Cloud Computing and Big Data Processing

Shivaram Venkataraman
UC Berkeley, AMP Lab

With slides from Matei Zaharia
Cloud Computing, Big Data
Hardware
Software

Open MPI

UPC

Hadoop

Spark
“…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
Google 2001

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

Data 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 (2014)

<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>1</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
<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>t2.micro</td>
<td>0.615</td>
<td>1</td>
<td>0</td>
<td>$0.013</td>
</tr>
<tr>
<td>r3.xlarge</td>
<td>15</td>
<td>8.13</td>
<td>1680 80(SSD)</td>
<td>$0.35</td>
</tr>
<tr>
<td>r3.8xlarge</td>
<td>60.5</td>
<td>88 104 (Ivy Bridge)</td>
<td>3360 640(SSD)</td>
<td>$2.80</td>
</tr>
</tbody>
</table>

1 ECU = CPU capacity of a 1.0-1.2 GHz 2007 Opteron or 2007 Xeon processor
<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>t2.nano</td>
<td>0.5</td>
<td>1</td>
<td>0</td>
<td>$0.006</td>
</tr>
<tr>
<td>t2.micro</td>
<td>0.615</td>
<td>1</td>
<td>0</td>
<td>$0.013</td>
</tr>
<tr>
<td>r3.8xlarge</td>
<td>60.5   244</td>
<td>88 104 (Ivy Bridge)</td>
<td>3360 640(SSD)</td>
<td>$2.80</td>
</tr>
<tr>
<td>x1 (TBA)</td>
<td>2 TB</td>
<td>4 * Xeon E7</td>
<td>?</td>
<td>?</td>
</tr>
</tbody>
</table>
Hardware
# Hopper vs. Datacenter

<table>
<thead>
<tr>
<th></th>
<th>Hopper</th>
<th>Datacenter²</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>

²[http://blog.cloudera.com/blog/2013/08/how-to-select-the-right-hardware-for-your-new-hadoop-cluster/](http://blog.cloudera.com/blog/2013/08/how-to-select-the-right-hardware-for-your-new-hadoop-cluster/)
## Hopper Cori Phase 1 vs. Datacenter

<table>
<thead>
<tr>
<th></th>
<th>Hopper Cori Phase 1</th>
<th>Datacenter&lt;sup&gt;2&lt;/sup&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nodes</td>
<td>6384-1630</td>
<td>1000s to 10000s</td>
</tr>
<tr>
<td>CPUs (per node)</td>
<td>2x12 2x16 cores</td>
<td>~2x6 cores</td>
</tr>
<tr>
<td>Memory (per node)</td>
<td>32-64GB 128 GB</td>
<td>~48-128GB</td>
</tr>
<tr>
<td>Storage (overall)</td>
<td>~4 PB ~30PB</td>
<td>120-480 PB</td>
</tr>
<tr>
<td>Interconnect</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/
Cori: Intel Xeon Phi

Many Core Integrated Architecture

> 60 cores per node

Vector Processing
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

A lightning strike for Amazon

many sites for Amazon

Message from Rackspace

July 9, 2009

Official Gmail Blog

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

Amazon EC2 and Amazon RDS Service Disruption

Posted

Gmail's people reported a problem

and we're taking action to all affected services, we would like to share more details with our customers about the events that caused such incidents from

Posted
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 maintenances (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

Exascale research: Fault Tolerant MPI (FTMPI)
Checkpointing-based techniques
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 *where* it runs (you schedule that)
   – In fact, feel free to run it *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 system for processing and generating large data sets. It was originally designed at Google to process the huge data sets generated daily by the company's search and advertising businesses. The programming model and system are now used by a wide variety of high-volume data processing applications. The key components and abstractions of the system are the MapReduce programming model and the Hadoop software framework, which provides a distributed file system and a job scheduling system for running MapReduce jobs. This article describes the system and its applications.

Hadoop is an open-source implementation of the MapReduce framework written at Yahoo! Inc. and later contributed to the open source community. It provides a software platform for running distributed applications.

