Spark: A framework for iterative and interactive cluster computing

Matei Zaharia
(presented by Anthony D. Joseph)

LASER Summer School
September 2013
My Talks at LASER 2013

1. AMP Lab introduction
2. The Datacenter Needs an Operating System
3. Mesos, part one
4. Dominant Resource Fairness
5. Mesos, part two
6. Spark
Researchers

• Matei Zaharia
• Mosharaf Chowdhury
• Michael Franklin
• Scott Shenker
• Ion Stoica

http://spark.incubator.apache.org/

Berkeley Data Analytics Stack

- Shark
- BlinkDB
- SQL
- Spark
- Streaming
- GraphX
- MLBase

Apache Spark

HDFS / Hadoop Storage / Tachyon

Apache Mesos / YARN Resource Manager
Motivation

Complex jobs, Machine Learning algorithms, interactive queries and online processing all need one thing that Hadoop MR lacks:

Efficient primitives for data sharing

Iterative job

Interactive mining

Stream processing
Transfer and Sharing in Hadoop
In-Memory Data Sharing

Input

one-time processing

Distributed memory

iter. 1 → iter. 2 → ...

query 1 → query 2 → query 3 → ...

Input
Spark Goals

Support iterative and stream jobs

Experiment with programmability
  » Leverage Scala to integrate cleanly into programs
  » Support interactive use from Scala interpreter

Retain MapReduce’s fine-grained fault-tolerance
Programming Model

Driver program
  » Implements high-level control flow of an application
  » Launches various operations in parallel

Distributed datasets
  » HDFS files, “parallelized” Scala collections
  » Can be transformed with map and filter
  » Can be cached across parallel operations

Parallel operations
  » For each, reduce, collect

Shared variables
  » Accumulators (add-only), Broadcast variables (read-only)
Distributed Datasets

Represented by a Scala object and constructed:

› From a file in a shared filesystem (e.g., HDFS)

› By “parallelizing” a Scala collection (e.g., an array) in the driver program – dividing it into a number of slices

› By “transforming” an existing Distributed Dataset (e.g., using a user-provided function A \( \Rightarrow \) List[B], or flatMap)
Parallel Operations

*reduce* – Combines dataset elements using an associative function to produce a result at the driver program

*collect* – Sends all elements of the dataset to the driver program (e.g., update an array in parallel with *parallelize*, *map*, and *collect*)

*foreach* – Passes each element through a user provided function

No grouped *reduce* operation
Shared Variables

Broadcast variables
» Used for large read-only data (e.g., lookup table) in multiple parallel operations – distributed once instead of packaging with every closure

Accumulators
» Variables that works can only “add” to using an associative operation, and only the driver program can read
Spark Version of Word Count

```scala
file = spark.textFile("hdfs://...")

file.flatMap(line => line.split(" "))
  .map(word => (word, 1))
  .reduceByKey(_ + _)
```
Example 1: Logistic Regression
Logistic Regression

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

Serial Version

```scala
val data = readData(...)

var w = Vector.random(D)

for (i <- 1 to ITERATIONS) {
  var gradient = Vector.zeros(D)
  for (p <- data) {
    val s = (1/(1+exp(-p.y*(w dot p.x)))-1) * p.y
    gradient += s * p.x
  }
  w -= gradient
}
println("Final w: " + w)
```

Spark Version

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

var w = Vector.random(D)

for (i <- 1 to ITERATIONS) {
  var gradient = spark.accumulator(
    Vector.zeros(D))
  for (p <- data) {
    val s = (1/(1+exp(-p.y*(w dot p.x)))-1) * p.y
    gradient += s * p.x
  }
  w -= gradient.value
}
println("Final w: " + w)
```
val data = readData(...)

var w = Vector.random(D)

for (i <- 1 to ITERATIONS) {
  var gradient = Vector.zeros(D)
  for (p <- data) {
    val scale = (1/(1+exp(-p.y*(w dot p.x))) - 1) * p.y
    gradient += scale * p.x
  }
  w -= gradient
}

println("Final w: ", w)
Spark Version

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

var w = Vector.random(D)

for (i <- 1 to ITERATIONS) {
    var gradient = spark.accumulator(Vector.zeros(D))
    for (p <- data) {
        val scale = (1/(1+exp(-p.y*(w dot p.x))) - 1) * p.y
        gradient += scale * p.x
    }
    w -= gradient.value
}

println("Final w: " + w)
```
Logistic Regression Implementations

