a Spark in the cloud

iterative and interactive cluster computing

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

RAD Lab
UC Berkeley

Background

MapReduce and Dryad raised level of abstraction in cluster programming by hiding scaling & faults

However, these systems provide a limited programming model: acyclic data flow

Can we design similarly powerful abstractions for a broader class of applications?

Spark Goals

Support applications with *working sets* (datasets reused across parallel operations)

» Iterative jobs (common in machine learning)
» Interactive data mining

Retain MapReduce's fault tolerance & scalability

Experiment with programmability

» Integrate into Scala programming language
» Support interactive use from Scala interpreter

Non-goals

Spark is not a general-purpose programming language

- One-size-fits-all architectures are also do-nothing-well architectures

Spark is not a scheduler, nor a resource manager

Mesos

- Generic resource scheduler with support for heterogeneous frameworks

Programming Model

Resilient distributed datasets (RDDs)

- » Created from HDFS files or "parallelized" arrays
- » Can be transformed with map and filter
- » *Can be cached across parallel operations*

Parallel operations on RDDs

- » Reduce, toArray, foreach

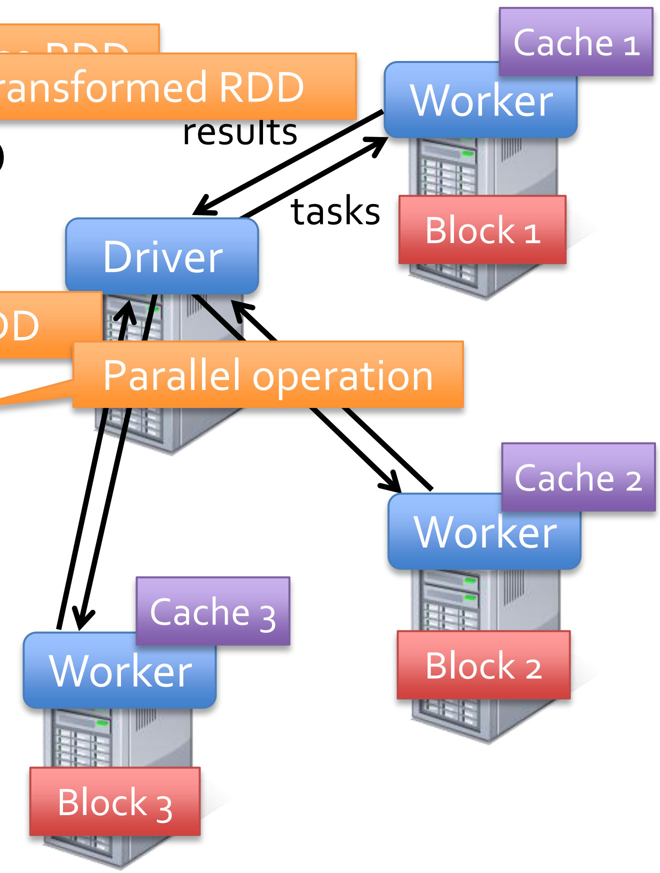
Shared variables

- » Accumulators (add-only), broadcast variables

Example: Log Mining

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

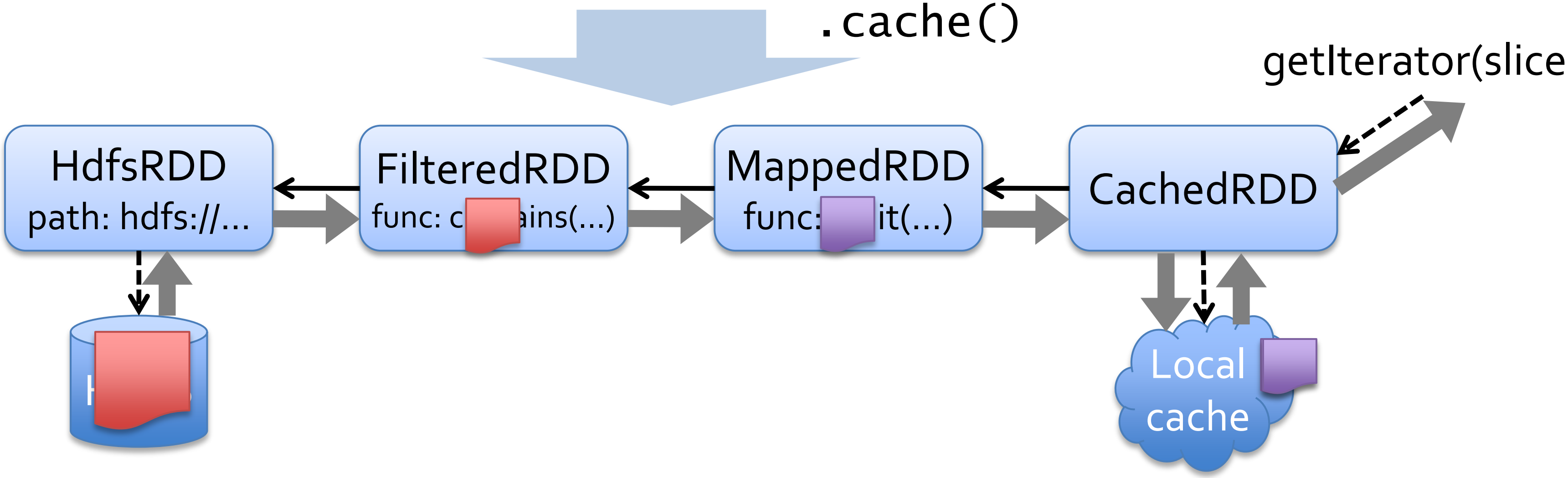
```
lines = spark.textFile("hdfs://...")
errors = lines.filter(_.startsWith("ERROR"))
messages = errors.map(_.split('\t')(1))
cachedMsgs = messages.cache()

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



RDD Representation

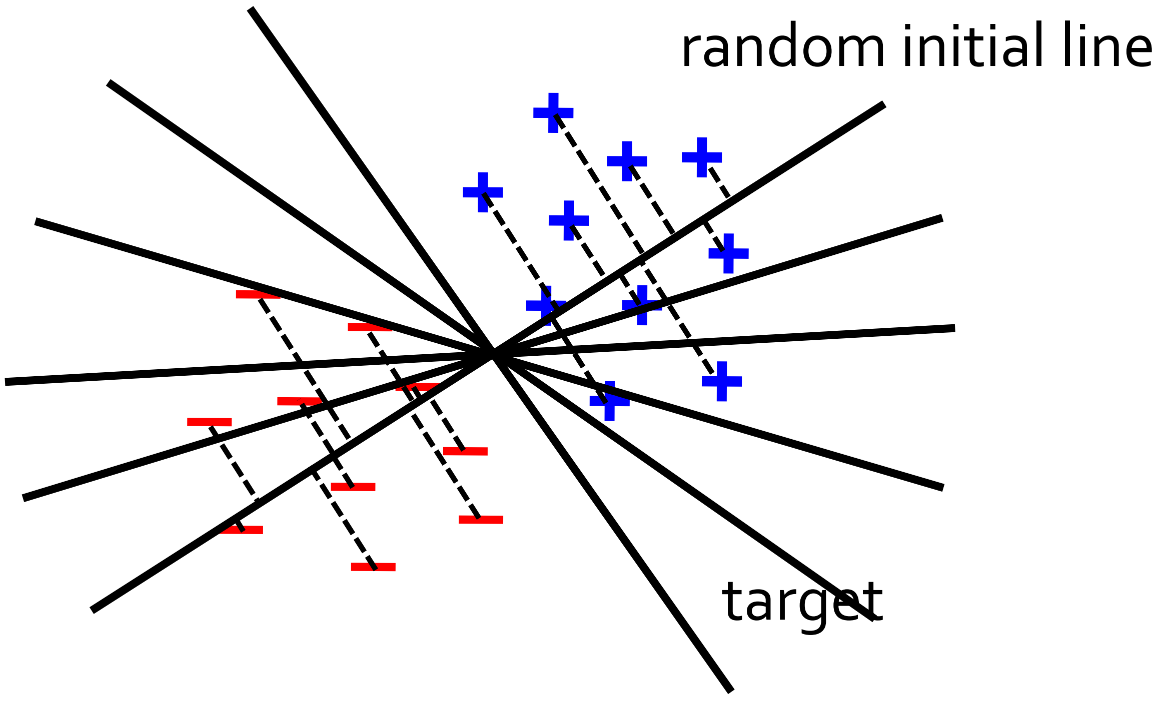
Each RDD object maintains a *lineage* that can be used to rebuild slices of it that are lost / fall out of cache

Ex:
```
cachedMsgs =
textFile("log").filter(_.contains("error"))
                                .map(_.split('\t')(1))
                                .cache()
```



Example: Logistic Regression

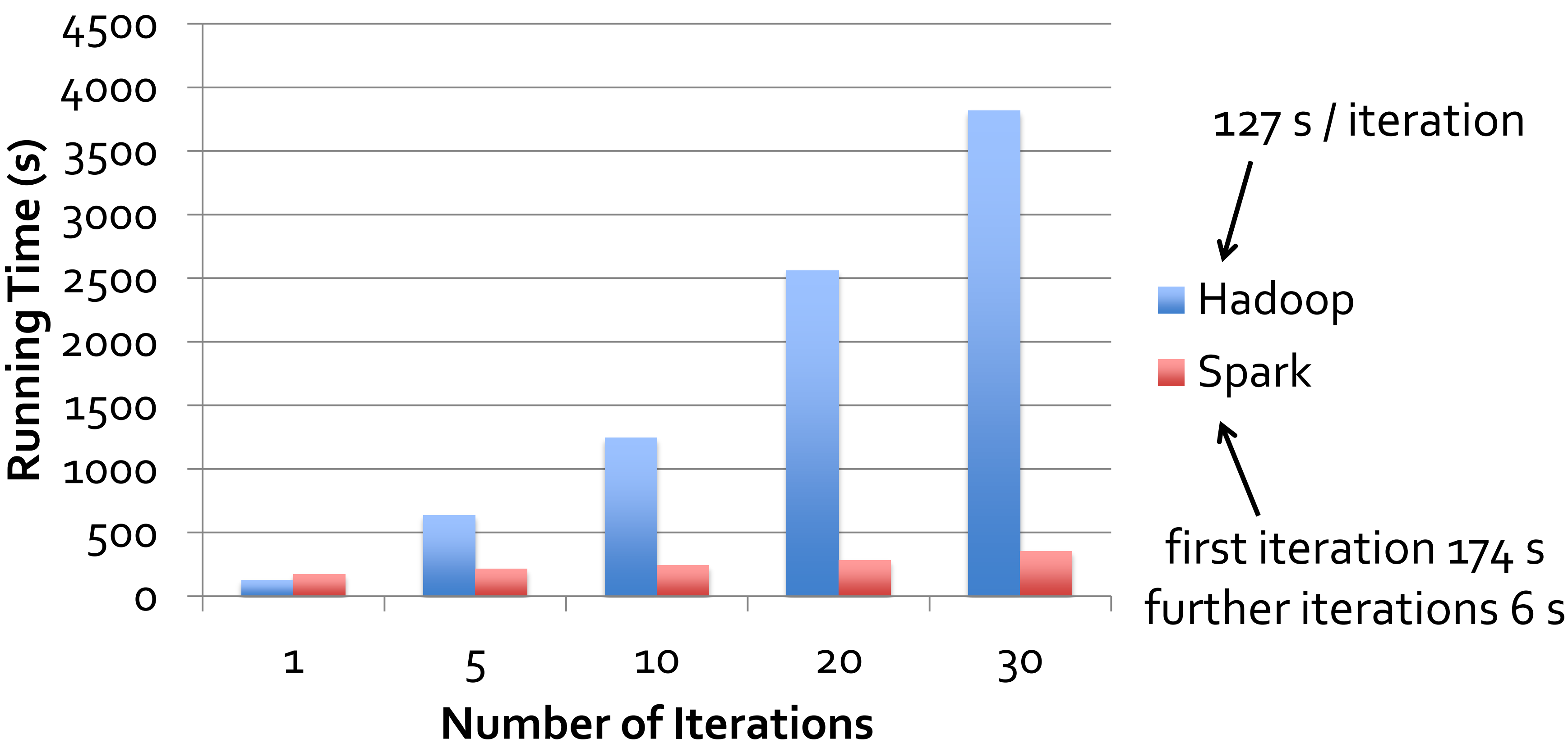
Goal: find best line separating two sets of points



Logistic Regression Code

