performance computing systems, and is the simple idea that:
    - **More than one processor can be used to perform a single simulation (or program).**
  - By contrast, distributed computing involves multiple computers doing *different* tasks.

So, supercomputers now…

- Parallelism is a simple idea, but in practice, doing it effectively has been a huge challenge shaping systems and software for decades…

- The term **supercomputer** is no longer used to reference any particular single architecture, but rather is used for the systems that deliver the highest performance, defined in this context as:
  - Delivering the largest number of floating point operations per second (FLOPS) to the solution of a *single problem* or a *single program*
  - (hence, Google’s system does not appear on the top 500 supercomputer list, as it solves many small problems, but can’t be focused on one large one).

Granularity

- An important factor in determining how well applications run on any parallel computer of any type is the **granularity** of the problem.
- Simply put granularity is the measure of how much work you do computing before needing to communicate with another processor (or file, or network).
  - A “coarse grain” application is loosely coupled, i.e. can do a lot of work before synchronizing with anyone else (think SETI@Home; download some data, crunch for an hour, send the result).
  - A “fine grain” application does only a few operations before needing to synchronize… like the finite difference example in an earlier slide.
- In general, for parallel computer design, we care a *lot* about solving fine grain problems, as most S&E simulations (and true parallel codes) are pretty fine grain.

Notes About Cluster Architecture (distinguishing from Clouds)

- Because clusters are about parallel computing, lots of investment goes into the network
  - Latency on Saguaro between any two nodes: **2.6 microseconds**
  - Latency between my desk and www.asu.edu: **4,251 microseconds**
  - 4.2 ms is fast enough for humans, but not for parallel codes).
- Because clusters run one large job, the storage system usually focuses (at considerable expense) on delivering bandwidth from one big file to all compute nodes.
- Keep these two things in mind
- Clusters, supercomputers, and any other architecture focused on solving one large problem really fast is what typically falls in the category ***High Performance Computing***

Cluster Computing

- In the scientific computing community, whenever we build one of these parallel systems, particularly in the distributed memory or hybrid model, and we build it using *primarily* commodity components, we call this a **cluster computer.**
- Officially, it's called a Beowulf Cluster, if:
  - It's built from commodity components
  - It's used to solve problems in parallel
  - It's system software is open source

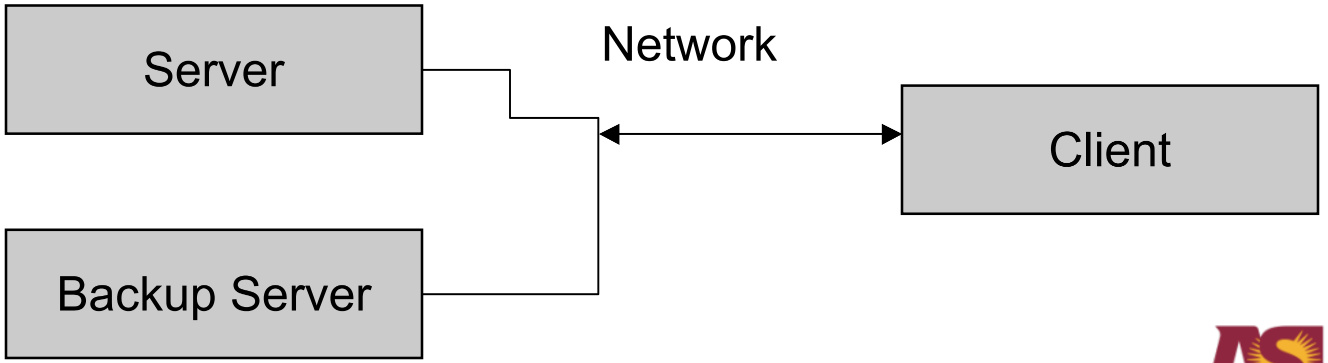
  (This is directly from the guy who coined the phrase "Beowulf Cluster")
- Unofficially, any large deployment of identical machines with identical software images now gets called a cluster…

Fork #2: High Availability

- While the technical computing folks were delivering parallel computers, enterprise computing folks were evolving their own distributed systems around a separate set of problems:
  - Servers must handle many clients
    - Database servers processing thousands of transactions
    - Web servers with millions of hits
    - Note in these environments, it's effectively a lot of little tasks that run independently, though there still is synchronization (e.g. the database must be consistent, but one query doesn't talk to another).
  - If one server handles many clients, that server is pretty important.

