Distributed Computing with Open-Source Software

Reza Zadeh

Presented at Infosys OSSmosis
Problem

Data growing faster than processing speeds

Only solution is to parallelize on large clusters
  » Wide use in both enterprises and web industry

How do we program these things?
Outline

Data flow vs. traditional network programming
Limitations of MapReduce
Spark computing engine
Machine Learning Example
Current State of Spark Ecosystem
Data flow vs.

Traditional network programming
Traditional Network Programming

Message-passing between nodes (e.g. MPI)

Very difficult to do at scale:

» How to split problem across nodes?
  • Must consider network & data locality

» How to deal with failures? (inevitable at scale)

» Even worse: stragglers

» Ethernet networking not fast

» Have to write programs per machine

Rarely used in commodity datacenters
Data Flow Models

Restrict the programming interface so that the system can do more automatically

Express jobs as graphs of high-level operators
  » System picks how to split each operator into tasks and where to run each task
  » Run parts twice fault recovery

Biggest example:
MapReduce
Example MapReduce Algorithms

Matrix-vector multiplication

Power iteration (e.g. PageRank)

Gradient descent methods

Stochastic SVD

Tall skinny QR

Many others!
Why Use a Data Flow Engine?

Ease of programming
  » High-level functions instead of message passing

Wide deployment
  » More common than MPI, especially “near” data

Scalability to very largest clusters

Examples:
  Pig, Hive, Scalding, Storm
Limitations of MapReduce
Limitations of MapReduce

MapReduce is great at one-pass computation, but inefficient for *multi-pass* algorithms

No efficient primitives for data sharing
- State between steps goes to distributed file system
- Slow due to replication & disk storage
Example: Iterative Apps

Commonly spend 90% of time doing I/O
Result

While MapReduce is simple, it can require asymptotically more communication or I/O
Spark computing engine
Spark Computing Engine

Extends a programming language with a distributed collection data-structure
» “Resilient distributed datasets” (RDD)

Open source at Apache
» Most active community in big data, with 50+ companies contributing

Clean APIs in Java, Scala, Python

Community: SparkR
Resilient Distributed Datasets (RDDs)

Main idea: Resilient Distributed Datasets
- Immutable collections of objects, spread across cluster
- Statically typed: RDD[T] has objects of type T

val sc = new SparkContext()
val lines = sc.textFile("log.txt")  // RDD[String]

// Transform using standard collection operations
val errors = lines.filter(_.startsWith("ERROR"))
val messages = errors.map(_.split("\t")(2))  // lazily evaluated
messages.saveAsTextFile("errors.txt")  // kicks off a computation
Key Idea

Resilient Distributed Datasets (RDDs)

» Collections of objects across a cluster with user controlled partitioning & storage (memory, disk, ...)
» Built via parallel transformations (map, filter, ...)
» The world only lets you make make RDDs such that they can be:

Automatically rebuilt on failure
Python, Java, Scala, R

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

// Java (better in java8!):
JavaRDD<String> lines = sc.textFile(...);
lines.filter(new Function<String, Boolean>() {
    Boolean call(String s) {
        return s.contains("error");
    }
}).count();
Fault Tolerance

RDDS track *lineage* info to rebuild lost data

```scala
file.map(lambda rec: (rec.type, 1))
  .reduceByKey(lambda x, y: x + y)
  .filter(lambda (type, count): count > 10)
```
Fault Tolerance

RDDs track *lineage* info to rebuild lost data

```python
file.map(lambda rec: (rec.type, 1))
   .reduceByKey(lambda x, y: x + y)
   .filter(lambda (type, count): count > 10)
```
Machine Learning example
Logistic Regression

\[ w \leftarrow w - \alpha \cdot \sum_{i=1}^{n} g(w; x_i, y_i) \]

```
val points = spark.textFile(...).map(parsePoint).cache()
var w = Vector.zeros(d)
for (i <- 1 to numIterations) {
  val gradient = points.map { p =>
    (1 / (1 + exp(-p.y * w.dot(p.x))) - 1) * p.y * p.x
  }.reduce(_ + _)
  w -= alpha * gradient
}
```
Logistic Regression Results

Running Time (s)

Number of Iterations

first iteration 80 s
further iterations 1 s

110 s / iteration

Hadoop
Spark

100 GB of data on 50 m1.xlarge EC2 machines
Behavior with Less RAM

The graph shows the iteration time (s) for different percentages of the working set in memory. The x-axis represents the percentage of the working set in memory (0%, 25%, 50%, 75%, 100%), and the y-axis represents the iteration time in seconds, ranging from 0 to 100. The iteration time decreases as the percentage of working set in memory increases, with the following values:

- 0%: 68.8 seconds
- 25%: 58.1 seconds
- 50%: 40.7 seconds
- 75%: 29.7 seconds
- 100%: 11.5 seconds

The error bars indicate the variability or uncertainty in the data.
Benefit for Users

Same engine performs data extraction, model training and interactive queries

Separate engines

Spark

DFS read parse train query

DFS read parse train query

DFS read train write

DFS read train write

DFS read query write

DFS read query write

...
State of the Spark ecosystem
Spark Community

Most active open source community in big data

200+ developers, 50+ companies contributing

Contributors in past year

- Spark
- Hadoop
- MapReduce

Logos of companies like Yahoo, Intel, Adobe, IBM, Red Hat, Amazon Web Services, eBay, Cloudera, DataStax, MapR, Alibaba, ClearStory, WebTrends, Bizo, Conviva, and Databricks are also present on the page.
Continuing Growth

Contributors per month to Spark

source: ohloh.net
A General Platform

Standard libraries included with Spark

- Spark SQL
- Spark Streaming
- GraphX
- MLlib
Conclusion

Data flow engines are becoming an important platform for numerical algorithms

While early models like MapReduce were inefficient, new ones like Spark close this gap

More info: [spark.apache.org](http://spark.apache.org)