Cloud Computing and Big Data Processing

Shivaram Venkataraman
UC Berkeley, AMP Lab

Slides from Matei Zaharia
Cloud Computing, Big Data
Hardware
Google 1997
Data, Data, Data

“...Storage space must be used efficiently to store indices and, optionally, the documents themselves. The indexing system must process hundreds of gigabytes of data efficiently...”

The Anatomy of a Large-Scale Hypertextual Web Search Engine

Sergey Brin and Lawrence Page
Commodity CPUs
Lots of disks
Low bandwidth network
Cheap!
Datacenter evolution

Facebook’s daily logs: 60 TB
1000 genomes project: 200 TB
Google web index: 10+ PB

Slide from Ion Stoica
Datacenter Evolution

Google data centers in The Dalles, Oregon
Datacenter Evolution

Capacity:
~10000 machines

Bandwidth:
12-24 disks per node

Latency:
256GB RAM cache
Datacenter Networking

Initially tree topology
Over subscribed links

Fat tree, Bcube, VL2 etc.

Lots of research to get full bisection bandwidth
Datacenter Design

Goals

Power usage effectiveness (PUE)

Cost-efficiency

Custom machine design

Open Compute Project (Facebook)
Datacenters → Cloud Computing

Above the Clouds: A Berkeley View of Cloud Computing

Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia

(Comments should be addressed to abovetheclouds@cs.berkeley.edu)

UC Berkeley Reliable Adaptive Distributed Systems Laboratory *
http://radlab.cs.berkeley.edu/

“...long-held dream of computing as a utility...”
From Mid 2006

Rent virtual computers in the “Cloud”

On-demand machines, spot pricing
## Amazon EC2

<table>
<thead>
<tr>
<th>Machine</th>
<th>Memory (GB)</th>
<th>Compute Units (ECU)</th>
<th>Local Storage (GB)</th>
<th>Cost / hour</th>
</tr>
</thead>
<tbody>
<tr>
<td>t1.micro</td>
<td>0.615</td>
<td>2</td>
<td>0</td>
<td>$0.02</td>
</tr>
<tr>
<td>m1.xlarge</td>
<td>15</td>
<td>8</td>
<td>1680</td>
<td>$0.48</td>
</tr>
<tr>
<td>cc2.8xlarge</td>
<td>60.5</td>
<td>88 (Xeon 2670)</td>
<td>3360</td>
<td>$2.40</td>
</tr>
</tbody>
</table>

1 ECU = CPU capacity of a 1.0-1.2 GHz 2007 Opteron or 2007 Xeon processor
Hardware
## Hopper vs. Datacenter

<table>
<thead>
<tr>
<th></th>
<th>Hopper</th>
<th>Datacenter&lt;sup&gt;2&lt;/sup&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Nodes</strong></td>
<td>6384</td>
<td>1000s to 10000s</td>
</tr>
<tr>
<td><strong>CPUs (per node)</strong></td>
<td>2x12 cores</td>
<td>~2x6 cores</td>
</tr>
<tr>
<td><strong>Memory (per node)</strong></td>
<td>32-64GB</td>
<td>~48-128GB</td>
</tr>
<tr>
<td><strong>Storage (overall)</strong></td>
<td>~4 PB</td>
<td>120-480 PB</td>
</tr>
<tr>
<td><strong>Interconnect</strong></td>
<td>~ 66.4 Gbps</td>
<td>~10Gbps</td>
</tr>
</tbody>
</table>

<sup>2</sup>http://blog.cloudera.com/blog/2013/08/how-to-select-the-right-hardware-for-your-new-hadoop-cluster/
Summary

Focus on Storage vs. FLOPS

Scale out with commodity components

Pay-as-you-go model
Outage in Dublin Knocks Amazon, Microsoft Data Centers Offline

By: Rich Miller  

August 7th, 2011

Dallas-Fort Worth Data Center Update

Filed in on July 9th, 2009

Official Gmail Blog

News, tips and tricks from Google's Gmail team and friends.

Rackspace Community

Some of our customers have reported interruption like this in the past, such incidents from Amazon's perspective:

Amazon EC2 and Amazon RDS Service Disruption.

Gmail's people are addressing the issue and working hard to restore services.
The Joys of Real Hardware

Typical first year for a new cluster:

~0.5 overheating (power down most machines in <5 mins, ~1-2 days to recover)
~1 PDU failure (~500-1000 machines suddenly disappear, ~6 hours to come back)
~1 rack-move (plenty of warning, ~500-1000 machines powered down, ~6 hours)
~1 network rewiring (rolling ~5% of machines down over 2-day span)
~20 rack failures (40-80 machines instantly disappear, 1-6 hours to get back)
~5 racks go wonky (40-80 machines see 50% packetloss)
~8 network maintenance (4 might cause ~30-minute random connectivity losses)
~12 router reloads (takes out DNS and external vips for a couple minutes)
~3 router failures (have to immediately pull traffic for an hour)
~dozens of minor 30-second blips for dns
~1000 individual machine failures
~thousands of hard drive failures
slow disks, bad memory, misconfigured machines, flaky machines, etc.