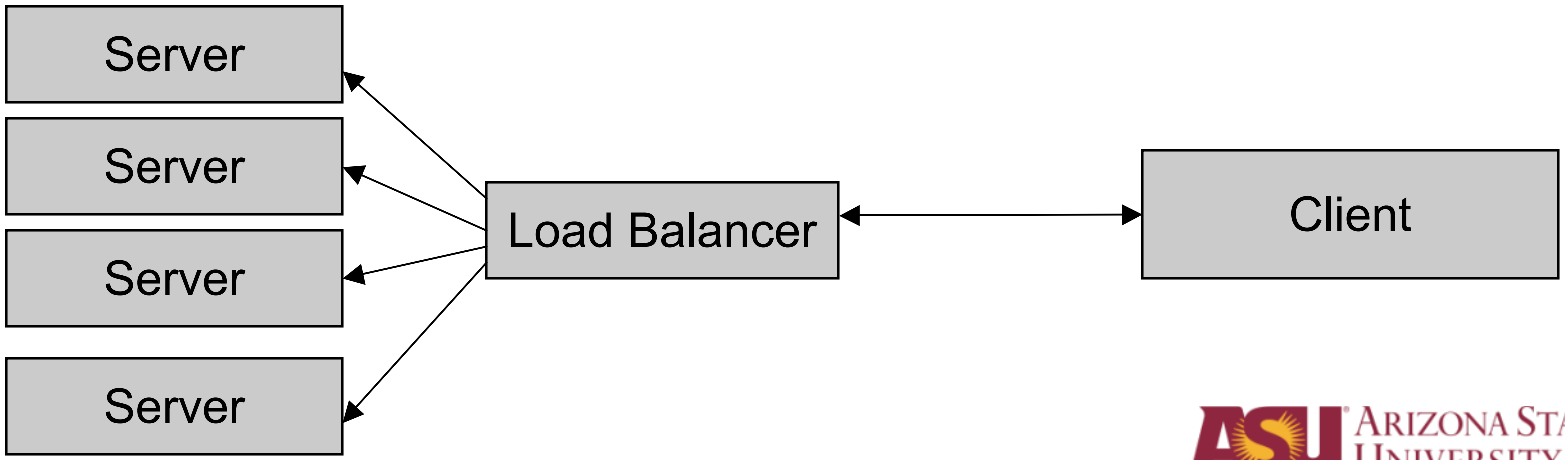
High Availability

- “Cluster Computing” got its start in the enterprise world in a failover pair.
  - If one server may fail, design an identical one that can pick up it’s load.
  - A Cisco firewall “Cluster” means two machines (the Ranger HPC cluster has 63,000 processors).



Load Balancing

- Database servers drove SMP designs for quite a while (as shared memory was useful for maintaining a consistent DB state).
- Eventually, they hit the same wall scientific users did
  - Machines don't have enough bandwidth, but two should have more than one.
  - If a failover pair has a second server that does the same job, why not use it all the time? Why not have more than two?



Enterprise Concurrency

- In this model, the *servers* don’t synchronize much, but the *storage* does.
  - So, at considerable expense, in the enterprise world, the SAN (Storage Area Network) came into existence, to allow concurrent servers the ability to talk to coherent, shared storage.
  - (Remember this for a few slides too).

Enterprise writ large

- As the enterprise architecture evolved, the endpoint became the deployment of massive “clusters”
  - Large sets of servers, with similar hardware
  - Maybe many software images
  - Network latency not an issue
  - Coherent storage more important than fast storage (and focus on small transactions).

Where do Grids fit in?

- Grid computing emerged from the scientific/academic computing community in the late 90's.
  - The fundamental idea is that computing (cycles) should be a commodity, delivered like the power grid:
    - Plug in anywhere and get the same thing
    - No one cares where the power comes from

Grid Computing

- The realization of grid was a software layer which:
  - Provided access to a set of resources with no central administrative control
  - Allowed single authorization to all resources
  - Transparently (?) moved work among resources in the grid.
- In practice, a few more restrictions.
- Globus, Condor, World Community Grid, Open Science Grid among largest examples.

Grid in Practice

- "The Grid", at least the Globus version, eventually merged with Web Services (SOAP). Much in common:
  - Need to pass credentials
  - Need to discover resources dynamically
  - Need for a standard interface to start work and receive results.
- Grid was hyped for about 5 years; the word came to mean just about anything connected to a network (Sun Grid Engine, for instance, manages clusters; since it's central administrative control, it's not a grid).
- Grid for business was replaced by web services, since really they wanted seamless access to distributed systems, not to share resources.
- Cloud hype has now effectively killed grid hype, because most folks can't tell the difference…

Grid Limitations

- Grids solve some problems really well, but…
  - Code must move from users to random places in the grid
  - *Data* must move from users to random places in the grid
  - Parallel simulation is all about latency; grids care not at all about latency.
- Grids are great when:
  - You have problems that are coarse grain (little need for synchronization)
  - You have problems that *don't* need a lot of data
  - The LHC (large hadron collider) will fill the world's grid for years:
    - Take a small amount of LHC data, process it on 1 processor for several hours, repeat millions of times; perfect grid problem.
- One problem with volunteer grids is the quality-of-service is low and unpredictable. Hence there soon came…

High Throughput Computing

- HTC is about turning around lots of smaller processing tasks quickly. HTC systems solve grid-friendly problems, but in a predictable QOS way.
- Architecturally, HTC is a cheap cluster (or a dedicated grid); by lots of identical servers with a cheap network, cheap shared storage, and solve many non-parallel problems on it over and over.
  - E.g. have 5,000 students submit Matlab runs repeatedly.
- Much cheaper than HPC systems, but can't solve parallel problems well.