```
val data = spark.textFile(...).map(readPoint).cache()

var w = Vector.random(D)

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

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

Logistic Regression Performance



Example: Collaborative Filtering

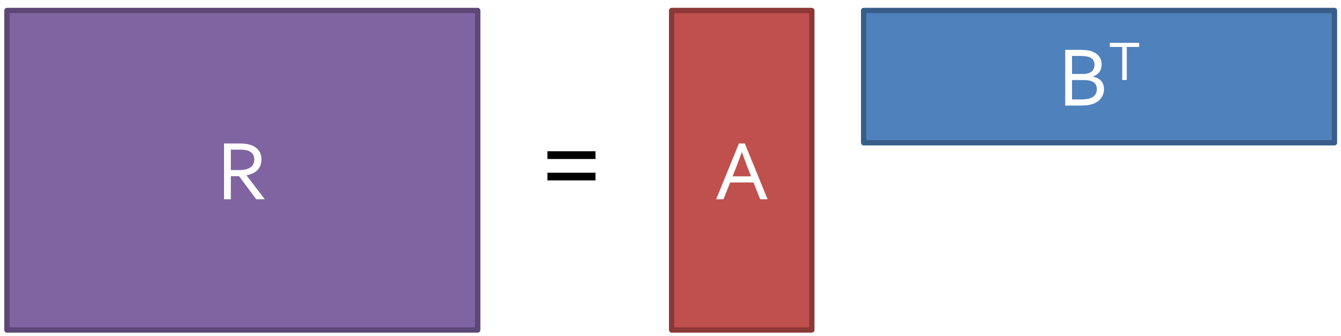
Predict movie ratings for a set of users based on their past ratings of other movies

