Spark

In-Memory Cluster Computing for Iterative and Interactive Applications

Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael Franklin, Scott Shenker, Ion Stoica
Environment
Motivation

Most current cluster programming models are based on *acyclic data flow* from stable storage to stable storage.
Motivation

Most current cluster programming models are based on *acyclic data flow* from stable storage to stable storage.

**Benefits of data flow:** runtime can decide where to run tasks and can automatically recover from failures.
Motivation

Acyclic data flow is inefficient for applications that repeatedly *reuse* a working set of data:
  » **Iterative** algorithms (machine learning, graphs)
  » **Interactive** data mining tools (R, Excel, Python)

With current frameworks, apps reload data from stable storage on each query
Example: Iterative Apps

Input → iteration 1 → result 1
Input → iteration 2 → result 2
Input → iteration 3 → result 3
...

Input → iter. 1 → iter. 2 → ...
Goal: Keep Working Set in RAM

Input

Distributed memory

one-time processing

iteration 1

iteration 2

iteration 3

...
Challenge

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

Existing distributed storage abstractions have interfaces based on fine-grained updates
  » Reads and writes to cells in a table
  » E.g. databases, key-value stores, distributed memory

Require replicating data or logs across nodes for fault tolerance ➔ expensive!
Solution: Resilient Distributed Datasets (RDDs)

Provide an interface based on coarse-grained transformations (map, group-by, join, ...)

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

Input

Distributed memory

one-time processing

iteration 1

iteration 2

iteration 3

Input

iter. 1

iter. 2

. . .

. . .
Generality of RDDs

Despite coarse-grained interface, RDDs can express surprisingly many parallel algorithms
  » These naturally *apply the same operation to many items*

Capture many current programming models
  » **Data flow models**: MapReduce, Dryad, SQL, ...
  » **Specialized models** for iterative apps:
    BSP (Pregel), iterative MapReduce, bulk incremental
  » Also support new apps that these models don’t
Outline

Programming interface

Applications

Implementation

Demo
Spark Programming Interface

Language-integrated API in Scala

Provides:
» Resilient distributed datasets (RDDs)
  • Partitioned collections with controllable caching
» Operations on RDDs
  • Transformations (define 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))
cachedMsgs = messages.cache()

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

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

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

**Ex:**

```
messages = textFile(...).filter(_.startsWith("ERROR"))
  .map(_.split(\'\t\')(2))
```
Example: Logistic Regression

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

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)
Logistic Regression Performance

- Running Time (s)
- Number of Iterations

<table>
<thead>
<tr>
<th>Iterations</th>
<th>Hadoop</th>
<th>Spark</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>174 s</td>
<td>174 s</td>
</tr>
<tr>
<td>5</td>
<td>6 s</td>
<td>6 s</td>
</tr>
<tr>
<td>10</td>
<td>6 s</td>
<td>6 s</td>
</tr>
<tr>
<td>20</td>
<td>127 s</td>
<td>6 s</td>
</tr>
<tr>
<td>30</td>
<td>127 s</td>
<td>6 s</td>
</tr>
</tbody>
</table>

127 s / iteration

- First iteration 174 s
- Further iterations 6 s
Example: Collaborative Filtering

Goal: predict users’ movie ratings based on past ratings of other movies

\[
R = \begin{pmatrix}
1 & ? & ? & 4 & 5 & ? & 3 \\
\end{pmatrix}
\]
Model and Algorithm

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

Alternating Least Squares (ALS)

» Start with random A & B
» Optimize user vectors (A) based on movies
» Optimize movie vectors (B) based on users
» Repeat until converged
Serial ALS

```java
var R = readRatingsMatrix(...)

var A = // array of U random vectors
var B = // array of M random vectors

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

```
var R = readRatingsMatrix(...)

var A = // array of U random vectors
var B = // array of M random vectors

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 iteration
Efficient Spark ALS

```
var R = spark.broadcast(readRatingsMatrix(...))

var A = // array of U random vectors
var B = // array of M random vectors

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()
}
```

Solution: mark R as broadcast variable

Result: 3× performance improvement
Scaling Up Broadcast

Initial version (HDFS)

Cornet broadcast

Communication

Computation

Iteration time (s)

Number of machines
Cornet Performance

1GB data to 100 receivers

Completion time (s)

- HDFS (R=3)
- HDFS (R=10)
- BitTornado
- Tree (D=2)
- Chain
- Cornet

[Chowdhury et al, SIGCOMM 2011]
Spark Applications

EM alg. for traffic prediction (Mobile Millennium)
Twitter spam classification (Monarch)
In-memory OLAP & anomaly detection (Conviva)
Time series analysis
Network simulation
...

