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

Improves efficiency through:
- General execution graphs
- In-memory storage

Improves usability through:
- Rich APIs in Java, Scala, Python
- Interactive shell

Up to 10× faster on disk, 100× in memory
2-5× less code
Project History

Spark started in 2009, open sourced 2010

In use at Intel, Yahoo!, Adobe, Quantifind, Conviva, Ooyala, Bizo and others

Entered Apache Incubator in June
Open Source Community

1000+ meetup members
70+ contributors
20 companies contributing

[Logos of Yahoo!, Intel, Adobe, Webtrends, Alibaba, ConViva, AdMobius, ClearStory, Tagged, WANDISCO, Bizo, Ooyala, quantiFind]
This Talk

Introduction to Spark

Tour of Spark operations

Job execution

Standalone apps
Key Idea

Write programs in terms of transformations on distributed datasets

Concept: resilient distributed datasets (RDDs)
  » 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)
Operations

Transformations (e.g. map, filter, groupBy)
  » 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

```python
lines = spark.textFile("hdfs://...")
errors = lines.filter(lambda s: s.startswith("ERROR"))
messages = errors.map(lambda s: s.split("\t")[2])
messages.cache()

messages.filter(lambda s: "foo" in s).count()
messages.filter(lambda s: "bar" in s).count()
...
```

Result: scaled to 1 TB data in 5 sec (vs 180 sec for on-disk data)
Fault Recovery

RDDs track *lineage* information that can be used to efficiently recompute lost data.

**Ex:**

```
msgs = textFile.filter(lambda s: s.startsWith("ERROR"))
          .map(lambda s: s.split("\t")[2])
```
Behavior with Less RAM

![Bar chart showing execution time (s) for different cache percentages.]

- **Cache disabled**: 69 seconds
- **25% of working set in cache**: 58 seconds
- **50% of working set in cache**: 41 seconds
- **75% of working set in cache**: 30 seconds
- **Fully cached**: 12 seconds

% of working set in cache
Spark in Scala and Java

// Scala:
val lines = sc.textFile(...)  
lines.filter(x => x.contains("ERROR")).count()

// Java:

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

Standalone programs can be written in any, but interactive shell is only Python & Scala

Python users: can do Python for both

Java users: consider learning Scala for shell

Performance: Java & Scala are faster due to static typing, but Python is often fine
Scala Cheat Sheet

Variables:

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

Collections and closures:

```scala
val nums = Array(1, 2, 3)
nums.map((x: Int) => x + 2)  // {3,4,5}
nums.map(x => x + 2)  // same
nums.map(_ + 2)  // same
nums.reduce((x, y) => x + y)  // 6
nums.reduce(_ + _)  // same
```

Functions:

```scala
def square(x: Int): Int = x*x
def square(x: Int): Int = {
    x*x  // last line returned
}
```

Java interop:

```scala
import java.net.URL
new URL("http://cnn.com").openStream()
```

More details: [scala-lang.org](http://scala-lang.org)
This Talk

Introduction to Spark

Tour of Spark operations

Job execution

Standalone apps
Learning Spark

Easiest way: the shell (spark-shell or pyspark)

» Special Scala / Python interpreters for cluster use

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

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

Main entry point to Spark functionality

Available in shell as variable sc

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

# Turn a Python collection into an RDD
sc.parallelize([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 existing Hadoop InputFormat (Java/Scala only)
sc.hadoopFile(keyClass, valClass, inputFmt, conf)
Basic Transformations

```python
nums = sc.parallelize([1, 2, 3])

# Pass each element through a function
squares = nums.map(lambda x: x*x)  # {1, 4, 9}

# Keep elements passing a predicate
even = squares.filter(lambda x: x % 2 == 0)  # {4}

# Map each element to zero or more others
nums.flatMap(lambda x: range(x))  # => {0, 0, 1, 0, 1, 2}
```
Basic Actions

```python
nums = sc.parallelize([1, 2, 3])

# Retrieve RDD contents as a local collection
nums.collect() # => [1, 2, 3]

# Return first K elements
nums.take(2) # => [1, 2]

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

# Merge elements with an associative function
nums.reduce(lambda x, y: x + y) # => 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

Python:
```python
pair = (a, b)
pair[0]  # => a
pair[1]  # => b
```

Scala:
```scala
val pair = (a, b)
pair._1 // => a
pair._2 // => b
```

Java:
```java
Tuple2 pair = new Tuple2(a, b);
pair._1 // => a
pair._2 // => b
```
Some Key-Value Operations

