Parallel Programming With Spark

Matei Zaharia

UC Berkeley

www.spark-project.org
What is Spark?

Fast and expressive cluster computing system compatible with Apache Hadoop
» Works with any Hadoop-supported storage system and data format (HDFS, S3, SequenceFile, …)

Improves efficiency through:
» In-memory computing primitives
» General computation graphs

Improves usability through rich Scala and Java APIs and interactive shell

As much as 30× faster

Often 2-10× less code
How to Run It

Local multicore: just a library in your program
EC2: scripts for launching a Spark cluster
Private cluster: Mesos, YARN*, standalone*

*Coming soon in Spark 0.6
Scala vs Java APIs

Spark originally written in Scala, which allows concise function syntax and interactive use

Recently added Java API for standalone apps (dev branch on GitHub)

Interactive shell still in Scala

This course: mostly Scala, with translations to Java
Outline

Introduction to Scala & functional programming

Spark concepts

Tour of Spark operations

Job execution
About Scala

High-level language for the Java VM
  » Object-oriented + functional programming

Statically typed
  » Comparable in speed to Java
  » But often no need to write types due to type inference

Interoperates with Java
  » Can use any Java class, inherit from it, etc; can also call Scala code from Java
Best Way to Learn Scala

Interactive shell: just type scala

Supports importing libraries, tab completion, and all constructs in the language
Quick Tour

Declaring variables:

```scala
var x: Int = 7
var x = 7  // type inferred
val y = "hi"  // read-only
```

Java equivalent:

```java
int x = 7;
final String y = "hi";
```

Functions:

```scala
def square(x: Int): Int = x*x
def square(x: Int): Int = {
  x*x
}
def announce(text: String) {
  println(text)
}
```

Java equivalent:

```java
int square(int x) {
  return x*x;
}
void announce(String text) {
  System.out.println(text);
}
```

Last expression in block returned
Quick Tour

Generic types:

```java
var arr = new Array[Int](8)
var lst = List(1, 2, 3)
// type of lst is List[Int]
```

Java equivalent:

```java
int[] arr = new int[8];
List<Integer> lst = new ArrayList<Integer>();
lst.add(...)
```

Indexing:

```java
arr(5) = 7
println(lst(5))
```

Java equivalent:

```java
arr[5] = 7;
System.out.println(lst.get(5));
```

Factory method

Can’t hold primitive types
Quick Tour

Processing collections with functional programming:

```scala
val list = List(1, 2, 3)

list.foreach(x => println(x))  // prints 1, 2, 3
list.foreach(println)          // same

list.map(x => x + 2)           // => List(3, 4, 5)
list.map(_ + 2)                // same, with placeholder notation

list.filter(x => x % 2 == 1)   // => List(1, 3)
list.filter(_ % 2 == 1)        // => List(1, 3)

list.reduce((x, y) => x + y)   // => 6
list.reduce(_ + _)             // => 6
```

All of these leave the list unchanged (List is immutable)
Scala Closure Syntax

(x: Int) => x + 2    // full version

x => x + 2         // type inferred

_ + 2              // when each argument is used exactly once

x => {             // when body is a block of code
  val numberToAdd = 2
  x + numberToAdd
}

// If closure is too long, can always pass a function

def addTwo(x: Int): Int = x + 2

list.map(addTwo)

Scala allows defining a “local function” inside another function
**Other Collection Methods**

Scala collections provide many other functional methods; for example, Google for “Scala Seq”

<table>
<thead>
<tr>
<th>Method on Seq[T]</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>map(f: T =&gt; U): Seq[U]</td>
<td>Pass each element through f</td>
</tr>
<tr>
<td>flatMap(f: T =&gt; Seq[U]): Seq[U]</td>
<td>One-to-many map</td>
</tr>
<tr>
<td>filter(f: T =&gt; Boolean): Seq[T]</td>
<td>Keep elements passing f</td>
</tr>
<tr>
<td>exists(f: T =&gt; Boolean): Boolean</td>
<td>True if one element passes</td>
</tr>
<tr>
<td>forall(f: T =&gt; Boolean): Boolean</td>
<td>True if all elements pass</td>
</tr>
<tr>
<td>reduce(f: (T, T) =&gt; T): T</td>
<td>Merge elements using f</td>
</tr>
<tr>
<td>groupBy(f: T =&gt; K): Map[K,List[T]]</td>
<td>Group elements by f(element)</td>
</tr>
<tr>
<td>sortBy(f: T =&gt; K): Seq[T]</td>
<td>Sort elements by f(element)</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

...
Outline

Introduction to Scala & functional programming

Spark concepts

Tour of Spark operations

Job execution
Spark Overview

Goal: work with distributed collections as you would with local ones

Concept: resilient distributed datasets (RDDs)
  » Immutable collections of objects spread across a cluster
  » Built through parallel transformations (map, filter, etc)
  » Automatically rebuilt on failure
  » Controllable persistence (e.g. caching in RAM) for reuse
Main Primitives

Resilient distributed datasets (RDDs)
  » Immutable, partitioned collections of objects

Transformations (e.g. map, filter, groupBy, join)
  » Lazy operations to build RDDs from other RDDs

Actions (e.g. count, collect, save)
  » Return a result or write it to storage
Example: Log Mining

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

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

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

**Result:** scaled to 1 TB data in 5-7 sec (vs 170 sec for on-disk data)
RDD Fault Tolerance

RDDs track the series of transformations used to build them (their *lineage*) to recompute lost data.