Mobile Millennium Project

Estimate city traffic using GPS observations from probe vehicles (e.g. SF taxis)
Sample Data

Credit: Tim Hunter, with support of the Mobile Millennium team; P.I. Alex Bayen; traffic.berkeley.edu
Challenge

Data is noisy and sparse (1 sample/minute)

Must infer path taken by each vehicle in addition to travel time distribution on each link
Challenge

Data is noisy and sparse (1 sample/minute)

Must infer path taken by each vehicle in addition to travel time distribution on each link
Solution

EM algorithm to estimate paths and travel time distributions simultaneously

- observations
- flatMap
- weighted path samples
- groupByKey
- link parameters
- broadcast
Results

3× speedup from caching, 4.5× from broadcast

[Hunter et al, SOCC 2011]
Cluster Programming Models

RDDs can express many proposed data-parallel programming models

» MapReduce, DryadLINQ
» Bulk incremental processing
» Pregel graph processing
» Iterative MapReduce (e.g. Haloop)
» SQL

Allow apps to efficiently *intermix* these models
Models We Have Built

Pregel on Spark (Bagel)
  » 200 lines of code

Halooop on Spark
  » 200 lines of code

Hive on Spark (Shark)
  » 3000 lines of code
  » Compatible with Apache Hive
  » ML operators in Scala
Implementation

Spark runs on the Mesos cluster manager [NSDI 11], letting it share resources with Hadoop & other apps.

Can read from any Hadoop input source (HDFS, S3, ...)

No changes to Scala language & compiler.
Outline

Programming interface
Applications
Implementation
Demo
Conclusion

Spark’s RDDs offer a simple and efficient programming model for a broad range of apps.

Solid foundation for higher-level abstractions.

Join our open source community:

www.spark-project.org
Related Work

DryadLINQ, FlumeJava
  » Similar “distributed collection” API, but cannot reuse datasets efficiently across queries

GraphLab, Piccolo, BigTable, RAMCloud
  » Fine-grained writes requiring replication or checkpoints

Iterative MapReduce (e.g. Twister, HaLoop)
  » Implicit data sharing for a fixed computation pattern

Relational databases
  » Lineage/provenance, logical logging, materialized views

Caching systems (e.g. Nectar)
  » Store data in files, no explicit control over what is cached
## Spark Operations

<table>
<thead>
<tr>
<th>Transformations (define a new RDD)</th>
<th>Actions (return a result to driver program)</th>
</tr>
</thead>
<tbody>
<tr>
<td>map filter sample groupByKey reduceByKey sortByKey</td>
<td>collect reduce count save lookupKey</td>
</tr>
<tr>
<td>flatMap union join cogroup cross mapValues</td>
<td></td>
</tr>
</tbody>
</table>

*Transformations* involve creating a new RDD from an existing one. *Actions* are used to return a result to the driver program.
Job Scheduler

Dryad-like task DAG

Reuses previously computed data

Partitioning-aware to avoid shuffles

Automatic pipelining

A = previously computed partition
Fault Recovery Results

<table>
<thead>
<tr>
<th>Iteration</th>
<th>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>

- Blue bars represent No Failure.
- Red bars represent Failure in the 6th Iteration.
Behavior with Not Enough RAM

<table>
<thead>
<tr>
<th>% of working set in memory</th>
<th>Iteration time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cache disabled</td>
<td>68.8</td>
</tr>
<tr>
<td>25%</td>
<td>58.1</td>
</tr>
<tr>
<td>50%</td>
<td>40.7</td>
</tr>
<tr>
<td>75%</td>
<td>29.7</td>
</tr>
<tr>
<td>Fully cached</td>
<td>11.5</td>
</tr>
</tbody>
</table>