$$R = \begin{pmatrix} 1 & ? & ? & 4 & 5 & ? & 3 \\ ? & ? & 3 & 5 & ? & ? & 3 \\ 5 & ? & 5 & ? & ? & ? & 1 \\ 4 & ? & ? & ? & ? & 2 & ? \end{pmatrix} \quad \updownarrow \text{Users}$$

$$\longleftarrow \text{Movies} \longrightarrow$$

Matrix Factorization Model

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



Problem: given subset of R, optimize A and B

Alternating Least Squares

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))
           .toArray()
  B = spark.parallelize(0 until M, numSlices)
           .map(i => updateMovie(i, A, R))
           .toArray()
}
```

Problem:
R re-sent to all nodes in each parallel operation

Efficient Spark ALS

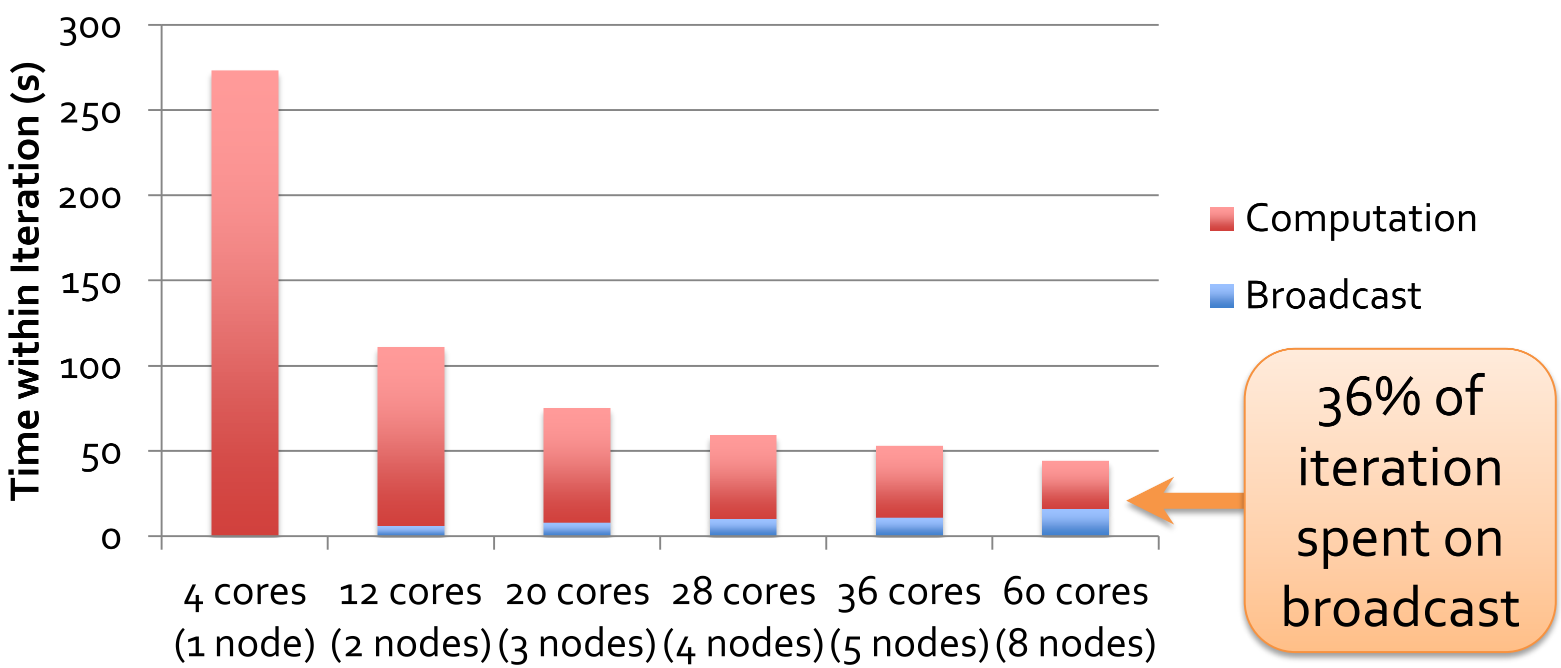
```
val R = spark.broadcast(readRatingsMatrix(...))