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:

$$(K_{inter}, \text{list}(V_{inter})) \rightarrow \text{list}(K_{out}, V_{out})$$
Example: Word Count

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
- how, 1
- now, 1
- brown, 1
- ate, 1
- mouse, 1
- cow, 1

Shuffle & Sort
- the, 1
- brown, 1
- fox, 1
- the, 1
- quick, 1
- how, 1
- now, 1
- brown, 1
- ate, 1
- mouse, 1
- cow, 1

Reduce
- brown, 2
- fox, 2
- how, 1
- now, 1
- the, 3
- ate, 1
- cow, 1
- mouse, 1
- quick, 1

Output
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
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

What about task inputs? File system replication

*Map*
- 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

the quick brown fox
the quick brown fox
the fox ate the mouse
how now brown cow
<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>- Focus on High Performance</td>
<td>- Focus on fault tolerance</td>
</tr>
</tbody>
</table>
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:

```c++
#include "mapreduce/mapreduce.h"

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);
}

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_REDUCER(Sum);
}

int main(int argc, char** argv) {
    ParseCommandLineFlags(argc, argv);
    MapReduceSpecification spec;
    for (int i = 1; i < argc; i++) {
        MapReduceInput* in = spec.add_input();
        in->set_format("text");
        in->set_filepattern(argv[i]);
        in->set_mapper_class("SplitWords");
    }
    // Specify the output files
    MapReduceOutput* out = spec.output();
    out->set_filebase("/gfs/test/freq");
    out->set_num_tasks(100);
    out->set_format("text");
    out->set_reducer_class("Sum");
    // Do partial sums within map
    out->set_combiner_class("Sum");
    // 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

Find error messages present in log files interactively
(Example: HTTP server logs)

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

Find error messages present in log files interactively
(Example: HTTP server logs)

```python
lines = spark.textFile("hdfs://...")
errors = lines.filter(_.startsWith("ERROR"))
messages = errors.map(_.split("\t")(2))
messages.cache()

messages.filter(_.contains("foo")).count
```
Example: Log Mining

Find error messages present in log files interactively
(Example: HTTP server logs)

```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: full-text search of Wikipedia in <1 sec (vs 20 sec for on-disk data)
Example: Log Mining

Find error messages present in log files interactively
(Example: HTTP server logs)

```scala
lines = spark.textFile("hdfs://...")
errors = lines.filter(_.startsWith("ERROR"))
messages = errors.map(_.split("\t")(2))
messages.cache()

result = messages.filter(_.contains("foo")).count
result = 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:  
\[
\text{messages = textFile(...).filter(_.startsWith("ERROR")\).map(_.split('\t')(2))}
\]
Demo: Digit Classification
MNIST
Minimize $\| Ax - b \|_2$

$$x = (A^T A)^{-1} A^T b$$
Minimize $\|Ax - b\|_2$

$$x = (A^T A)^{-1} A^T b$$

Use QR Decomposition!
# Other RDD Operations

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

Java

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

Python

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

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

= cached partition
Higher-Level Abstractions

SparkStreaming: API for streaming data
GraphX: Graph processing model
MLLib: Machine learning library
SparkSQL: SQL queries, DataFrames

...
Scientific Computing on Spark

Large Scale Machine Learning (AMPLab, Aspire)
  - KeystoneML: Framework for ML
  - Improved Algorithms

Spark on HPC systems (NERSC, Cray, AMPLab)
  - Real world applications
  - Profiling, hardware customizations
Scientific Computing on Spark

Computation
- Efficient native operations using JNI
- Use BLAS / OpenMP per-node

Communication
- Limited programming model
- Shuffle could add latency, bandwidth
Hands-on Exercises using Spark, SparkSQL, MLLib

~250 in person
~2000 online

http://ampcamp.berkeley.edu/6
Spark Adoption

Open source Apache Project, > 400 contributors

Packaged by Cloudera, Hortonworks

Databricks: Spark as a cloud service

Unified Platform for Big Data Applications
Course Project Ideas

Linear Algebra on Spark
  Solvers e.g., Conjugate gradient, LSQR
  Sparse Matrix Algorithms

Measurement studies
  Spark + Xeon Phi / GPU - Benefits / Challenges ?
  Applications (NERSC, Machine Learning)
Conclusion

Commodity clusters needed for big data

Key challenges: Fault tolerance, stragglers

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