Numerical Computing with Spark

Hossein Falaki
Challenges of numerical computation over big data

When applying any algorithm to big data watch for

1. Correctness
2. Performance
3. Trade-off between accuracy and performance
Three Practical Examples

- Point estimation (Variance)
- Approximate estimation (Cardinality)
- Matrix operations (PageRank)

We use these examples to demonstrate Spark internals, data flow, and challenges of implementing algorithms for Big Data.
1. Big Data Variance

The plain variance formula requires two passes over data.

\[ \text{Var}(X) = \frac{1}{N} \sum_{i=1}^{N} (x_i - \mu)^2 \]
Fast but inaccurate solution

\[ \text{Var}(X) = E[X^2] - E[X]^2 \]

\[ = \frac{\sum x^2}{N} - \left( \frac{\sum x}{N} \right)^2 \]

- Can be performed in a single pass, but
- Subtracts two very close and large numbers!
Accumulator Pattern

An object that incrementally tracks the variance

```scala
class RunningVar {
  var variance: Double = 0.0

  // Compute initial variance for numbers
  def this(numbers: Iterator[Double]) {
    numbers.foreach(this.add(_))
  }

  // Update variance for a single value
  def add(value: Double) {
    ...
  }
}
```
Parallelize for performance

- Distribute adding values in map phase
- Merge partial results in reduce phase

```scala
Class RunningVar {
  ...  
  // Merge another RunningVar object  
  // and update variance
  def merge(other: RunningVar) = {
    ...  
  }
}
```
Computing Variance in Spark

• Use the RunningVar in Spark

```java
doubleRDD
  .mapPartitions(v => Iterator(new RunningVar(v)))
  .reduce((a, b) => a.merge(b))
```

• Or simply use the Spark API

```java
doubleRDD.variance()
```
2. Approximate Estimations

• Often an approximate estimate is *good enough* especially if it can be computed faster or cheaper

  1. Trade accuracy with memory

  2. Trade accuracy with running time

• We really like the cases where there is a bound on error that can be controlled
Cardinality Problem

Example: Count number of unique words in Shakespeare’s work.

- Using a HashSet requires \(~10\)GB of memory
- This can be much worse in many real world applications involving large strings, such as counting web visitors
Linear Probabilistic Counting

1. Allocate a bitmap of size $m$ and initialize to zero.
   A. Hash each value to a position in the bitmap
   B. Set corresponding bit to 1

2. Count number of empty bit entries: $v$

\[
\text{count} \approx -m \ln \frac{v}{m}
\]
The Spark API

• Use the LogLinearCounter in Spark

```java
rdd
  .mapPartitions(v => Iterator(new LPCCounter(v)))
  .reduce((a, b) => a.merge(b)).getCardinality
```

• Or simply use the Spark API

```java
myRDD.countApproxDistinct(0.01)
```
3. Google PageRank

Popular algorithm originally introduced by Google
PageRank Algorithm

• Start each page with a rank of 1

• On each iteration:

  A. \( contrib = \frac{curRank}{|neighbors|} \)

  B. \( curRank = 0.15 + 0.85 \sum_{neighbors} contrib_i \)
PageRank Example
PageRank Example
PageRank Example

![PageRank Diagram]

- Page 1: 1.85
- Page 2: 0.58
- Page 3: 0.58
- Page 4: 1.0
PageRank Example
PageRank Example

0.39 → 1.31 → 1.72
0.58 → 1.31
PageRank Example

Eventually
PageRank as Matrix Multiplication

• Rank of each page is the probability of landing on that page for a random surfer on the web

• Probability of visiting all pages after k steps is

\[ V_k = A^k \times V^t \]

\( V \): the initial rank vector
\( A \): the link structure (sparse matrix)
Data Representation in Spark

- Each page is identified by its unique URL rather than an index
- Ranks vectors ($\mathbf{V}$): RDD[((URL, Double))]
- Links matrix ($\mathbf{A}$): RDD[((URL, List(URL)))]
Spark Implementation

```scala
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(...)```

Matrix Multiplication

- Repeatedly multiply sparse matrix and vector

...
Spark can do much better

• Using cache(), keep neighbors in memory
• Do not write intermediate results on disk
Spark can do much better

- Do not partition neighbors every time
Spark Implementation

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

links.partitionBy(hashFunction).cache()

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(...)

```
```
Conclusions

When applying any algorithm to big data watch for

1. Correctness

2. Performance
   - Cache RDDs to avoid I/O
   - Avoid unnecessary computation

3. Trade-off between accuracy and performance
Numerical Computing with Spark