Big Data Frameworks: Developing Spark Algorithms

31.03.2015

Eemil Lagerspetz

These slides: http://is.gd/bigdataalgo

Outline

- Part I: Intro
  - Use Cases
  - Spark Vision
  - MLlib
  - Some Algorithms
- Part II: How to Design Spark Algorithms
  - Designing Spark algorithms
  - Important numbers
  - The Hard Part
  - Idioms and Patterns
  - Tips

Developing Spark algorithms: Why?

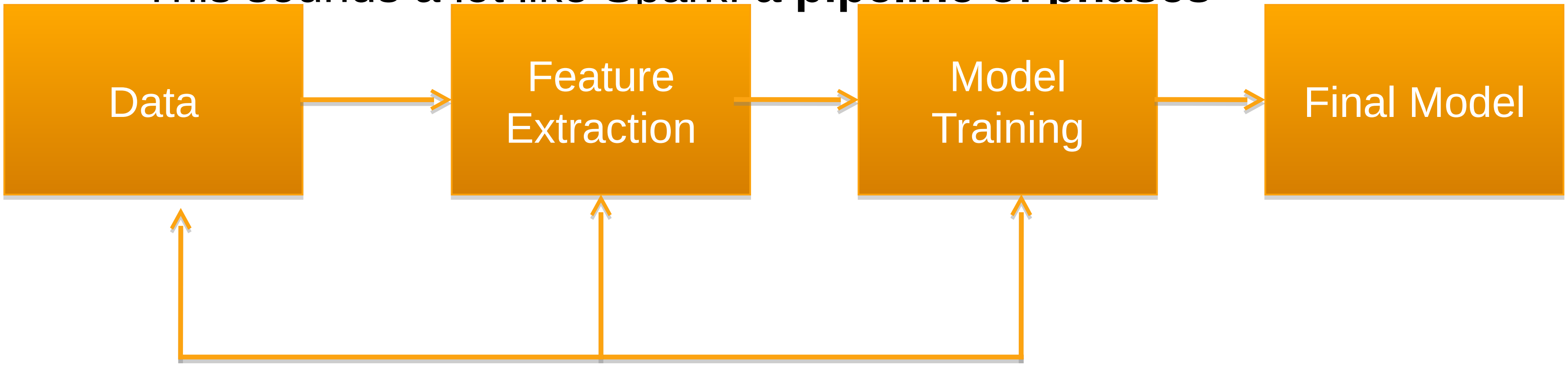
- Why not Hadoop or <Insert another framework here>?
  - Richer programming model
  - Not just Map and Reduce (and Combine)
  - Speed and In-Memory computation
- With Hadoop/Map-Reduce, it is difficult to represent complex algorithms

Spark Algorithms: Use Cases

- Analytics and Statistics, Data Mining, Machine Learning, …
  - Pattern recognition, anomaly detection (spam, malware, fraud)
  - Identification of key or popular topics
  - Content classification and clustering, recommender systems
- Large-scale, Scalable Systems
- More Efficient Parallel Algorithms
  - You don't need to implement the parallelism every time
- Cost Optimization, Flexibility – Cloud instead of Grid