Long distance links: wild dogs, sharks, dead horses, drunken hunters, etc.

Jeff Dean @ Google
How do we program this?
Programming Models

Message Passing Models (MPI)
Fine-grained messages + computation

Hard to deal with disk locality, failures, stragglers
1 server fails every 3 years →
10K nodes see 10 faults/day
Programming Models

Data Parallel Models
Restrict the programming interface
Automatically handle failures, locality etc.

“Here’s an operation, run it on all of the data”
   – I don’t care \textit{where} it runs (you schedule that)
   – In fact, feel free to run it \textit{retry} on different nodes
MapReduce: Simplified Data Processing on Large Clusters

Jeffrey Dean and Sanjay Ghemawat

jeff@google.com, sanjay@google.com

Google, Inc.

Abstract

MapReduce is a programming model and an associated runtime environment for processing and generating large datasets. Programs specify a map function that processes a key/value pair to generate some intermediate key/value pairs, and a reduce function that merges all intermediate values associated with the same intermediate key. ManyMapReduce is a programming model and an associated runtime environment for processing and generating large datasets. Programs specify a map function that processes a key/value pair to generate some intermediate key/value pairs, and a reduce function that merges all intermediate values associated with the same intermediate key. Many such computations are expressible in this model, as shown given day, etc. Most such computations are expressible in this model, as shown by a variety of applications such as finding PageRank or recomputing an aggregate statistic on a daily basis. However, the input data and the computations have to be distributed across hundreds or thousands of machines in order to complete in a reasonable amount of time. The issues of high data locality and efficient communication conspire to obscure the original situation with large amounts of complex code. MapReduce addresses these issues.

Google 2004

Build search index
Compute PageRank

Hadoop: Open-source
at Yahoo, Facebook
MapReduce Programming Model

Data type: Each record is (key, value)

Map function:
\[(K_{in}, V_{in}) \rightarrow \text{list}(K_{inter}, V_{inter})\]

Reduce function:
\[(K_{inter}, \text{list}(V_{inter})) \rightarrow \text{list}(K_{out}, V_{out})\]
```python
def mapper(line):
    for word in line.split():
        output(word, 1)

def reducer(key, values):
    output(key, sum(values))
```
Word Count Execution

Input

the quick brown fox
the fox ate the mouse
how now brown cow

Map

the, 1 brown, 1 fox, 1
the, 1 fox, 1 the, 1
how, 1 now, 1 brown, 1

Shuffle & Sort

Reduce

Output

brown, 2 fox, 2 how, 1 now, 1 the, 3
ate, 1 cow, 1 mouse, 1 quick, 1
Word Count Execution

Submit a Job
Automatically split work

JobTracker

Schedule tasks with locality

Map
the quick brown fox

Map
the fox ate the mouse

Map
how now brown cow
Fault Recovery

If a task crashes:
– Retry on another node
– If the same task repeatedly fails, end the job

Requires user code to be deterministic
Fault Recovery

If a node crashes:

– Relaunch its current tasks on other nodes
– Relaunch tasks whose outputs were lost

quick brown fox
the fox ate the mouse
how now brown cow
Fault Recovery

If a task is going slowly (straggler):

– Launch second copy of task on another node
– Take the output of whichever finishes first
Applications
1. Search

**Input:** (lineNumber, line) records

**Output:** lines matching a given pattern

**Map:**

```python
if(line matches pattern):
    output(line)
```

**Reduce:** Identity function

– Alternative: no reducer (map-only job)
2. Inverted Index

- hamlet.txt
  - to be or not to be

- 12th.txt
  - be not afraid of greatness

- afraid, (12th.txt)
- be, (12th.txt, hamlet.txt)
- greatness, (12th.txt)
- not, (12th.txt, hamlet.txt)
- of, (12th.txt)
- or, (hamlet.txt)
- to, (hamlet.txt)
2. Inverted Index

Input: (filename, text) records
Output: list of files containing each word

Map:
    foreach word in text.split():
        output(word, filename)

Reduce:
    def reduce(word, filenames):
        output(word, unique(filenames))
2. Inverted Index

- **hamlet.txt**
  - to be or not to be
    - to, hamlet.txt
    - be, hamlet.txt
    - or, hamlet.txt