```python
pets = sc.parallelize(
    [(
        "cat",
        1
    ),
    (
        "dog",
        1
    ),
    (
        "cat",
        2
    )])

pets.reduceByKey(lambda x, y: x + y)
    # => {(cat, 3), (dog, 1)}

pets.groupByKey()  # => {(cat, [1, 2]), (dog, [1])}

pets.sortByKey()  # => {(cat, 1), (cat, 2), (dog, 1)}
```

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

```
lines = sc.textFile("hamlet.txt")

counts = lines.flatMap(lambda line: line.split(" "))
  .map(lambda word => (word, 1))
  .reduceByKey(lambda x, y: x + y)
```
visits = sc.parallelize([ ("index.html", "1.2.3.4"),
                        ("about.html", "3.4.5.6"),
                        ("index.html", "1.3.3.1") ])

pageNames = sc.parallelize([ ("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", (["1.2.3.4", "1.3.3.1"], ["Home"]))
# ("about.html", (["3.4.5.6"], ["About"]))
Setting the Level of Parallelism

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

```python
words.reduceByKey(lambda x, y: x + y, 5)

words.groupByKey(5)

visits.join(pageViews, 5)
```
Using Local Variables

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

```python
query = sys.stdin.readline()
pages.filter(lambda x: query in x).count()
```

Some caveats:
» Each task gets a new copy (updates aren’t sent back)
» Variable must be Serializable / Pickle-able
» 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 Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Operator</th>
<th>Operator</th>
</tr>
</thead>
<tbody>
<tr>
<td>map</td>
<td>reduce</td>
<td>sample</td>
</tr>
<tr>
<td>filter</td>
<td>count</td>
<td>take</td>
</tr>
<tr>
<td>groupBy</td>
<td>fold</td>
<td>first</td>
</tr>
<tr>
<td>sort</td>
<td>reduceByKey</td>
<td>partitionBy</td>
</tr>
<tr>
<td>union</td>
<td>groupByKey</td>
<td>mapWith</td>
</tr>
<tr>
<td>join</td>
<td>cogroup</td>
<td>pipe</td>
</tr>
<tr>
<td>leftOuterJoin</td>
<td>cross</td>
<td>save</td>
</tr>
<tr>
<td>rightOuterJoin</td>
<td>zip</td>
<td>...</td>
</tr>
</tbody>
</table>

More details: [spark-project.org/docs/latest/](http://spark-project.org/docs/latest/)
This Talk

Introduction to Spark

Tour of Spark operations

Job execution

Standalone apps
Software Components

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

Runs tasks locally or on cluster
  » Mesos, YARN or standalone mode

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

![Diagram of Spark components]

Your application

SparkContext

Cluster manager

Local threads

Worker

Spark executor

Worker

Spark executor

HDFS or other storage
Task Scheduler

General task graphs

Automatically pipelines functions

Data locality aware

Partitioning aware to avoid shuffles

<table>
<thead>
<tr>
<th>Stage 1</th>
<th>Stage 2</th>
<th>Stage 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>A:</td>
<td>C:</td>
<td>F:</td>
</tr>
<tr>
<td>B:</td>
<td>D:</td>
<td></td>
</tr>
</tbody>
</table>

= RDD  = cached partition
Advanced Features

Controllable partitioning
  » Speed up joins against a dataset

Controllable storage formats
  » Keep data serialized for efficiency, replicate to multiple nodes, cache on disk

Shared variables: broadcasts, accumulators

See online docs for details!
This Talk

Introduction to Spark

Tour of Spark operations

Job execution

Standalone apps
Add Spark to Your Project

Scala / Java: add a Maven dependency on

groupId: org.spark-project
artifactId: spark-core_2.9.3
version: 0.7.3

Python: run program with our pyspark script
Create a SparkContext

Scala

```scala
import spark.SparkContext
import spark.SparkContext._

val sc = new SparkContext("url", "name", "sparkHome", Seq("app.jar"))
```

Java

```java
import spark.api.java.JavaSparkContext

JavaSparkContext sc = new JavaSparkContext(
    "masterUrl", "name", "sparkHome", new String[] {"app.jar"})
```

Python