Analytics and Statistics

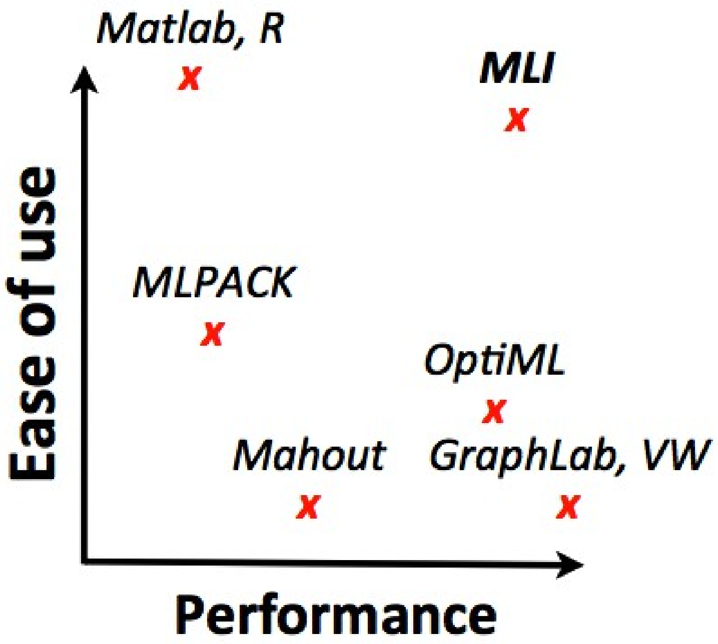
- The aim of data analytics is to find new information in data
- The process builds on statistics
- Why Spark? Processing large datasets
  - Past and current data
- The overall process has many steps
  - Data selection, preprocessing, transformations, data mining and development of patterns and models, interpretation and evaluation
  - This sounds a lot like Spark: **a pipeline of phases**



Data mining techniques

- Classification, Clustering
- Collaborative filtering
- Dimensionality reduction
- Frequent pattern mining
- Regression, Anomaly detection
- Supervised learning, Feature learning, Online learning
- Topic models
- Unsupervised learning
- All of this can probably be done with Spark, but **may require case-by-case algorithm redesign**
  - Why? **Ability to process very large datasets**

MLlib



- *MLI (MLlib): An API for Distributed Machine Learning*
- **Evan Sparks**, **Ameet Talwalkar**, et al.
- International Conference on Data Mining (2013)
- **http://arxiv.org/abs/1310.5426**

Algorithms in MLlib v1.0

- Classification
  - logistic regression, linear support vector machines (SVM), naïve Bayes, least squares, decision trees
- Regression
  - linear regression, regression trees
- Collaborative filtering
  - alternating least squares (ALS), non-negative matrix factorization (NMF)
- Clustering
  - k-means
- Optimization
  - stochastic gradient descent (SGD), limited memory BFGS
- Dimensionality reduction
  - singular value decompositon (SVD), principal component analysis (PCA)

Spark K-Means Example

```
val data = sc.textFile("kmeans.txt")
val parsedData = data.map(_.split(" ")
  .map(_.toDouble)).cache()


val clusters = KMeans.train(parsedData, 2,
  numIterations=20)


val cost = clusters.computeCost(parsedData)


println("Sum of squared errors: " + cost)
```

Source: MLlib and Distributing the Singular Value Decomposition, Reza Zadeh, ICME and Databricks, 2014.

Without MLLib

```
// Initialize K cluster centers
centers = data.takeSample(false, K, seed)
while (d > epsilon) {
  // assign each data point to the closest cluster
  closest = data.map( p =>
    (closestPoint(p, centers), p))
    // assign each center to be the mean of its data points
    pointsGroup = closest.groupByKey()
    newCenters = pointsGroup.mapValues(
      ps => average(ps))
    d = distance(centers, newCenters)
}
```

S. Venkataraman. Machine Learning on Spark. Strata Conference, February 2013.

Part II

- Developing Spark Algorithms
- How to convert existing local algorithms to the Spark parallel model

Developing Spark Algorithms

- Do not duplicate work
- Check if it exists in MLlib
- Is there a Spark Package for it? http://spark-packages.org/
- Has someone made it for Hadoop?
- If no, then…
  - Find a pseudocode or the math
  - Think it through in a parallel way
    - **The hard part**

The Hard Part: Global State 1/2

- Check for immutable global data structures
- and replace them with Broadcasts in Spark

```
int[] supportData = {0, 1, 2, 3, 4}  →
val supportArray = Array(0, 1, 2, 3, 4)
val supportData = sc.broadcast(supportArray)
```

The Hard Part: Global State 2/2

- Check for mutable global data structures
  - Change them to Broadcasts, and only change their content after a transformation/action is complete

```
int[] mutableData = {0, 1, 2, 3, 4}  →
var mut = sc.broadcast(Array(0, 1, 2, 3, 4))
val updated:Array[(Int, Int)] = dataRdd.map{x =>
  x%5 -> mut.value(x%5)*x
  /* which index was updated, and what is the new value */
}.reduceByKey(_ + _).collect.sortBy(_._1)
mut = sc.broadcast(updated.map(_._2))
```

- And then loop again ...