Serial Version

```scala
val data = readData(...) 

var w = Vector.random(D)

for (i <- 1 to ITERATIONS) {
  var gradient = Vector.zeros(D)
  for (p <- data) {
    val s = (1/(1+exp(-p.y*(w dot p.x)))-1) * p.y 
    gradient += s * p.x
  }
  w -= gradient
}

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

Spark Version

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

var w = Vector.random(D)

for (i <- 1 to ITERATIONS) {
  var gradient = spark.accumulator( 
    Vector.zeros(D))
  for (p <- data) {
    val s = (1/(1+exp(-p.y*(w dot p.x)))-1) * p.y 
    gradient += s * p.x
  }
  w -= gradient.value
}

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

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

var w = Vector.random(D)

for (i <- 1 to ITERATIONS) {
  var gradient = spark.accumulator(Vector.zeros(D))
  for (p <- data) {
    val scale = (1/(1+exp(-p.y*(w dot p.x))) - 1) * p.y
    gradient += scale * p.x
  }
  w -= gradient.value
}

println("Final w: " + w)
Logistic Regression Implementations

Serial Version

```scala
val data = readData(...) 

var w = Vector.random(D)

for (i <- 1 to ITERATIONS) {
  var gradient = Vector.zeros(D)
  for (p <- data) {
    val s = (1/(1+exp(-p.y* (w dot p.x)))-1) * p.y
    gradient += s * p.x
  }
  w -= gradient
}
println("Final w: " + w)
```

Spark Version

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

var w = Vector.random(D)

for (i <- 1 to ITERATIONS) {
  var gradient = spark.accumulator( Vector.zeros(D))
  data.foreach(p => {
    val s = (1/(1+exp(-p.y* (w dot p.x)))-1) * p.y
    gradient += s * p.x
  })
  w -= gradient.value
}
println("Final w: " + w)
```
val data = spark.hdfsTextFile(...).map(readPoint).cache()

var w = Vector.random(D)

for (i <- 1 to ITERATIONS) {
    var gradient = spark.accumulator(Vector.zeros(D))
    data.foreach(p => {
        val scale = (1/(1+exp(-p.y*(w dot p.x))) - 1) * p.y
        gradient += scale * p.x
    })
    w -= gradient.value
}

println("Final w: " + w)
Functional Programming Version

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

var w = Vector.random(D)

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

}

println("Final w: " + w)
Job Execution

Spark

update
param

aggregate

Master

Slave 1
Slave 2
Slave 3
Slave 4

R1
R2
R3
R4

param
Job Execution

Spark

Hadoop / Dryad
Performance

Running Time (s)

<table>
<thead>
<tr>
<th>Number of Iterations</th>
<th>Hadoop</th>
<th>Spark</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>127 s</td>
<td>26 s</td>
</tr>
<tr>
<td>5</td>
<td>127 s</td>
<td>26 s</td>
</tr>
<tr>
<td>10</td>
<td>127 s</td>
<td>26 s</td>
</tr>
<tr>
<td>20</td>
<td>127 s</td>
<td>26 s</td>
</tr>
<tr>
<td>30</td>
<td>127 s</td>
<td>26 s</td>
</tr>
</tbody>
</table>

First iteration: 174 s
Further iterations: 6 s
Example 2:
Alternating Least Squares
Collaborative Filtering

Predict movie ratings for a set of users based on their past ratings

<table>
<thead>
<tr>
<th></th>
<th>Movies</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Users</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Matrix Factorization

Model $R$ as product of user and movie matrices $A$ and $B$ of dimensions $U \times K$ and $M \times K$

$$R = AB^T$$

Problem: given subset of $R$, optimize $A$ and $B$
Alternating Least Squares Algorithm

Start with random A and B

Repeat:

1. Fixing B, optimize A to minimize error on scores in R
2. Fixing A, optimize B to minimize error on scores in R
Serial ALS

val R = readRatingsMatrix(...)

var A = (0 until U).map(i => Vector.random(K))
var B = (0 until M).map(i => Vector.random(K))

for (i <- 1 to ITERATIONS) {
  A = (0 until U).map(i => updateUser(i, B, R))
  B = (0 until M).map(i => updateMovie(i, A, R))
}
Naïve Spark ALS

val R = readRatingsMatrix(...)

var A = (0 until U).map(i => Vector.random(K))
var B = (0 until M).map(i => Vector.random(K))

for (i <- 1 to ITERATIONS) {
  A = spark.parallelize(0 until U, numSlices)
      .map(i => updateUser(i, B, R))
      .collect()
  B = spark.parallelize(0 until M, numSlices)
      .map(i => updateMovie(i, A, R))
      .collect()
}