Software as a Service

- Technically, SaaS is a pretty simple concept
  - It's the evolution of client-server to its logical end:
    - Server provides the application
    - Client requests application, gets interface from server, user interacts with app through client.

- The genius of SaaS isn't technical, it's the business model:
  - Your product is not a shrink-wrapped box, it's a service you purchase again and again.
  - Think carefully… what's Microsoft's business model if they ever write a perfect version of Windows and Office that never needs an upgrade? It's not like software wears out…
  - SaaS turns a one time purchase into a long time subscription/revenue stream

Virtualization

- When writing client-server apps, web service apps, SaaS apps, etc., you sure seem to need a lot of servers.
  - Turns out, for many software models, the computer itself (the server) is a pretty handy unit of abstraction.
    - You don't need a *physical* computer, you just need something that behaves like one; has a name, has an OS, has memory and storage, runs your code.
  - We haven't figured out higher level language abstractions in programming to make code portable and migratable, but we *can* just abstract the entire computer
- Virtual servers do just that…
- One advantage of this is you can consolidate servers, but much more important, you don't need a *specific* server.
  - Run on any hardware, without needing to customize/reinstall/debug
  - Run any place
  - Move when the hardware fails
- So, now that virtualization works well, the next logical step is:

Utility Computing

- OK, virtual servers run all your apps. No matter how you built them, they all need the server abstraction. The hardware is generic (pretty much an HTC cluster).
- So, let’s build a market for servers
  - Build massive datacenters
  - Sell “servers” by the hour, on demand, at varying scales
  - Think early versions of Amazon’s Elastic Compute Cloud
- Still can’t do big parallel problems, or big data problems

Umm, wasn't this talk about Clouds?

- So, now we live in a SaaS, Web Service, load balanced, failover world, where servers are virtual and offered through utility computing as a commodity.
- Is this a cloud?
- Most would say yes…
- I still say no.
- Let's hear from others…

Cloud definitions from the internet cloud…

- From “Twenty-one experts define clouds”:

- "**Cloud computing** is **one** of those catch all buzz words that tries to encompass a variety of aspects ranging from deployment, load balancing, provisioning, business model and architecture (like Web2.0). For me the simplest explanation for **cloud computing** is describing it as, "internet centric software."

- "I view **cloud computing** as a broad array of web-based services aimed at allowing users to obtain a wide range of functional capabilities on a 'pay-as-you-go' basis...”
  - **Jeff Kaplan**

- "Clouds are vast resource pools with on-demand resource allocation. …Clouds are virtualized…Clouds *tend* to be priced like utilities”
  - **Jan Pritzker**

Cloud definitions from the internet cloud…

- "I think 'Clouds' are the next hype-term for the next year or two. People are coming to grips with Virtualization and how it reshapes IT, creates service and software based models, and in many ways changes a lot of the physical layer we are used to. Clouds will be the next transformation over the next several years, building off of the software models that virtualization enabled."
  - **Douglas Gourlay**

- "The way I understand it, "**cloud computing**" refers to the bigger picture…basically the concept of using the internet to allow people to access services. According to Gartner, those services must be 'massively scalable' to qualify as true '**cloud computing**'. So according to that definition, every time I log into Facebook, or search for flights online, I am taking advantage of **cloud computing**."

Cloud definitions from the internet cloud…

- "Clouds are the new Web 2.0. Nice marketing shine on top of existing technology. Remember back when every company threw some ajax on their site and said “Ta da! We’re a web 2.0 company now!”? Same story, new buzz word.

  - Damon Edwards

- SaaS is one consumer facing usage of cloud computing… Put simply cloud computing is the infrastructural paradigm shift that enables the ascension of SaaS."
  - **Ben Kepes**

So, clouds are big piles of other people's machines, plus virtualization.

I think we can do a little better… let's dig a little deeper into the cloud that really mattered first.

The Google Cloud

- Cloud didn't make sense for me until I understood what Google did, and how it really changed things.
- Yes, Google's cloud is a big, enormous pile of servers.
- No, they aren't virtual, but they figured out early on that no server would be fast enough or reliable, so while not virtual, they did make them dispensable (any one can fail and it's not a problem).
- What they added to this equation, and everyone else followed on, is two key innovations that make big piles of servers solve big problems.

Innovation #1: The storage approach

- Once upon a time (1986), a guy named Danny Hillis (Thinking Machines) decided the way to solve the Von Neumann bottleneck (remember that?) was to build "processor in memory"; essentially, distribute processing power throughout all the RAM, to give you lots of concurrent access points.
- Google figured out that at "internet-scale", the real bottleneck isn't to memory, it's to the massive datasets *on disk*. They built "processor-in-storage"
  - Give each server some storage
  - Spread data across lots of servers
  - Don't build one big shared filesystem.
  - Do what you will do to the data *before* you move it off the server
  - The magic is in their distributed filesystem, borrowing concepts from parallel filesystems, but using the local processors.