E.g.: messages = textFile(...).filter(_.contains("error")).map(_.split('\t')(2))
Fault Recovery Test

- Iteration time (s)

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Iteration time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>119</td>
</tr>
<tr>
<td>2</td>
<td>57</td>
</tr>
<tr>
<td>3</td>
<td>56</td>
</tr>
<tr>
<td>4</td>
<td>58</td>
</tr>
<tr>
<td>5</td>
<td>58</td>
</tr>
<tr>
<td>6</td>
<td>81</td>
</tr>
<tr>
<td>7</td>
<td>57</td>
</tr>
<tr>
<td>8</td>
<td>59</td>
</tr>
<tr>
<td>9</td>
<td>57</td>
</tr>
<tr>
<td>10</td>
<td>59</td>
</tr>
</tbody>
</table>

Failure happens at Iteration 6.
Behavior with Less RAM

<table>
<thead>
<tr>
<th>% of working set in cache</th>
<th>Iteration time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cache disabled</td>
<td>69</td>
</tr>
<tr>
<td>25%</td>
<td>58</td>
</tr>
<tr>
<td>50%</td>
<td>41</td>
</tr>
<tr>
<td>75%</td>
<td>30</td>
</tr>
<tr>
<td>Fully cached</td>
<td>12</td>
</tr>
</tbody>
</table>
How it Looks in Java

JavaRDD<String> lines = ...;

lines.filter(_.contains("error")).count();

More examples in the next talk
Outline

Introduction to Scala & functional programming

Spark concepts

Tour of Spark operations

Job execution
Learning Spark

Easiest way: Spark interpreter (spark-shell)
  » Modified version of Scala interpreter for cluster use

Runs in local mode on 1 thread by default, but can control through MASTER environment var:

MASTER=local     ./spark-shell # local, 1 thread
MASTER=local[2]  ./spark-shell # local, 2 threads
MASTER=host:port ./spark-shell # run on Mesos
First Stop: SparkContext

Main entry point to Spark functionality

Created for you in spark-shell as variable sc

In standalone programs, you’d make your own
(see later for details)
Creating RDDs

// Turn a Scala collection into an RDD
sc.parallelize(List(1, 2, 3))

// Load text file from local FS, HDFS, or S3
sc.textFile("file.txt")
sc.textFile("directory/*.txt")
sc.textFile("hdfs://namenode:9000/path/file")

// Use any existing Hadoop InputFormat
sc.hadoopFile(keyClass, valClass, inputFmt, conf)
Basic Transformations

```scala
val nums = sc.parallelize(List(1, 2, 3))

// Pass each element through a function
val squares = nums.map(x => x*x)  // {1, 4, 9}

// Keep elements passing a predicate
val even = squares.filter(_ % 2 == 0)  // {4}

// Map each element to zero or more others
nums.flatMap(x => 1 to x)  // => {1, 1, 2, 1, 2, 3}
```

Range object (sequence of numbers 1, 2, ..., x)
Basic Actions

val nums = sc.parallelize(List(1, 2, 3))

// Retrieve RDD contents as a local collection
nums.collect()  // => Array(1, 2, 3)

// Return first K elements
nums.take(2)    // => Array(1, 2)

// Count number of elements
nums.count()    // => 3

// Merge elements with an associative function
nums.reduce(_ + _)  // => 6

// Write elements to a text file
nums.saveAsTextFile("hdfs://file.txt")
Working with Key-Value Pairs

Spark’s “distributed reduce” transformations operate on RDDs of key-value pairs

Scala pair syntax:

```scala
val pair = (a, b)  // sugar for new Tuple2(a, b)
```

Accessing pair elements:

```scala
pair._1  // => a
pair._2  // => b
```
Some Key-Value Operations

```scala
val pets = sc.parallelize(List(("cat", 1), ("dog", 1), ("cat", 2)))
pets.reduceByKey(_ + _) // => {(cat, 3), (dog, 1)}
pets.groupByKey() // => {(cat, Seq(1, 2)), (dog, Seq(1))}
pets.sortByKey() // => {(cat, 1), (cat, 2), (dog, 1)}

reduceByKey also automatically implements combiners on the map side
```
Example: Word Count

```scala
val lines = sc.textFile("hamlet.txt")
val counts = lines.flatMap(line => line.split(" "))
  .map(word => (word, 1))
  .reduceByKey(_ + _)
```