- **12th.txt**
  - be not afraid of greatness
    - be, 12th.txt
    - not, 12th.txt
    - afraid, 12th.txt

- **afraid, (12th.txt)**
  - be, (12th.txt, hamlet.txt)
  - greatness, (12th.txt)

- **not, (12th.txt, hamlet.txt)**

- **(12th.txt)**
  - of, (12th.txt)
  - or, (hamlet.txt)
  - to, (hamlet.txt)
<table>
<thead>
<tr>
<th>MPI</th>
<th>MapReduce</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parallel process model</td>
<td>High level data-parallel</td>
</tr>
<tr>
<td>Fine grain control</td>
<td>Automate locality, data transfers</td>
</tr>
<tr>
<td>High Performance</td>
<td>Focus on fault tolerance</td>
</tr>
</tbody>
</table>
Summary

MapReduce data-parallel model
Simplified cluster programming

Automates

– Division of job into tasks
– Locality-aware scheduling
– Load balancing
– Recovery from failures & stragglers
When an Abstraction is Useful...

People want to compose it!

Most real applications require multiple MR steps

- Google indexing pipeline: 21 steps
- Analytics queries (e.g. sessions, top K): 2-5 steps
- Iterative algorithms (e.g. PageRank): 10’s of steps
Programmability

Multi-step jobs create spaghetti code
- 21 MR steps → 21 mapper and reducer classes

Lots of boilerplate wrapper code per step
API doesn’t provide type safety
Performance

MR only provides one pass of computation
  – Must write out data to file system in-between

Expensive for apps that need to *reuse* data
  – Multi-step algorithms (e.g. PageRank)
  – Interactive data mining
Spark

Programmability: clean, functional API
- Parallel transformations on collections
- 5-10x less code than MR
- Available in Scala, Java, Python and R

Performance
- In-memory computing primitives
- Optimization across operators
Spark Programmability

Google MapReduce WordCount:

```cpp
#include "mapreduce/mapreduce.h"

// User's map function
class SplitWords: public Mapper {
    public:
        virtual void Map(const MapInput& input) {
            const string& text = input.value();
            const int n = text.size();
            for (int i = 0; i < n; ) {
                // Skip past leading whitespace
                while (i < n && isspace(text[i]))
                    i++;
                // Find word end
                int start = i;
                while (i < n && !isspace(text[i]))
                    i++;
                if (start < i)
                    Emit(text.substr(start, i - start), "1");
            }
        }
    REGISTER_MAPPER(SplitWords);}

    // User's reduce function
class Sum: public Reducer {
    public:
        virtual void Reduce(ReduceInput* input) {
            // Iterate over all entries with the same key and add the values
            int64 value = 0;
            while (!input->done()) {
                value += StringToInt(input->value());
                input->NextValue();
            }
            // Emit sum for input->key()
            Emit(IntToString(value));
        }
    REGISTER_REDUCEER(Sum);

    // Specify the output files
    MapReduceOutput* out = spec.output();
    out->set_base("/gfs/test/freq");
    out->set_num_tasks(100);
    out->set_format("text");
    out->set_reducer_class("Sum");
    out->set_combiner_class("Sum");
    // Do partial sums within map
    out->set_filebase("/gfs/test/freq");
    // Tuning parameters
    spec.set_machines(2000);
    spec.set_map_megabytes(100);
    spec.set_reduce_megabytes(100);
    // Now run it
    MapReduceResult result;
    if (!MapReduce(spec, &result)) abort();
    return 0;
};
```
Spark WordCount:

```scala
val file = spark.textFile("hdfs://...")
val counts = file.flatMap(line => line.split(" "))
  .map(word => (word, 1))
  .reduceByKey(_ + _)

counts.save("out.txt")
```
Spark Performance

Iterative algorithms:

### K-means Clustering
- Hadoop MR: 121 sec
- Spark: 4.1 sec

### Logistic Regression
- Hadoop MR: 80 sec
- Spark: 0.96 sec
Spark Concepts

Resilient distributed datasets (RDDs)
- Immutable, partitioned collections of objects
- May be cached in memory for fast reuse

Operations on RDDs
- *Transformations* (build RDDs)
- *Actions* (compute results)

Restricted shared variables
- Broadcast, accumulators
Example: Log Mining

Load error messages from a log into memory, then interactively search for various patterns

```scala
lines = spark.textFile("hdfs://...")
errors = lines.filter(_.startsWith("ERROR"))
messages = errors.map(_.split('\t')(2))
messages.cache()

messages.filter(_.contains("foo")).count
messages.filter(_.contains("bar")).count

Result: search 1 TB data in 5-7 sec (vs 170 sec for on-disk data)
```
Fault Recovery

RDDs track *lineage* information that can be used to efficiently reconstruct lost partitions.