Innovation #2:

- Google figured out that “server” wasn’t the abstraction you need, because to scale up client server apps, you need to be able to add servers; lots of them; without changing the code.
- Google adopted a map-reduce strategy in conjunction with their filesystem:
  - Create tasks that operate on data
  - Move the tasks to the servers where the data is
  - As you add more data, create more copies of the code
  - Send results only at the end, without ever moving most of the data.
- This new, higher level API changes everything…
- The Google cloud is not about computation in the traditional sense, it’s about solving the loosely coupled data-intensive problems that HPC systems do not.

Summing Up…

So what’s a Cloud?

Characteristics We're Sure About in Clouds

- Users don't physically touch them
  - *Remote*
- They can do big things; some notion of "internet scale"
  - *Scalable*
- Clouds can move resources between applications
  - *Dynamic*

Characteristics We Think **We're Sure About in Clouds**

- You don't use a particular computer, you use an abstraction of a computer
  - *Virtual*
- They offer an interface with a higher level of abstraction than a simple server.
  - *High Level API*
- They are focused on loosely synchronized, coarse grain parallel tasks
  - *Loosely Coupled*

Characteristics people might argue with me about **in clouds**

- Clouds require a large scale, distributed storage system that abstracts how data is distributed and accessed.
- The most important aspect of a cloud is the way it provides concurrent access to large data volumes… in essence:
  - ***A cloud isn't about computing; A cloud *is* smart data.***

Cloud Architecture

- Lots of systems
- A storage mechanism that distributes data among the systems, with no large central storage
- An API that abstracts systems, and moves work to data.

A Controversial End

- Many people think of clouds as just getting rid of the hardware and outsourcing your datacenters.
- Clouds can do this, but so can utility computing of virtual servers (it's like using an HPC system to solve a grid problem; you're swatting flies with a bazooka).
- What HPC did for big simulation, clouds do for big data… and it's the dawn of the age of big data! In science, business, and everywhere else.
- Master the cloud, master large data, world domination will surely follow.

Let's figure out how to program them…

Introduction to Map/Reduce Programming

Raghu Santanam
Adrian Sannier
Dan Stanzione

Some materials adapted from slides by

- Christophe Bisciglia, Aaron Kimball, & Sierra Michels-Slettvet, Google Distributed Computing Seminar, 2007
- Jimmy Lin, University of Maryland

ASU Arizona State University
Ira A. Fulton School of Engineering
High Performance Computing

AGENDA

- Quick Introduction to Functional Programming
- Hadoop Map/Reduce Framework

FUNCTIONAL PROGRAMMING (FP)

- Lambda Calculus
  - Developed by Alonzo Church
  - A foundation for functional programming
- Functional programming deals exclusively with functions
- No side effects!
  - Variables can only be assigned to once & *cannot be modified*
  - Thus input data to a function is never changed
  - New data is created as output
- *This is a big difference from imperative programming (what you are used to)*

FUNCTIONAL PROGRAMMING LANGUAGES

- ML
- Haskell
- Erlang
- Lisp
- Mathematica
- Prolog

- Erlang is very popular in Telecom systems
- Developed by Ericsson, it is now used by major companies including T-Mobile, Amazon (SimpleDB),
- Used in Facebook Chat and Yahoo (Delicious) as well

WHAT'S SO GREAT ABOUT FP?

- Original data is always preserved!
- Order of execution does not matter
- Concurrency issues are handled without special constructs
  - No semaphores/synchronizations!
  - Thread safe by default!
- Hot deployment (deploy updates at run time) is an easy possibility
- You can pass functions as arguments

FP WITH MATHEMATICA

- We will use mathematica syntax to show examples
- You can try these on mathematica
- Define function ***fm*** and ***fn*** as follows:

fn=Function[{u,v},u^2+v^2]

Input: fn[4,5]
Output: 41

Input: fn[2,2]
Output: 8

fm=Function[{u},fn[u,2*u]]

Input: fm[2]
Output: 20

Input: fm[fn[2,2]]
Output: ??

Input:
fn[fm[2],fm[fn[2,2]]]
Output: ??

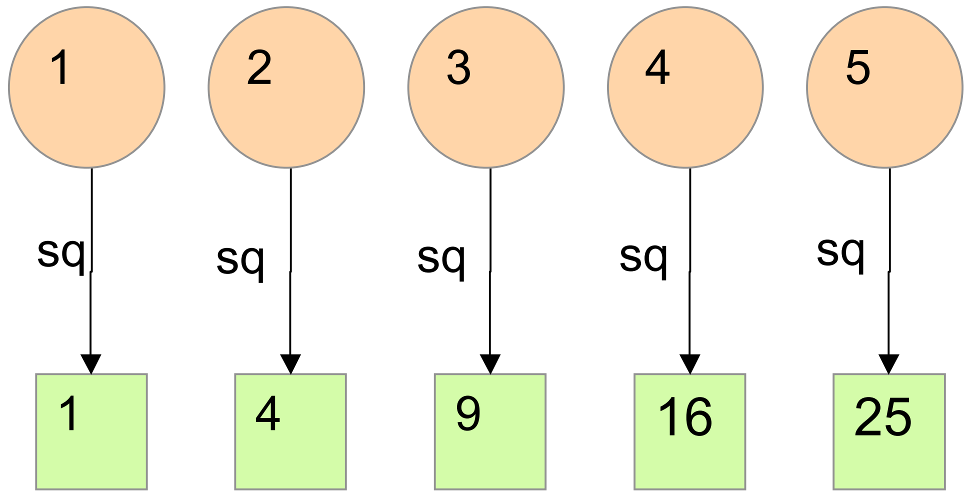
ARIZONA STATE UNIVERSITY
IRA A. FULTON SCHOOL OF ENGINEERING
HIGH PERFORMANCE COMPUTING

