Large-Scale Matrix Operations Using a Data Flow Engine

Matei Zaharia
Outline

Data flow vs. traditional network programming

Limitations of MapReduce

Spark computing engine

Matrix operations on Spark
Problem

Data growing faster than processing speeds

Only solution is to parallelize on large clusters

» Wide use in both enterprises and web industry
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?
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 (node not failed, but slow)
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 (node not failed, but slow)

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
MapReduce for Matrix Operations
Matrix-vector multiply
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
» Even HPC world is now concerned about resilience
Outline

Data flow vs. traditional network programming

Limitations of MapReduce

Spark computing engine

Matrix operations on Spark
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
  » No control of data partitioning across steps
Example: Iterative Apps

Commonly spend 90% of time doing I/O
Example: PageRank

Repeatedly multiply sparse matrix and vector

Requires repeatedly hashing together page adjacency lists and rank vector

Neighbors (id, edges)

Ranks (id, rank)

iteration 1
Example: PageRank

Repeatedly multiply sparse matrix and vector

Requires repeatedly hashing together page adjacency lists and rank vector
Example: PageRank

Repeatedly multiply sparse matrix and vector

Requires repeatedly hashing together page adjacency lists and rank vector

Neighbors (id, edges)

Ranks (id, rank)

iteration 1  iteration 2  iteration 3
Example: PageRank

Repeatedly multiply sparse matrix and vector

Requires repeatedly hashing together page adjacency lists and rank vector

Iteration 1 Iteration 2 Iteration 3

Neighbors (id, edges)

Ranks (id, rank)

Same file grouped over and over
Spark Programming Model

Extends MapReduce with primitives for efficient data sharing
  » “Resilient distributed datasets”

Open source in Apache Incubator
  » Growing community with 100+ contributors

APIs in Java, Scala & Python
Resilient Distributed Datasets (RDDs)

Collections of objects stored across a cluster
User-controlled partitioning & storage (memory, disk, ...)

Automatically rebuilt on failure

```
urls = spark.textFile("hdfs://...")
records = urls.map(lambda s: (s, 1))
counts = records.reduceByKey(lambda a, b: a + b)
bigCounts = counts.filter(lambda (url, cnt): cnt > 10)

bigCounts.cache()

bigCounts.filter(
    lambda (k,v): "news" in k).count()

bigCounts.join(otherPartitionedRDD)
```
Performance

K-Means Clustering

- Hadoop: 155 seconds
- Spark: 4.1 seconds

Logistic Regression

- Hadoop: 110 seconds
- Spark: 0.96 seconds

Time per Iteration (s)
Outline

Data flow vs. traditional network programming

Limitations of MapReduce

Spark computing engine

Matrix operations on Spark
PageRank

Using cache(), keep neighbors in RAM
Using partitioning, avoid repeated hashing
PageRank

Using cache(), keep neighbors in RAM

Using partitioning, avoid repeated hashing
PageRank

Using cache(), keep neighbors in RAM

Using partitioning, avoid repeated hashing
PageRank

Using cache(), keep neighbors in RAM

Using partitioning, avoid repeated hashing
PageRank

Using `cache()`, keep neighbors in RAM

Using partitioning, avoid repeated hashing
PageRank

Using cache(), keep neighbors in RAM

Using partitioning, avoid repeated hashing
PageRank

Using cache(), keep neighbors in RAM

Using partitioning, avoid repeated hashing
PageRank

Using cache(), keep neighbors in RAM

Using partitioning, avoid repeated hashing
PageRank

Using cache(), keep neighbors in RAM

Using partitioning, avoid repeated hashing
PageRank

Using cache(), keep neighbors in RAM

Using partitioning, avoid repeated hashing
PageRank Code

# RDD of (id, neighbors) pairs
links = spark.textFile(...).map(parsePage)
    .partitionBy(128).cache()

ranks = links.mapValues(lambda v: 1.0)  # RDD of (id, rank)

for i in range(ITERATIONS):
    ranks = links.join(ranks).flatMap(
        lambda (id, (links, rank)):
            [(d, rank/links.size) for d in links]
    ).reduceByKey(lambda a, b: a + b)
PageRank Results

- Hadoop: 171 seconds
- Basic Spark: 72 seconds
- Spark + Controlled Partitioning: 23 seconds
Alternating Least Squares

1. Start with random $A_1$, $B_1$
2. Solve for $A_2$ to minimize $\|R - A_2B_1^T\|$
3. Solve for $B_2$ to minimize $\|R - A_2B_2^T\|$
4. Repeat until convergence

$$R = A B^T$$
ALS on Spark

Joint work with Joey Gonzales, Virginia Smith

Cache 2 copies of $R$ in memory, one partitioned by rows and one by columns
Keep $A$ & $B$ partitioned in corresponding way
Operate on blocks to lower communication
ALS Results

- Mahout / Hadoop: 4208 seconds
- Spark (Scala): 481 seconds
- GraphLab (C++): 297 seconds
Benefit for Users

Same engine performs data extraction, model training and interactive queries

Separate engines

Spark

DFS read  parse  train  DFS write
DFS read  train  DFS write
DFS read  Java  DFS write

DFS read  parse  train  query

DFS
Other Projects on Spark

**MLlib**: built-in Spark library for ML
- Includes ALS, K-means||, various algorithms on SGD
- Frankin, Gonzales et al. [MLOSS ‘13]

**MLI**: Matlab-like language for writing apps
- Basic ALS in 35 lines of code
- Evan Sparks, Ameet Talwalkar et al. [ICDM ‘13]
Spark Community

100+ developers, 25+ companies contributing;
most active development community after Hadoop
Conclusion

Data flow engines are becoming an important platform for matrix algorithms.

Spark offers a simple programming model that greatly speeds these up.

More info: spark.incubator.apache.org