Dealing with sliding windows

```
int[] data = input;
int[] result = new int[data.length];
for (int i = 1; i < data.length; i++){
  result[i-1] = data[i] * data[i-1];
}
```

- The above takes pairs starting from 0, 1
- In Spark we can use zip to do this

Dealing with sliding windows

```
val data0 = 12
val rdd = sc.textFile("input")
val paired:RDD[(Int, Int)] =
  rdd.dropRight(1).zip(rdd.drop(1))
paired.map(p =>
  p._1*p._2).saveAsTextFile("result")
```

- If we also need ordering, we need to
  - Have line numbers in the original file, or
  - Use `rdd.zipPartitions(rdd2)` `.mapPartitionsWithIndex(...)`

Dealing with ordering

```
val ord = rdd.mapPartitionsWithIndex{part =>
  val (idx:Int, items:Iterable[(Int, Int)]) = part
  items.toArray.zipWithIndex.map{p =>
    val ((p1, p2), index) = p
    (idx, index) -> p._1*p._2 }}
```

Now items are prefixed with file part and their position in that part, e.g. `(0, 0), (0, 1), (0, 2),` … `(1, 0), (1, 1), (1, 2),` …

We can sort by this and save the results to preserve order

```
ord.sortBy(_._1).map(_._2)
  .saveAsTextFile("results")
```

The Hard Part: Interdependent loops

- Check for state dependencies in loops

```
int[] data = new int[n];
for (int i = 0; i < n; i++) {
  for (int j = 0; j < i; j++)
    data[i] += i * data[j];
}
```

- This depends on all the previously calculated values
  - It will be very hard to convert this kind of algorithm to Spark, perhaps find another approach
  - Or figure it out case-by-case...

The Hard Part: Figuring it out

```
int[] data = new int[n];
for (int i = 0; i < n; i++) {
  for (int j = 0; j < i; j++)
    data[i] += i * data[j];
}
```

- The above just does `i * sum(0 until i)` (i is not included)
- We can do that in Spark easily:

```
val rdd = sc.parallelize(0 until n)
val finalData = rdd.map(i => i * (0 until i).sum)
```

- So, even interdependent loops can be converted, if you figure them out

Hard Part over …

- Spark Practicals next
- (You can wake up now)

Important Numbers 1/3

- Data size: How many GB? How large files? How many files?
  - Hadoop/Spark prefers ~ 500MB files
- How many tasks?
  `sc.textFile("data", tasks)`
  - Minimum: 2xCPU cores in cluster
  - Optimal: Each computer's memory is used fully for tasks
  - Maximum: Too large → high overhead
- Spark does not tune this for you – It depends on your job

Important Numbers 2/3

- How many objects? How large are they? How many cores per computer, how many objects does each process?
  - --conf spark.task.cpus=X to control this
- Example VM: 8 cores/tasks, 32GB RAM → Spark has 4GB / core. Too little?
  - Set cpus=2 and Spark will assign 8/2 = 4 tasks to the node at a time.

Important Numbers 3/3

- Does your task cause the data size to grow? How much?
  - Deserialization, data grouping, data multiplication with cartesian() or tasks size possibly doubling with join()
- `rdd.mapPartitions(func)` makes one task handle one file/partition, this is more efficient if your objects are small
  - With this, all the objects of one partition are available at the VM
- `rdd.mapPartitions( part => part.map( x => x*2 ))`
- Results in the same thing as

  `rdd.map( x => x*2 )`
- Tip: mapPartitions returns an Iterable, so you can do filtering too before returning it

Practicals: import SparkContext._

Import org.apache.spark.SparkContext
Import SparkContext._

- The above is needed so that RDDs have groupByKey etc advanced operations
    - This imports Spark's implicit imports, like PairedRDD functions (key-value stuff)