LISTS

- We are assuming you know what a list is…
- Just in case:
- {1,2,3,4,5} is a list
- {abc, def,ghi, jkl} is a list
- {{abc,def},{1,2,3}, {ee,ff}} is a list

MAP

- Map is a special function that applies the first argument repeatedly to the second argument and constructs a new list
- Map[f,{2,3,4,5}] ➔ {f[2],f[3],f[4],f[5]}



Input: sq = Function[u,u^2]
Input: Map[sq,{1,2,3,4,5}]
Output: {1,4,9,16,25}

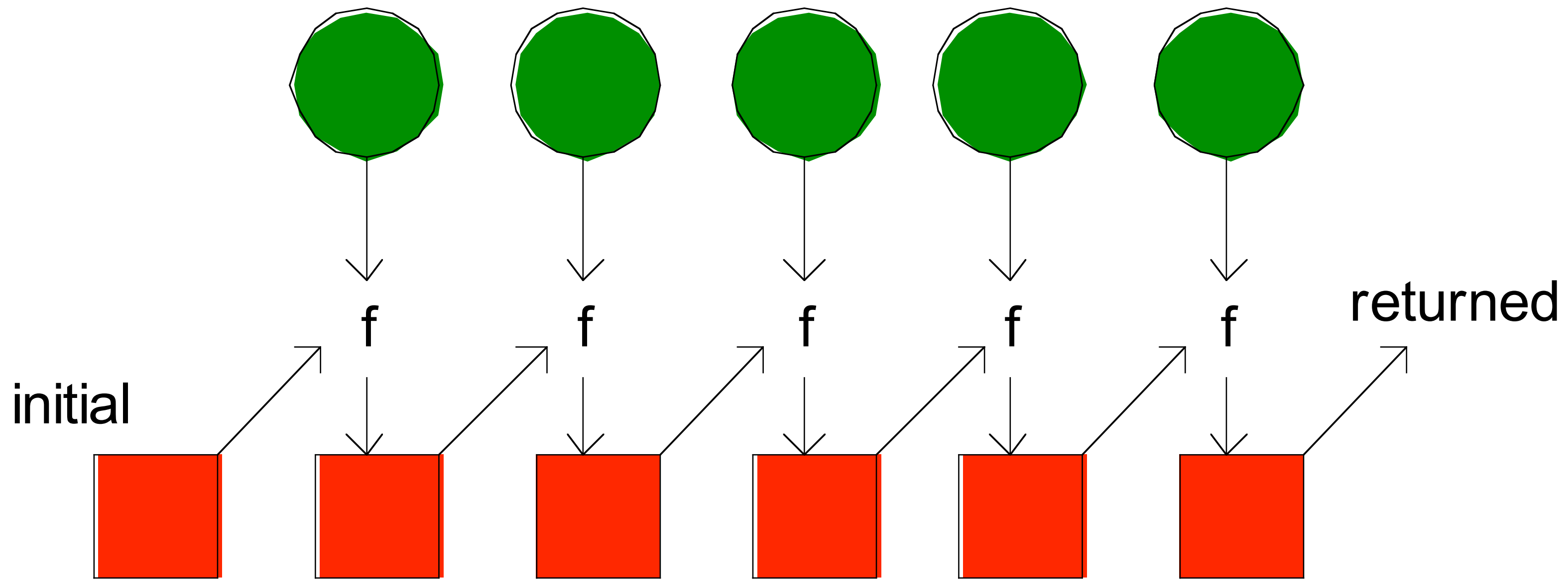
Can this be done in parallel?

REDUCE (FOLD)

- Fold[f,x,list]
- Sets an accumulator
- Initial value is x
- Applies *f* to each element of the list plus the *accumulator*.
- Result is the final value of the accumulator

- **Fold[f,x,{a,b,c}]**
- => f[f[f[x,a],b],c]
- **Remember *fn*?**
- **fn=Function[{u,v},u^2+v^2]**
- **Fold[fn,0,{0,1,2,3}]**
- =>Output ??