```python
from pyspark import SparkContext

sc = SparkContext("masterUrl", "name", "sparkHome", ["library.py"])
```
Example: PageRank

Good example of a more complex algorithm
 » Multiple stages of map & reduce

Benefits from Spark’s in-memory caching
 » Multiple iterations over the same data
Basic Idea

Give pages ranks (scores) based on links to them

» Links from many pages ➔ high rank
» Link from a high-rank page ➔ high rank
Algorithm

1. Start each page at a rank of 1
2. On each iteration, have page \( p \) contribute \( \text{rank}_p / |\text{neighbors}_p| \) to its neighbors
3. Set each page’s rank to \( 0.15 + 0.85 \times \text{contribs} \)
Algorithm

1. Start each page at a rank of 1
2. On each iteration, have page $p$ contribute $\frac{\text{rank}_p}{|\text{neighbors}_p|}$ to its neighbors
3. Set each page’s rank to $0.15 + 0.85 \times \text{contribs}$
Algorithm

1. Start each page at a rank of 1
2. On each iteration, have page $p$ contribute $\frac{\text{rank}_p}{|\text{neighbors}_p|}$ to its neighbors
3. Set each page’s rank to $0.15 + 0.85 \times \text{contribs}$
Algorithm

1. Start each page at a rank of 1
2. On each iteration, have page $p$ contribute $\frac{\text{rank}_p}{|\text{neighbors}_p|}$ to its neighbors
3. Set each page’s rank to $0.15 + 0.85 \times \text{contribs}$
Algorithm

1. Start each page at a rank of 1
2. On each iteration, have page \( p \) contribute \( \frac{\text{rank}_p}{|\text{neighbors}_p|} \) to its neighbors
3. Set each page’s rank to \( 0.15 + 0.85 \times \text{contribs} \)
Algorithm

1. Start each page at a rank of 1
2. On each iteration, have page $p$ contribute $\frac{\text{rank}_p}{|\text{neighbors}_p|}$ to its neighbors
3. Set each page’s rank to $0.15 + 0.85 \times \text{contribs}$
Scala Implementation

val sc = new SparkContext("local", "PageRank", sparkHome, Seq("pagerank.jar"))

val links = // load RDD of (url, neighbors) pairs
var ranks = // load RDD of (url, rank) pairs

for (i <- 1 to ITERATIONS) {
  val contribs = links.join(ranks).flatMap {
    case (url, (links, rank)) =>
      links.map(dest => (dest, rank/links.size))
  }
  ranks = contribs.reduceByKey(_ + _)
    .mapValues(0.15 + 0.85 * _)
}

ranks.saveAsTextFile(...
PageRank Performance

<table>
<thead>
<tr>
<th>Iteration time (s)</th>
<th>Number of machines</th>
<th>Hadoop</th>
<th>Spark</th>
</tr>
</thead>
<tbody>
<tr>
<td>171</td>
<td>30</td>
<td>23</td>
<td>0</td>
</tr>
<tr>
<td>80</td>
<td>60</td>
<td>14</td>
<td>0</td>
</tr>
</tbody>
</table>
Other Iterative Algorithms

- **K-Means Clustering**
  - Hadoop: 155 seconds
  - Spark: 4.1 seconds

- **Logistic Regression**
  - Hadoop: 110 seconds
  - Spark: 0.96 seconds

Time per Iteration (s)
Getting Started

Download Spark: spark-project.org/downloads

Documentation and video tutorials: www.spark-project.org/documentation

Several ways to run:
  » Local mode (just need Java), EC2, private clusters
Local Execution

Just pass `local` or `local[k]` as master URL

Debug using local debuggers
  » For Java / Scala, just run your program in a debugger
  » For Python, use an attachable debugger (e.g. PyDev)

Great for development & unit tests
Cluster Execution

Easiest way to launch is EC2:

```
./spark-ec2 -k keypair -i id_rsa.pem -s slaves \ 
[launch|stop|start|destroy] clusterName
```

Several options for private clusters:

- Standalone mode (similar to Hadoop’s deploy scripts)
- Mesos
- Hadoop YARN

Amazon EMR: [tinyurl.com/spark-emr](tinyurl.com/spark-emr)
Conclusion

Spark offers a rich API to make data analytics *fast*: both fast to write and fast to run

Achieves 100x speedups in real applications

Growing community with 20+ companies contributing

[www.spark-project.org](http://www.spark-project.org)