Practicals: Keys and Values

- In Spark, any RDD of type (a, b) is an RDD of keys and values
  - Keys can be used in groupByKey, reduceByKey, etc.
- Example:

```
val data =
  Array( ((0,1), "first"),((1,0), "second"))
val rdd = sc.parallelize(data)
// rdd: RDD[((Int, Int), String)]
```

- Here (Int, Int) is the key
- Key can be any class that implements hashCode, for example, any Tuple (of any length), or any case class
  - Optionally, you can implement Ordering to allow sortByKey

Spark key-value RDD pattern

- Use RDDs of type k -> v to allow reduceByKey, groupByKey, sortByKey, ...
  - Cleaner than rdd.groupBy(_.x)
- Load text data directly to (k, v) RDD:

```
case class MovieInfo(title:String, genres: Array[String])
val txt = sc.textFile("moviedata")
val movies = txt.map{ line => line.split("::") match {
  case Array(id, title, genres, _*) =>
    id.toInt -> new MovieInfo(title, genres.split("|"))
  }
}  // movies: RDD[(Int, MovieInfo)]
```

Naming Tuple Fields in Transformations

```
something.map{ x =>
  val (a, b, c, d, e) = x
  (a, c, d) }
```

- Allows splitting a tuple to elements on the first line
- Naming: no need to get confused by tuple indices

```
something.map{ x =>
  val (a, (b, c, d, e)) = x
  (c, (a, e)) }
```

- Supports nested tuples like above. Can be done with case too:

```
something.map{ case (a, (b, c, d, e)) => (c, (a, e)) }
```

Tips: Joining two datasets

```
val xrdd = sc.textFile(p).map(x => x.head -> x.tail)
val yrdd = sc.textFile(q).map(y => y.head -> y.tail)
```

- First element is always key, second the data

```
val grouped = xrdd.groupWith(yrdd)

// grouped: RDD[(Char, (Iterable[String], Iterable[String]))]
```

- This is the same as

```
val xg = xrdd.groupByKey
val yg = yrdd.groupByKey
val grouped = xg.join(yg)
```

Idioms

- Text data parsing: Scala pattern match Idiom

```
case class Movie(id:Int, title:String, genres:
Array[String])
val txt = sc.textFile("moviedata")
val movies = txt.map{ line =>  line.split("::") match {
  case Array(id, title, genres) =>
    new Movie(id.toInt, title, genres.split("|"))
  }
}
```

- The use of the pattern matcher avoids array size exceptions
- Allows naming fields of the split Array directly at the case statement

Idioms

- Self-contained text formatting and print statement

```
println(s"""First items are: ${xrdd.take(5).mkString(", ")}
And the mean is ${xrdd.reduce(_+_) / xrdd.count}""")
```

- `s"$var"` (String interpolation)
  allows complex code inside a String
- Multiline Strings (`"""`) help make this readable
- Separate mkString and other string formatting logic from main program logic (to reduce clutter)
- Keep printing and formatting-related code in a single place

Bring argument class fields to scope

```scala
// x is a
case class Movie(movieId:Int, title:String,
  genres:Array[String])
something.map{ x =>
  import x._
  movieId -> title
}
```

Tips: Changing data path and master

- Main program structure for running on a cluster plus testing locally
- Give -Dspark.master=local[2] in VM args in Eclipse, or command line replace with -Dspark.master=spark://ukko123.hpc.cs.helsinki.fi:7077

```
main(args: Array[String]) {
  val dataPath = args(0)
  val conf = new
    SparkConf().setAppName(getClass.getName)
  val sc = new SparkContext(conf)
  val dataRdd = sc.textFile(dataPath)
  //…
}
```

Thanks

- **These slides: http://is.gd/bigdataalgo**
- **Spark API:**http://spark.apache.org/docs/latest/api/scala/index.html
- **Eemil Lagerspetz Eemil.lagerspetz@cs.helsinki.fi**
- **IRC channel: #tkt-bdf**
- **After Thu 2015-04-02, this slideset will include converting the Pearson correlation algorithm to Spark**
- Contact us for more tips :)