FOLD(REDUCE)



MAP/REDUCE

- Map is implicitly parallel
  - No side effects; each list element is operated on separately
- Order of application of function does not matter
- *So map operations can be parallelized*
- The results can be brought together in a fold(reduce) operation

MAP/REDUCE FRAMEWORK

- End Programmer implements interface of two functions:
- **map (key, value)**
  - **Outputs**

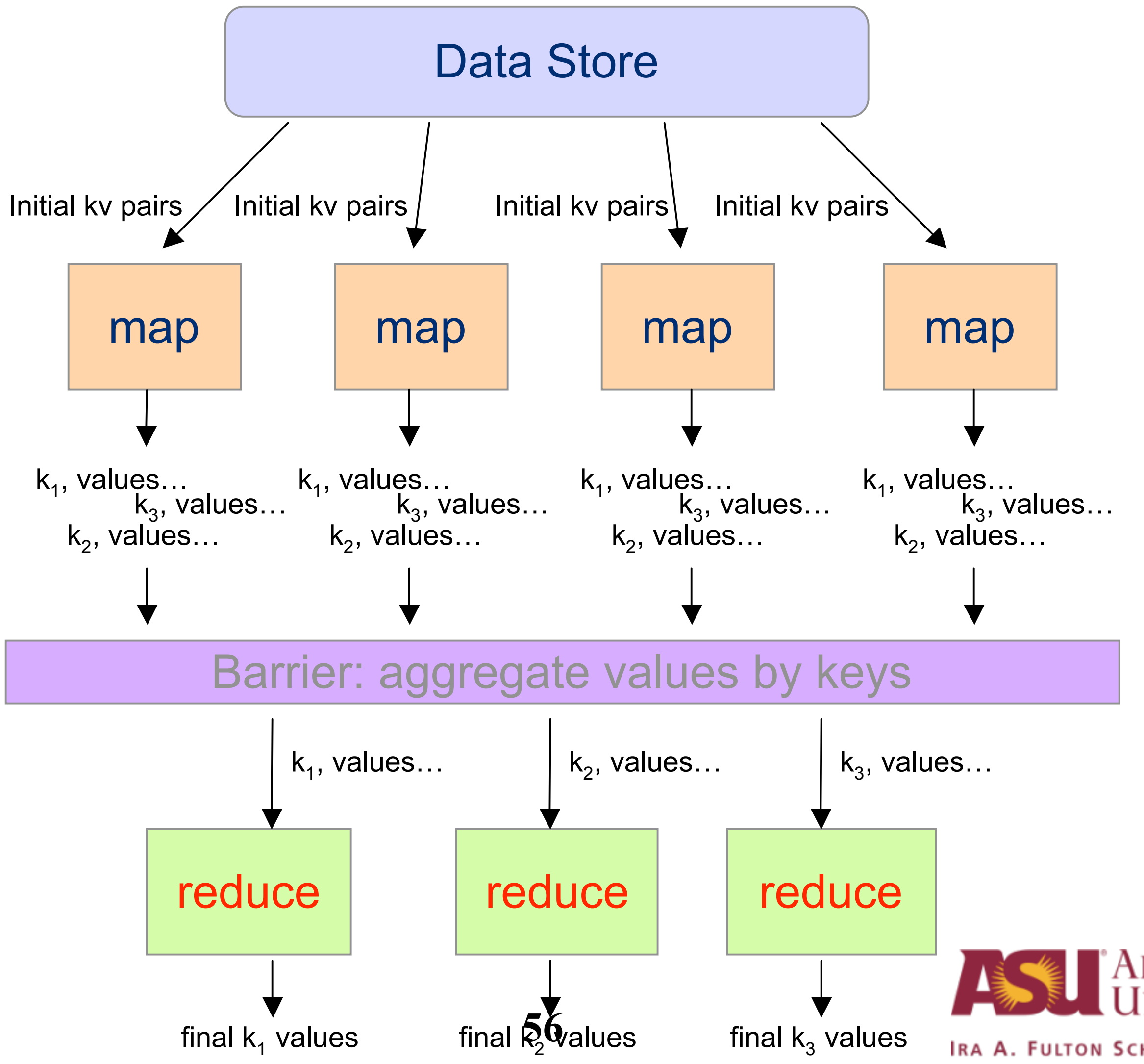
    list of (key'', value'')

- **reduce (key''', value list)**

  List of output value

  All keys with the same *key'''* are sent together in the *value list*
  - reduce phase does not start until map phase is completely finished.

MAP/REDUCE OPERATION



MAP/REDUCE FRAMEWORK

- Programmer does not have to handle
  - work distribution
  - Scheduling
  - Networking
  - Synchronization
  - Fault recovery (if a map or reduce node fails)
  - Moving data between nodes

EXAMPLE: WORD COUNT

- Document content:

My first program in map reduce

Hello map reduce program

Map output*:*

(My 1) (first 1) (program 1) (in 1) (map 1) (reduce 1)

(Hello 1) (map 1) (reduce 1) (program 1)

Reduce output*:*

(My 1) (first 1) (program 2) (in 1) (map 2)

(reduce 2) (Hello 1)

EXAMPLE – WORD COUNT

- You want to count the number of occurrences of each word in a set of documents

```
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));
```

SEEMS WASTEFUL?