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))
           .toArray()
  B = spark.parallelize(0 until M, numSlices)
           .map(i => updateMovie(i, A, R.value))
           .toArray()
}
```

Solution: mark R as "broadcast variable"

How to Implement Broadcast?

Just using broadcast variables gives a significant performance boost, but not enough for all apps

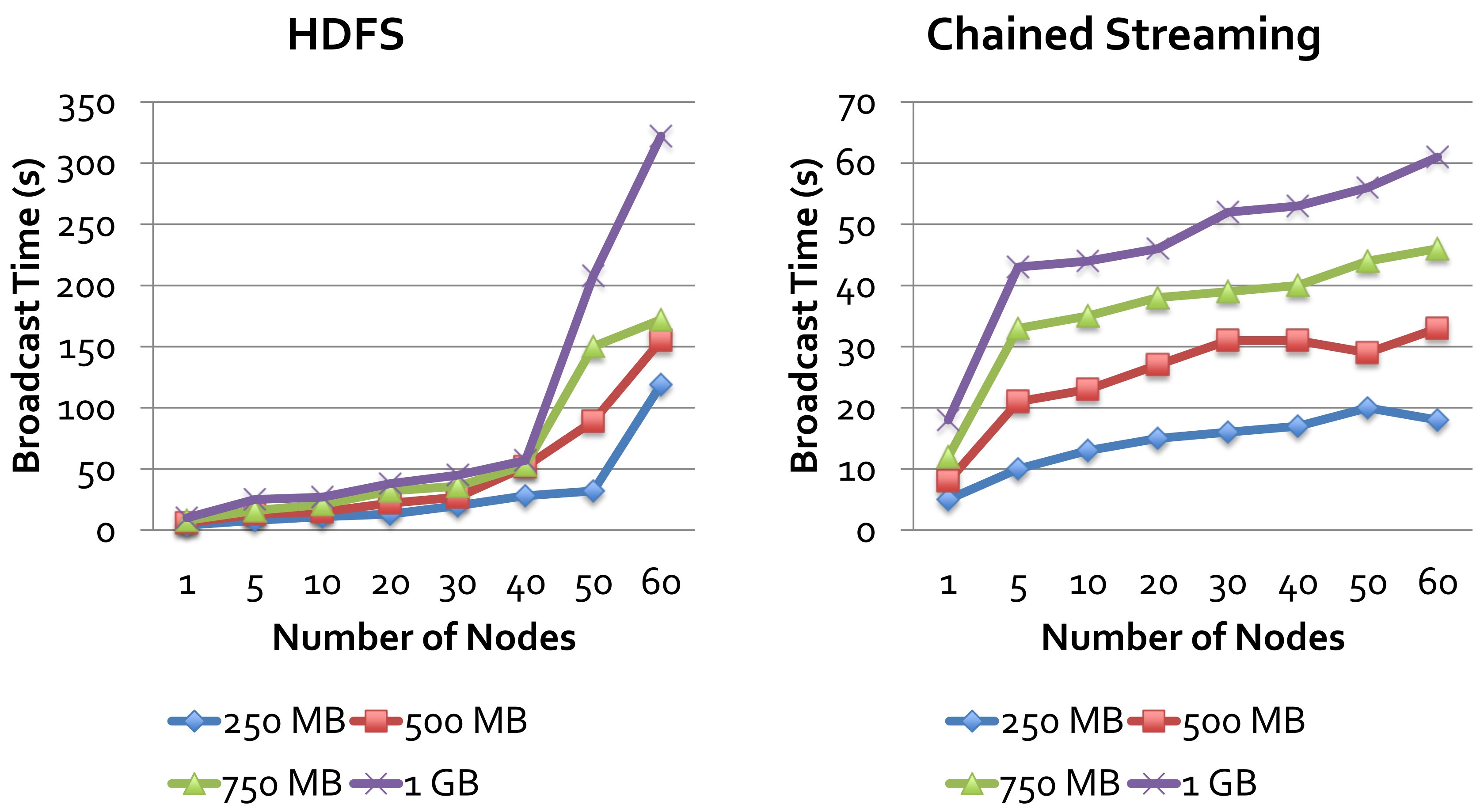
Example: ALS broadcasts 100's of MB / iteration, which quickly bottlenecked our initial HDFS-based broadcast



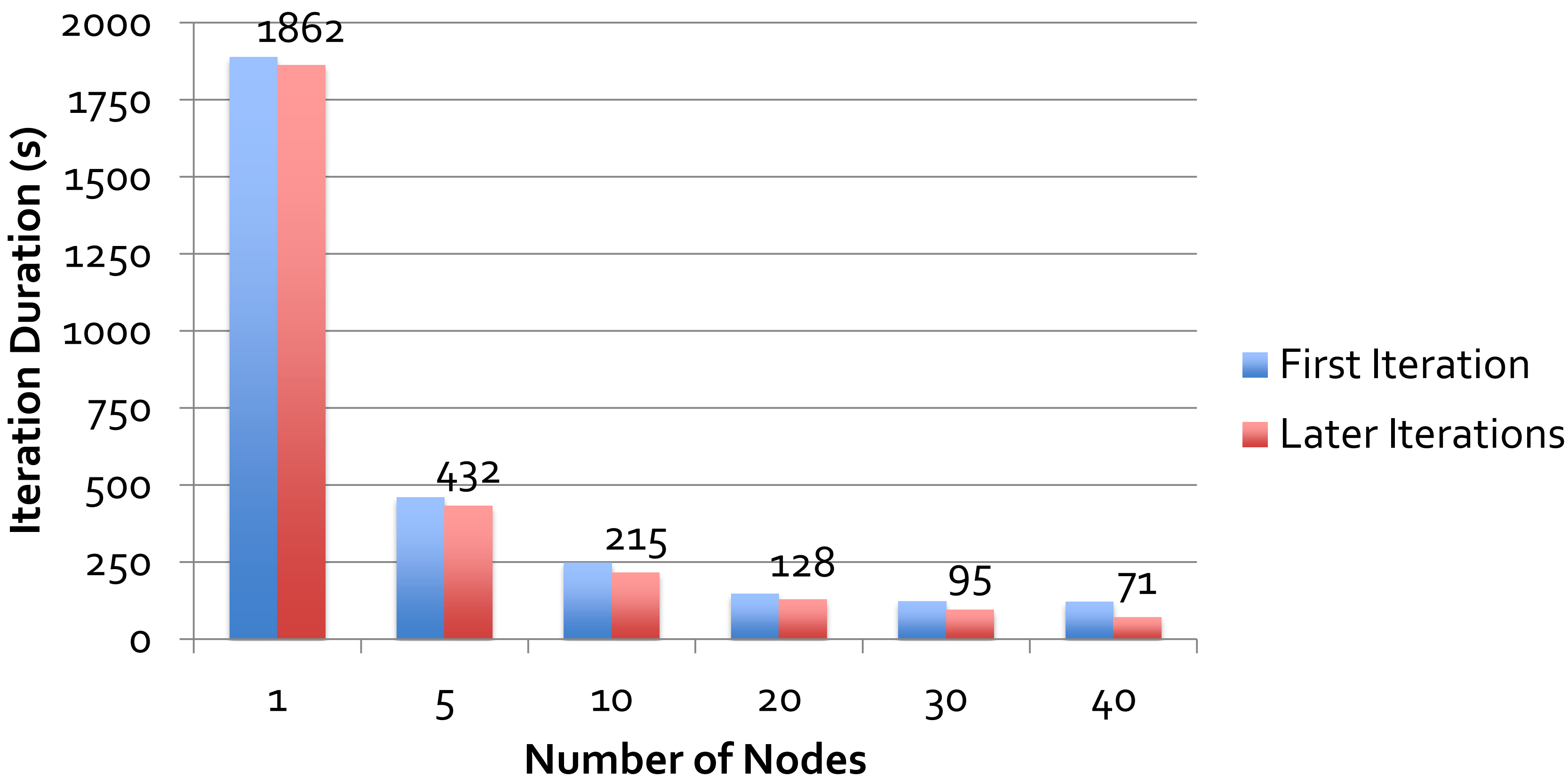
Broadcast Methods Explored

Method	Results
NFS	Server becomes bottleneck
HDFS	Scales further than NFS, but limited
Chained Streaming	Initial results promising, but straggler nodes cause problems
BitTorrent	Off-the-shelf BT adds too much overhead in data center environment