**Ex:**

```scala
messages = textFile(...).filter(_.startsWith("ERROR"))
  .map(_.split('\t')(2))
```

![Diagram showing the process of fault recovery with RDDs.](image)
Demo
Example: Logistic Regression

Goal: find best line separating two sets of points
Example: Logistic Regression

```scala
val data = spark.textFile(...).map(readPoint).cache()

var w = Vector.random(D)

for (i <- 1 to ITERATIONS) {
  val gradient = data.map(p => (1 / (1 + exp(-p.y*(w dot p.x))) - 1) * p.y * p.x).reduce(_ + _)
  w -= gradient
}

println("Final w: " + w)
```

w automatically shipped to cluster
Logistic Regression Performance

Running Time (min) vs Number of Iterations

- **1st iteration**: 80 s
- **Further iterations**: 1 s
- **Hadoop**: 110 s / iteration
- **Spark**: first iteration 80 s, further iterations 1 s
Shared Variables

RDD operations: use local variables from scope

Two other kinds of shared variables:
   Broadcast Variables
   Accumulators
val data = spark.textFile(...).map(readPoint).cache()

// Random Projection
val M = Matrix.random(N)

var w = Vector.random(D)

for (i <- 1 to ITERATIONS) {
  val gradient = data.map(p =>
      (1 / (1 + exp(-p.y*(w.dot(p.x.dot(M)))))) - 1)
    * p.y * p.x
  ).reduce(_ + _)
  w -= gradient
}

println("Final w: "+ w)
Broadcast Variables

val data = spark.textFile(...).map(readPoint).cache()

// Random Projection
Val M = spark.broadcast(Matrix.random(N))

var w = Vector.random(D)

for (i <- 1 to ITERATIONS) {
  val gradient = data.map(p =>
    (1 / (1 + exp(-p.y*(w.dot(p.x.dot(M.value))))) - 1)
    * p.y * p.x
  ).reduce(_ + _)
  w -= gradient
}

println("Final w: " + w)
### Other RDD Operations

<table>
<thead>
<tr>
<th>Transformations (define a new RDD)</th>
<th>Actions (output a result)</th>
</tr>
</thead>
<tbody>
<tr>
<td>map</td>
<td>collect</td>
</tr>
<tr>
<td>filter</td>
<td>reduce</td>
</tr>
<tr>
<td>sample</td>
<td>take</td>
</tr>
<tr>
<td>groupByKey</td>
<td>fold</td>
</tr>
<tr>
<td>reduceByKey</td>
<td>count</td>
</tr>
<tr>
<td>cogroup</td>
<td>saveAsTextFile</td>
</tr>
<tr>
<td>flatMap</td>
<td>union</td>
</tr>
<tr>
<td>union</td>
<td>join</td>
</tr>
<tr>
<td>join</td>
<td>cross</td>
</tr>
<tr>
<td>count</td>
<td>saveAsHadoopFile</td>
</tr>
<tr>
<td>mapValues</td>
<td>...</td>
</tr>
<tr>
<td>cogroup</td>
<td>...</td>
</tr>
</tbody>
</table>
Java

```java
JavaRDD<String> lines = sc.textFile(...);
lines.filter(new Function<String, Boolean>() {
    Boolean call(String s) {
        return s.contains("error");
    }
}).count();
```

Python

```python
lines = sc.textFile(...)  
lines.filter(lambda x: "error" in x).count()
```

R

```r
lines <- textFile(sc, ...)  
filter(lines, function(x) grepl("error", x))
```
Job Scheduler

Captures RDD dependency graph
Pipelines functions into “stages”
Cache-aware for data reuse & locality
Partitioning-aware to avoid shuffles

Stage 1:
A: B: groupBy

Stage 2:
C: D: E: F: map union

Stage 3:
join

= cached partition
Higher-Level Abstractions

SparkStreaming: API for streaming data
GraphX: Graph processing model
MLLib: Machine learning library
Shark: SQL queries
Hands-on Exercises using Spark, Shark etc.

~250 in person
3000 online

http://ampcamp.berkeley.edu
Course Project Ideas

Linear Algebra on commodity clusters
Optimizing algorithms
Cost model for datacenter topology

Measurement studies
Comparing EC2 vs Hopper
Optimizing BLAS for virtual machines
Conclusion

Commodity clusters needed for big data

Key challenges: Fault tolerance, stragglers

Data-parallel models: MapReduce and Spark
  Simplify programming
  Handle faults automatically