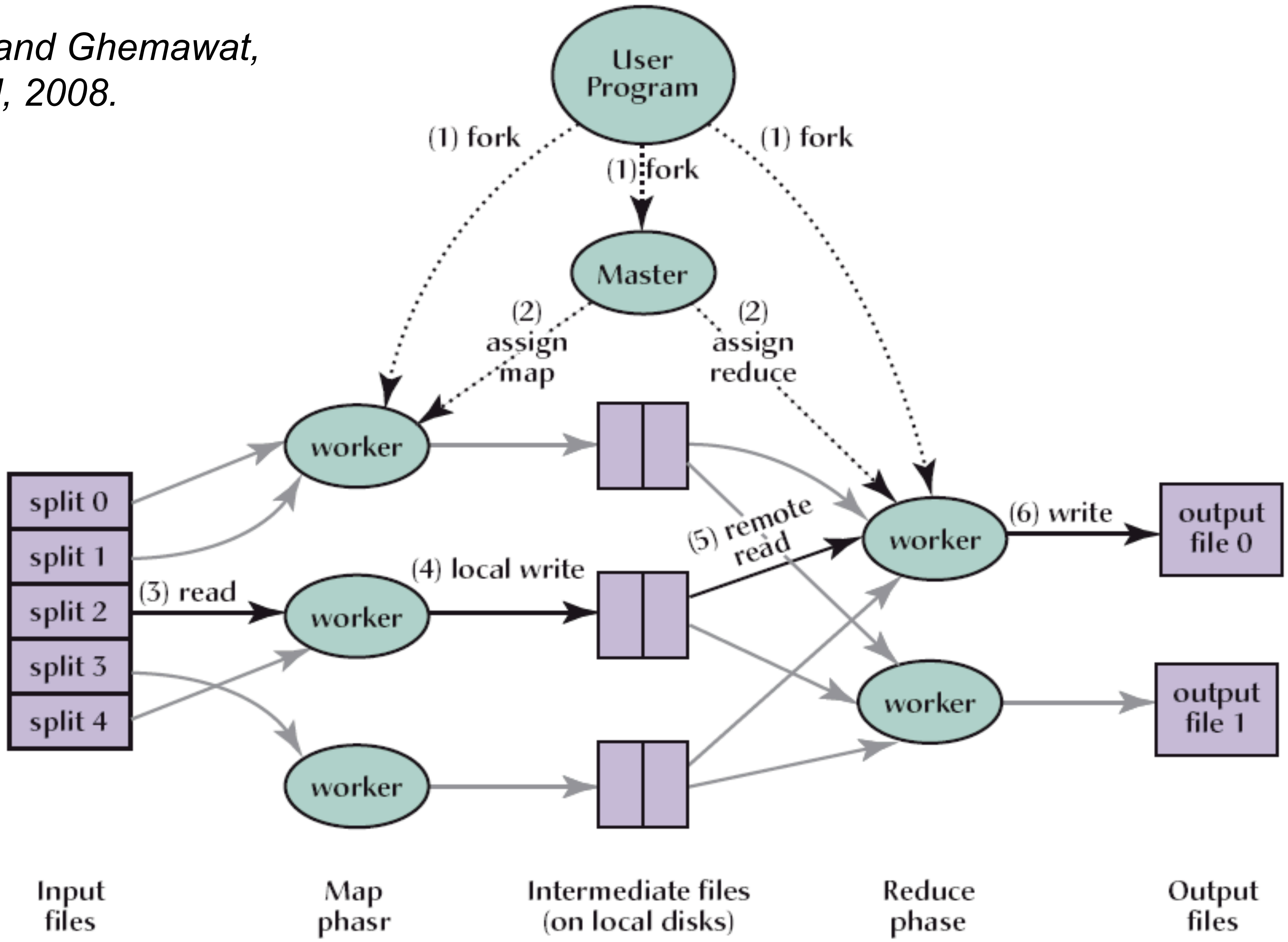
- Large set of intermediate values
- Lot of shuffling of data
- Sorting and resorting of data

MAP/REDUCE OPTIMIZATIONS

- A distributed file system is used to spread the data across nodes
- Map tasks are sent to the nodes where the data resides (or on the same rack)
  - i.e., move the program not the data!
- If a map node fails, it is restarted on another node
- Data is replicated on multiple nodes (usually 3)
- If any map node is unusually slow to complete, it is restarted on another node
  - Uses results from first completed map task

MAP/REDUCE EXECUTION

Dean and Ghemawat, CACM, 2008.



MAP/REDUCE USER INTERFACE

- You implement the Mapper and Reducer interfaces for your application
- Configure your job
  - Set your Mapper's output key, value types
  - Identify your Mapper and Reducer implementations (and the Combiner, if you want to)
  - Set input and output stream types, file paths
- Run the job

MAP IMPLEMENTATION

- Overall, Mapper implementations are passed the JobConf for the job via the JobConfigurable.configure(JobConf) method
  - override it to initialize application specific variables
- The framework then calls map(WritableComparable, Writable, OutputCollector, Reporter) for each line/row/record in the InputSplit for that task.
- Output is written through the OutputCollector.collect(WritableComparable,Writable)

REDUCE IMPLEMENTATION

- As with Map, Reducer implementations are passed the JobConf for the job via the JobConfigurable.configure(JobConf) method
- The framework then calls reduce(WritableComparable, Iterator, OutputCollector, Reporter) method for each <key, (list of values)> pair in the grouped inputs.
- The output of the reduce task is typically written to the FileSystem via OutputCollector.collect(WritableComparable, Writable).