SplitStream	Scales well in theory, but needs to be modified for fault tolerance

Broadcast Results



ALS Performance with Chained Streaming Broadcast



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

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

Demo

Conclusions

Spark provides a limited but efficient set of fault tolerant distributed memory abstractions

- Resilient distributed datasets (RDDs)
- Restricted shared variables

Planned extensions:

- More RDD transformations (e.g., shuffle)
- More RDD persistence options (e.g., disk + memory)
- Updatable RDDs (for incremental or streaming jobs)
- Data sharing across applications

Related Work

DryadLINQ
- Build queries through language-integrated SQL operations on lazy datasets
- Cannot have a dataset persist *across* queries
- No concept of shared variables for broadcast etc.

Pig and Hive
- Query languages that can call into Java/Python/etc UDFs
- No support for caching a datasets 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

Twister and Haloop
- Iterative MapReduce implementations using caching
- Can't define multiple distributed datasets, run multiple map & reduce pairs on them, or decide which operations to run next interactively

Questions

Backup

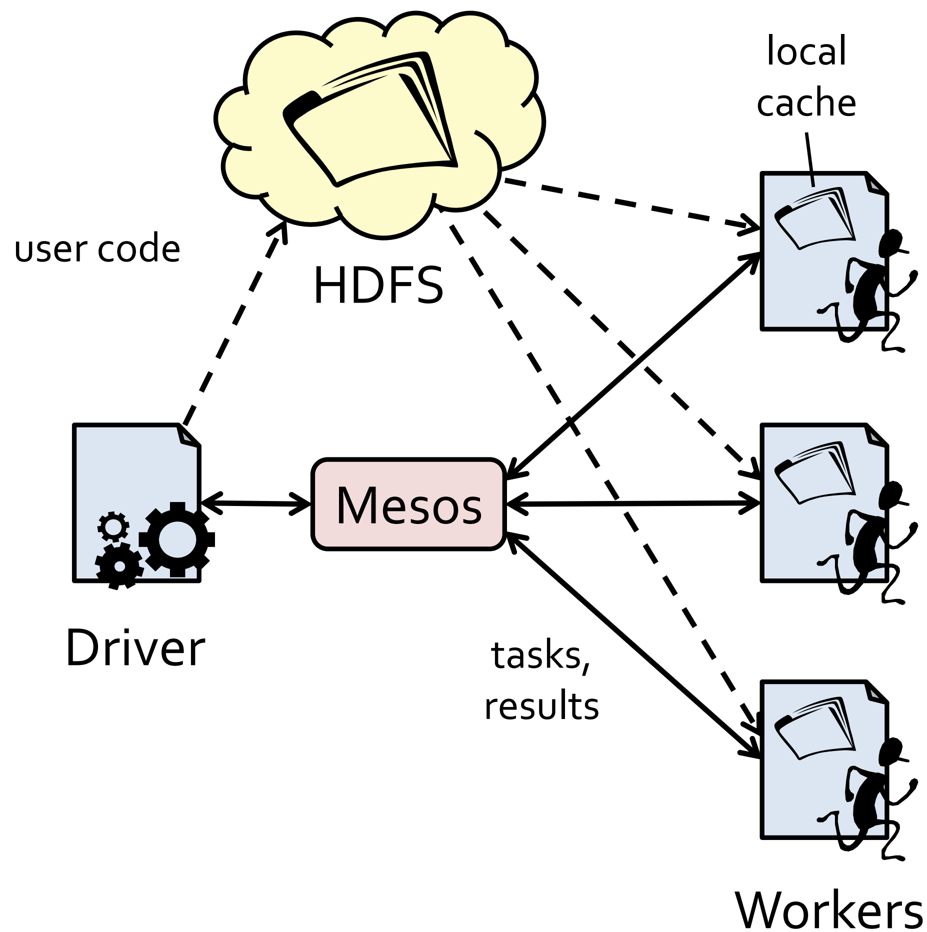
Architecture

Driver program connects to Mesos and schedules tasks

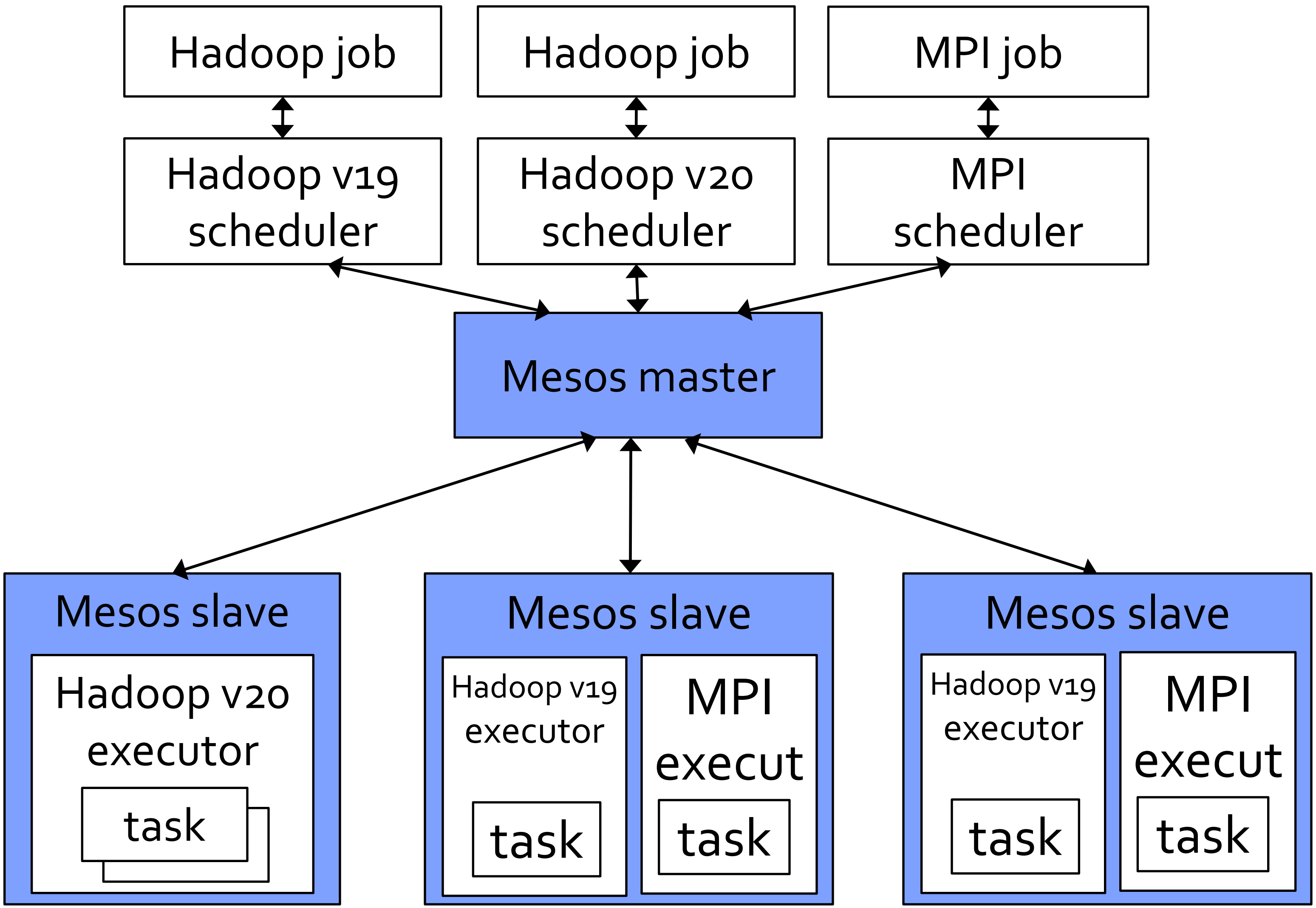
Workers run tasks, report results and variable updates

Data shared with HDFS/NFS

No communication between workers for now



Mesos Architecture



Serial Version