Problem:
R re-sent to all nodes in each parallel operation
val R = spark.broadcast(readRatingsMatrix(...))

var A = (0 until U).map(i => Vector.random(K))
var B = (0 until M).map(i => Vector.random(K))

for (i <- 1 to ITERATIONS) {
  A = spark.parallelize(0 until U, numSlices)
    .map(i => updateUser(i, B, R.value))
    .collect()
  B = spark.parallelize(0 until M, numSlices)
    .map(i => updateMovie(i, A, R.value))
    .collect()
}
ALS Performance

Cluster Configuration

- 4 cores (1 node)
- 12 cores (2 nodes)
- 20 cores (3 nodes)
- 28 cores (4 nodes)
- 36 cores (5 nodes)
- 60 cores (8 nodes)

First Iteration
Subsequent Iterations

Iteration Duration (s)
Subsequent Iteration Breakdown

36% of iteration spent on broadcast
Architecture

Driver program connects to Mesos and schedules tasks

Workers run tasks, report results and variable updates

Data shared with HDFS/NFS

No communication between workers for now
Challenge

How to design a distributed memory abstraction that is both fault-tolerant and efficient?

Traditional in-memory storage systems replicate data or update logs across nodes ➔ slow!
  ➔ Network write is 10-100× slower than memory
Resilient Distributed Datasets

Each distributed dataset object maintains a lineage that is used to rebuild slices that are lost / fall out of cache

Ex: errors = textFile("log").filter(_.contains("error")).map(_.split(\'\t\')(1)).cache()
Resilient Distributed Datasets

RDDs provide an interface for coarse-grained transformations (map, group-by, join, …)

Efficient fault recovery using lineage
  » Log one operation to apply to many elements
  » Recompute lost partitions of RDD on failure
  » No cost if nothing fails

Rich enough to capture many models:
  » Data flow models: MapReduce, Dryad, SQL, …
  » Specialized models for iterative apps: Pregel, Hama, …

Interactive Spark

Modified Scala interpreter to allow Spark to be used interactively from the command line

Required two changes:

» Modified wrapper code generation so that each “line” typed has references to objects for its dependencies
» Place generated classes in distributed filesystem

Enables in-memory exploration of big data
Milestones

2010: Spark open sourced
Feb 2013: Spark Streaming alpha open sourced
Jun 2013: Spark entered Apache Incubator
Aug 2013: Machine Learning library for Spark
Frameworks Built on Spark

MapReduce

HaLoop
  » Iterative MapReduce from UC Irvine / U Washington

Pregel on Spark (Bagel)
  » Graph processing framework from Google based on BSP message-passing model

Hive on Spark (Shark)
  » In progress
Calling in your passion for data, let's meet!

July 26 - 5:00 PM
Pramati Technologies

The first Spark User Group - Hyderabad, meetup invites everyone passionate about data... Let's meet, discuss and showcase our work around data mining, analytics and engineering.

Sachin Anto

41 attended
Related Work

Daytona
» Supports iteration and ML workloads
» Azure-specific integration

DryadLINQ
» SQL-like queries integrated in C# programs
» Build queries through operations on lazy datasets
» Cannot have a dataset persist across queries
» No concept of shared variables for broadcast etc

Pig & Hive
» Query languages that can call into Java/Python/etc UDFs
» No support for caching a dataset across queries

OpenMP
» Compiler extension for parallel loops in C++
» Annotate variables as read-only or accumulator above loop
» Cluster version exists, but not fault-tolerant
Conclusions

Spark provides two abstractions that enable iterative jobs and interactive use:

1. **Distributed datasets** with controllable persistence, supporting fault-tolerant parallel operations

2. **Shared variables** for efficient broadcast and imperative style programming

Language integration achieved using Scala features + some amount of hacking

All this is surprisingly little code (~1600 lines)

http://spark.incubator.apache.org/
Apache Mesos and Spark

Conceived of by graduate students
Implemented locally first
Tested and scaled up using Amazon EC2
Released and open sourced to community

You can do the same!
Berkeley Data Analytics Stack

http://amplab.cs.berkeley.edu/
Language Integration

Scala closures are Serializable objects
  » Serialize on driver, load & run on workers

Not quite enough
  » Nested closures may reference entire outer scope
  » May pull in non-Serializable variables not used inside
  » Solution: bytecode analysis + reflection

Shared variables
  » Accumulators: serialized form contains ID
  » Broadcast vars: serialized form is path to HDFS file