NUMBER OF MAPS AND REDUCES

- JobConf can be used to set number of mappers and reducers
  - You can also set the number of reducers to zero, if needed
- For production applications, one can estimate good numbers based on datasize
  - Or you can leave it to the framework to decide
- You can write a Combiner if you want to reduce data transferred to Reduce stage
  - Executed locally at the map
  - Can often be the Reducer itself (E.g., wordcount)

A Full Hadoop Sample Code

Note: API's are still changing fast; this may or may not work with your version!

Headers (I love Java)

```
package WordCount;

import java.io.IOException;
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;
import java.util.StringTokenizer;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.conf.Configured;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapred.FileInputFormat;
import org.apache.hadoop.mapred.FileOutputFormat;
import org.apache.hadoop.mapred.JobClient;
import org.apache.hadoop.mapred.JobConf;
import org.apache.hadoop.mapred.MapReduceBase;
import org.apache.hadoop.mapred.Mapper;
import org.apache.hadoop.mapred.OutputCollector;
import org.apache.hadoop.mapred.Reducer;
import org.apache.hadoop.mapred.Reporter;
import org.apache.hadoop.util.Tool;
import org.apache.hadoop.util.ToolRunner;
```

More Setup

```
/**
 * This is an example Hadoop Map/Reduce application.
 * It reads the text input files, breaks each line into words
 * and counts them. The output is a locally sorted list of words and the
 * count of how often they occurred.
 *
 * To run: bin/hadoop jar build/hadoop-examples.jar wordcount
 *            [-m <i>maps</i>] [-r <i>reduces</i>] <i>in-dir</i> <i>out-dir</i>
 */
public class WordCount extends Configured implements Tool {

  /**
   * Counts the words in each line.
   * For each line of input, break the line into words and emit them as
   * (<b>word</b>, <b>1</b>).
   */
```

Mapper

```
public static class MapClass extends MapReduceBase
    implements Mapper<LongWritable, Text, Text, IntWritable>
  {

    private final static IntWritable one = new
  IntWritable(1);
    private Text word = new Text();

    public void map(LongWritable key, Text value,
                    OutputCollector<Text, IntWritable>
  output,
                    Reporter reporter) throws IOException {
      String line = value.toString();
      StringTokenizer itr = new StringTokenizer(line);
      while (itr.hasMoreTokens()) {
        word.set(itr.nextToken());
        output.collect(word, one);
      }
    }
  }
```

Reducer

```
/**
  * A reducer class that just emits the sum of the input values.
  */
 public static class Reduce extends MapReduceBase
   implements Reducer<Text, IntWritable, Text, IntWritable> {

   public void reduce(Text key, Iterator<IntWritable> values,
                      OutputCollector<Text, IntWritable> output,
                      Reporter reporter) throws IOException {
     int sum = 0;
     while (values.hasNext()) {
       sum += values.next().get();
     }
     output.collect(key, new IntWritable(sum));
   }
 }
```

Driver (1)

```
/**
 * The main driver for word count map/reduce program.
 * Invoke this method to submit the map/reduce job.
 * @throws IOException When there is communication problems with the
 *                     job tracker.
 */
public int run(String[] args) throws Exception {
  JobConf conf = new JobConf(getConf(), WordCount.class);
  conf.setJobName("wordcount");

  // the keys are words (strings)
  conf.setOutputKeyClass(Text.class);
  // the values are counts (ints)
  conf.setOutputValueClass(IntWritable.class);

  conf.setMapperClass(MapClass.class);
  conf.setCombinerClass(Reduce.class);
  conf.setReducerClass(Reduce.class);
```

Driver (2)

```
List<String> other_args = new ArrayList<String>();
  for(int i=0; i < args.length; ++i) {
    try {
      if ("-m".equals(args[i])) {
        conf.setNumMapTasks(Integer.parseInt(args[++i]));
      } else if ("-r".equals(args[i])) {
        conf.setNumReduceTasks(Integer.parseInt(args[++i]));
      } else {
        other_args.add(args[i]);
      }
    } catch (NumberFormatException except) {
      System.out.println("ERROR: Integer expected instead of " + args[i]);
      return printUsage();
    } catch (ArrayIndexOutOfBoundsException except) {
      System.out.println("ERROR: Required parameter missing from " +
                         args[i-1]);
      return printUsage();
    }
  }
  // Make sure there are exactly 2 parameters left.
  if (other_args.size() != 2) {
    System.out.println("ERROR: Wrong number of parameters: " +
                       other_args.size() + " instead of 2.");
    return printUsage();
  }
  FileInputFormat.setInputPaths(conf, other_args.get(0));
  FileOutputFormat.setOutputPath(conf, new Path(other_args.get(1)));

  JobClient.runJob(conf);
  return 0;
}
```

See how that’s simpler than MPI?

- Neither do I…

Map/Reduce

- No interprocessor communication (other than thru the really slow filesystem).
- Compelling for algorithms involving large scale stream processing of massive datasets
- Unlikely to ever solve a fluid dynamics problem
- A good match for the way programming is taught in modern CS programs (sadly…).
- Could provide a limited form of concurrency “to the masses”.