Converting Pearson to Spark

- Start with the math (Wikipedia)

$$r = r_{xy} = \frac{\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^{n}(x_i - \bar{x})^2}\sqrt{\sum_{i=1}^{n}(y_i - \bar{y})^2}}$$

- Required components:
- Two datasets with equal length (n)
- Mean of both datasets (mx and my)
- Upper:
  - Product of difference from mean at each index i of both datasets
- Lower:
  - Standard deviation (sqrt of square difference sum) of each dataset separately, multiplied

Converting Pearson to Spark

- Mean is needed before calculating SD and the upper side, so do it separately:

$$r = r_{xy} = \frac{\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^{n}(x_i - \bar{x})^2}\sqrt{\sum_{i=1}^{n}(y_i - \bar{y})^2}}$$

```
val xdata = Array(0.0, 2.0, 4.0, 6.0, 8.0, 10.0)
val xrdd = sc.parallelize(xdata)
val ydata = Array(1.0, 3.0, 5.0, 7.0, 9.0, 9.0)
// Correct result for these is r=0.982
val yrdd = sc.parallelize(ydata)
val mx = xrdd.reduce(_+_) / xrdd.count // 5.0
val my = yrdd.reduce(_+_) / xrdd.count // 5.67
```

Converting Pearson to Spark

$$r = r_{xy} = \frac{\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^{n}(x_i - \bar{x})^2}\sqrt{\sum_{i=1}^{n}(y_i - \bar{y})^2}}$$

- Upper part needs us to combine both datasets by index. We do this with zip:

```
val both = xrdd.zip(yrdd)
val upper = both.map{ pair =>
  (pair._1 - mx)*(pair._2 - my)}.reduce(_+_) // 60.0
```

- The lower part has similar components, but the difference is squared before summing up:

```
val (lowerx, lowery) = both.map{ pair =>
    math.pow((pair._1 - mx), 2) -> math.pow((pair._2 – my), 2)}
    .reduce((a, b) => (a._1+b._1, a._2+b._2)) // 70.0, 53.33
```

```
val r = upper / (math.sqrt(lowerx) * math.sqrt(lowery))
// 0.9819805060619657
```

- Correct result.

Optimizing Pearson

- We ran three map-reduces (mean, upper, lower). What if we could do it in two? (mean, upper+lower)

```
val (upper, lowerx, lowery) = both.map{ pair =>
    val up = (pair._1 - mx)*(pair._2 - my)
    val lowx = math.pow((pair._1 - mx), 2)
    val lowy = math.pow((pair._2 - my), 2)
    (up, lowx, lowy)}.reduce{(a, b) =>
      (a._1+b._1, a._2+b._2, a._3+b._3)}
// 60.0, 70.0, 53.33
val r = upper / (math.sqrt(lowerx) * math.sqrt(lowery))
// 0.9819805060619657
```

Whole thing on one slide

```
val xrdd = sc.parallelize(Array(0.0, 2.0, 4.0, 6.0, 8.0, 10.0))
val yrdd = sc.parallelize(Array(1.0, 3.0, 5.0, 7.0, 9.0, 9.0))
val mx = xrdd.reduce(_+_) / xrdd.count // 5.0
val my = yrdd.reduce(_+_) / xrdd.count // 5.67
val (upper, lowerx, lowery) = xrdd.zip(yrdd).map{ pair =>
val up = (pair._1 - mx)*(pair._2 - my)
val lowx = math.pow((pair._1 - mx), 2)
val lowy = math.pow((pair._2 - my), 2)
(up, lowx, lowy)}.reduce{(a, b) =>  (a._1+b._1, a._2+b._2,
a._3+b._3)}
// 60.0, 70.0, 53.33
val r = upper / (math.sqrt(lowerx) * math.sqrt(lowery))
// 0.9819805060619657
```