Example:

<table>
<thead>
<tr>
<th>Word</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;to&quot;</td>
<td>1</td>
</tr>
<tr>
<td>&quot;be&quot;</td>
<td>1</td>
</tr>
<tr>
<td>&quot;or&quot;</td>
<td>1</td>
</tr>
<tr>
<td>&quot;not&quot;</td>
<td>1</td>
</tr>
<tr>
<td>&quot;to&quot;</td>
<td>2</td>
</tr>
<tr>
<td>&quot;be&quot;</td>
<td>2</td>
</tr>
<tr>
<td>&quot;not&quot;</td>
<td>1</td>
</tr>
<tr>
<td>&quot;or&quot;</td>
<td>1</td>
</tr>
<tr>
<td>&quot;to&quot;</td>
<td>2</td>
</tr>
<tr>
<td>&quot;be&quot;</td>
<td>2</td>
</tr>
<tr>
<td>&quot;not&quot;</td>
<td>1</td>
</tr>
<tr>
<td>&quot;or&quot;</td>
<td>1</td>
</tr>
<tr>
<td>&quot;to&quot;</td>
<td>2</td>
</tr>
<tr>
<td>&quot;be&quot;</td>
<td>2</td>
</tr>
</tbody>
</table>
Other Key-Value Operations

```scala
def visits = sc.parallelize(List(
  ("index.html", "1.2.3.4"),
  ("about.html", "3.4.5.6"),
  ("index.html", "1.3.3.1"))

val pageNames = sc.parallelize(List(
  ("index.html", "Home"), ("about.html", "About")))

visits.join(pageNames)
// ("index.html", ("1.2.3.4", "Home"))
// ("index.html", ("1.3.3.1", "Home"))
// ("about.html", ("3.4.5.6", "About"))

visits.cogroup(pageNames)
// ("index.html", (Seq("1.2.3.4", "1.3.3.1"), Seq("Home"))
// ("about.html", (Seq("3.4.5.6"), Seq("About")))
```
Controlling The Number of Reduce Tasks

All the pair RDD operations take an optional second parameter for number of tasks

```scala
words.reduceByKey(_ + _, 5)
```

```scala
words.groupByKey(5)
```

```scala
visits.join(pageViews, 5)
```

Can also set `spark.default.parallelism` property
Using Local Variables

Any external variables you use in a closure will automatically be shipped to the cluster:

```scala
val query = Console.readLine()
val pages = filter(_.contains(query)).count()
```

Some caveats:

» Each task gets a new copy (updates aren’t sent back)
» Variable must be Serializable
» Don’t use fields of an outer object (ships all of it!)
Closure Mishap Example

class MyCoolRddApp {
  val param = 3.14
  val log = new Log(...)
  ...
  def work(rdd: RDD[Int]) {
    rdd.map(x => x + param)
    .reduce(...)
  }
}

How to get around it:

class MyCoolRddApp {
  ...
  def work(rdd: RDD[Int]) {
    val param_ = param
    rdd.map(x => x + param_)
    .reduce(...)
  }
}

NotSerializableException: MyCoolRddApp (or Log)

References only local variable instead of this.param
Other RDD Operations

`sample()`: deterministically sample a subset

`union()`: merge two RDDs

`cartesian()`: cross product

`pipe()`: pass through external program

See Programming Guide for more: www.spark-project.org/documentation.html
Outline

Introduction to Scala & functional programming

Spark concepts

Tour of Spark operations

Job execution
Software Components

Spark runs as a library in your program (1 instance per app)

Runs tasks locally or on Mesos
  » dev branch also supports YARN, standalone deployment

Accesses storage systems via Hadoop InputFormat API
  » Can use HBase, HDFS, S3, ...

Your application
  SparkContext

Mesos master

Local threads

Slave Spark worker

Slave Spark worker

HDFS or other storage
Task Scheduler

Runs general task graphs

Pipelines functions where possible

Cache-aware data reuse & locality

Partitioning-aware to avoid shuffles

A: B: C: D: E: F:

Stage 1: A: B: groupBy

Stage 2: C: D: map, E: filter

Stage 3: join

 RDD = cached partition
Data Storage

Cached RDDs normally stored as Java objects
   » Fastest access on JVM, but can be larger than ideal

Can also store in serialized format
   » Spark 0.5: spark.cache.class=spark.SerializingCache

Default serialization library is Java serialization
   » Very slow for large data!
   » Can customize through spark.serializer (see later)
How to Get Started

git clone git://github.com/mesos/spark

cd spark

sbt/sbt compile

./spark-shell
More Information

Scala resources:
  » www.artima.com/scalazine/articles/steps.html (First Steps to Scala)
  » www.artima.com/pins1ed (free book)

Spark documentation:
www.spark-project.org/documentation.html