```
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

```
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)
```

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)
```

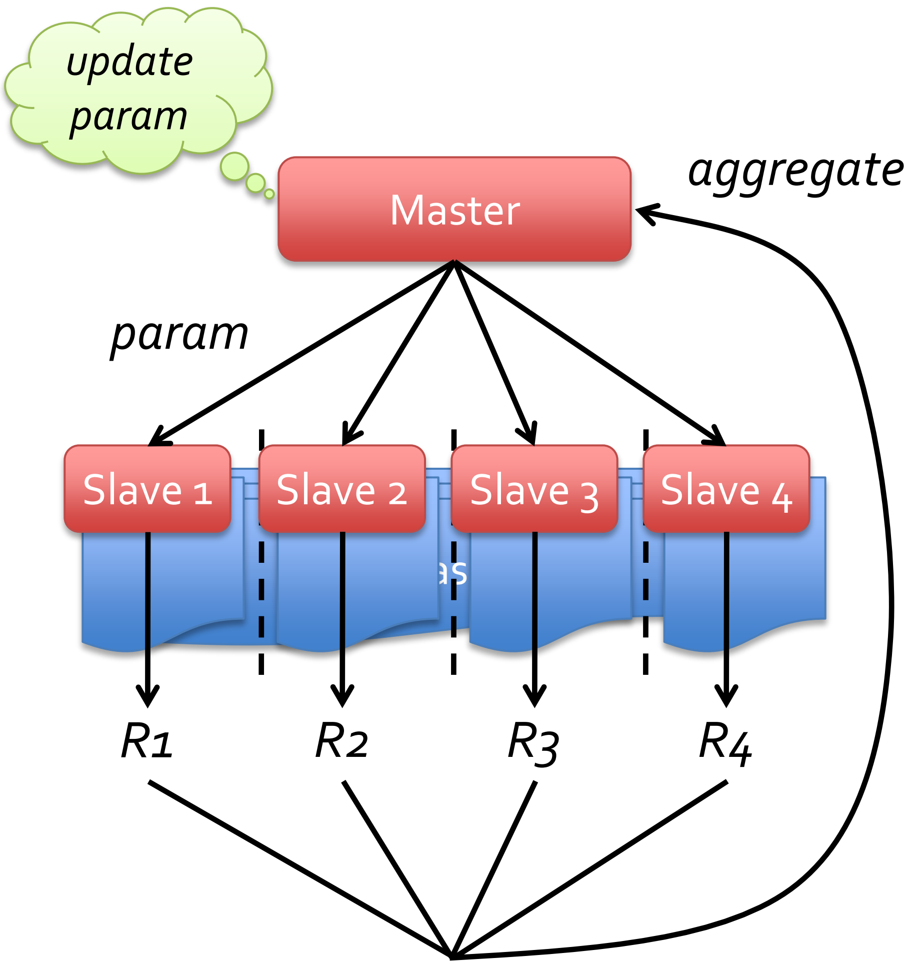
Spark Version

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

var w = Vector.random(D)

for (i <- 1 to ITERATIONS) {
  var gradient = spark.accumulator(Vector.zeros(D))
  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

Job Execution