Distributed Systems

15-440 / 15-640
Fall 2017

Spark and Distributed ML
Topics Today

Motivation
  Going beyond Map/Reduce

Spark
  Data sharing and fault tolerance
  Distributed programming model

Distributed Machine Learning
  The scalability challenge
Distributed Data Processing

Map/Reduce (Hadoop) Framework:

Simplified data analysis on large, unreliable clusters
Highly expressive and scalable
Limitations of Map/Reduce I

Fault-tolerance: store input/output after every step on disk

While throughput scales with #servers, latency gets worse

Does not work for interactive data exploration
Limitations of Map/Reduce II

Expressiveness relies on iterating Map/Reduce steps

Iteration steps are small, but there are many

⇒ 90% spent on I/O to disks and over network
⇒ 10% spent on computing (using the CPUs)

Does not work for distributed machine learning
Limitations of Map/Reduce III

M/R abstraction not expressive enough

Explosion of specialized analytics systems

- Streaming analytics: 
  - Storm
  - Kafka

- Iterative ML algorithms: 
  - GraphLab

- Graph/social data: 
  - Giraph
  - Google Dremel

Learn all of them? Share data between them?
Topics Today

Motivation

Beyond Map/Reduce

Spark

Data sharing and fault tolerance

Distributed programming model

Distributed Machine Learning

The scalability challenge
Towards a New Unified Framework

Berkeley Extensions to M/R framework (⇒ Apache Spark)

1. In-memory computation and data-sharing
   - 10-100x faster than disks or network
   - Key problem: fault tolerance

2. Unified computation abstraction
   - Power of iterations ("local work + message passing")
   - Key problem: easy-of-use and generality
In-memory computation and data-sharing

How to build **fault-tolerance** and **efficient** system?

Traditional fault-tolerance in-memory storage systems

- Fine-grained update interface (databases, KV-stores, ..)
- Requires replicating data or logs across nodes

⇒ Expensive (10-100x slowdown)
Approach 1: RamCloud

Replicate in-memory objects 3x, on disks
Parallelize disk reads to speed-up recovery

Involves disks, again!

Approach 2: RDDs and Lineage

Resilient Distributed Datasets

- Provide interface based on coarse-grained operations
  - Map, group-by, filter, sample, ...

- Efficient fault recovery using lineage
  - Each operation is applied to many elements
  - Low each operation
  - Recompute lost partitions on failure

RDD Operations in Spark

1) Transformations: create new RDD from existing one
   - Map, filter, sample, groupByKey, sortByKey, union, join, cross

2) Actions: return value to caller
   - Count, sum, reduce, save, collect

3) Persist RDD to memory

⇒ Transformations are lazy: evaluation triggered by Action
Example 1: Log Mining

Parse error messages from logs, filter, and query

```scala
lines = spark.textFile("hdfs://...")
errors = lines.filter(_.startsWith("ERROR"))
messages = errors.map(_.split("\t")(2))
cachedMsgs = messages.persist()
cachedMsgs.filter(_.contains("foo")).count
.. cachedMsgs.filter(_.contains("bar")).count
```

Full-text search of Wikipedia in <1 sec (vs 20 sec for on-disk data).

Recall: fault recovery via lineage.
Example 2: Logistic Regression

Find a line (hyperplane) that separates some data points

```scala
var w = Vector.random(D-1)
for (i <- 1 to ITERATIONS) {
  val gradient = data.
    map(p => (1 / (1 + exp(-p.y*(w dot p.x))) - 1) * p.y * p.x)
    reduce(_+_)
  w -= gradient
}
```
RDD Consistency and Fault Recovery

RDDs are immutable datasets

- Deterministic functions of input
  - recreate any RDD any time
  - how to incorporate randomness?
- Simplifies consistency (caching, sharing, ..)
- Still need period RDD checkpoints

Design implications?

- High overhead: copying data (no mutate-in-place)
- Needs lots of memory (might not be able to run your workload)
Towards a New Unified Framework

Berkeley Extensions to M/R framework (⇒ Apache Spark)

1. In-memory computation and data-sharing
   - 10-100x faster than disks or network
   - Key problem: fault tolerance

2. New computation abstraction
   - Power of iterations ("local work + message passing")
   - Key problem: accessible and unified framework
BSP computation abstraction

- Surprising power of iterations
  - (e.g., iterative Map/Reduce)
- Explained by theory of bulk synchronous parallel (BSP) model

Theorem (Leslie Valiant, 1990):
“Any distributed system can be emulated as local work + message passing” (=BSP).

⇒ Spark implements BSP
Optimizing Communication Overhead

Communication overhead scales badly

e.g., for Netflix-like recommender systems

Solution: apply P2P ideas to Spark

- BitTorrent-like protocol
- Modified for datacenters:
  - 4MB blocks
  - full collaboration (no tit-for-tat)
  - no integrity checks (SHA1 is expensive)

Spark Pipeline and Scheduler

Supports DAGs of RDD operations

Pipelines functions within a stage

Partitioning/Cache-aware scheduling minimizes shuffles
Spark as a Uniform Framework

Pregel on Spark (Bagel)
⇒ “200 lines of Spark code”

Iterative MapReduce
⇒ “200 lines of Spark code”

Hive on Spark (Shark)
⇒ “5000 lines of code”

ML-lib and other distributed ML implementations
Limitations of Spark

Spark is not a good fit for (examples):

- Applications with fine-grained updates to shared state
- Storage for web applications
- Incremental web crawlers
- Data sets that don’t fit into memory
- If cannot be expressed as batch-analytics, i.e., apply same function uniformly over dataset
Topics Today

Motivation
Beyond Map/Reduce

Spark
Data sharing and fault tolerance
Distributed programming model

Distributed Machine Learning
The scalability challenge
Machine Learning

The ML hype

Enabled by huge leap in parallelization

Often: fast enough to build a powerful machine, lots of GPUs
Systems that can learn from data

Some ML systems drive significant revenue

Some ML systems benefit from humongous amount of data

Some ML systems outscale even powerful machines (GPUs et al)

A case for Distributed Machine Learning Platforms
arg max (θ) = Φ({x_i,y_i}^N; θ) + Ω(θ)
solved via iterative convergent algorithm
θ^{t+1} = g(θ^t, F(θ^t(Φ)))

data
parameters
parallelize this
Scaling Out Distributed Machine Learning

Often data/models fits into just a few nodes

Scale out for **performance**

100x speedup from 1000 machines

\[
\text{arg max } (\theta) = \Phi(\{x_i,y_i\}; \theta) + \Omega(\theta)
\]

solved via iterative convergent algorithm

\[
\theta^{t+1} = g(\theta^t, F(\theta^t(\Phi)))
\]

parallelize this

\[
\text{Iterations / Sec}
\]

\[
\text{Machine Count}
\]

ideal speedup

good speedup

pathetic speedup

100x speedup from 1000 machines
Distributed ML Scalability Challenge

Communication overhead
⇒ P2P, selective communication

Fundamental limitation in BSP model
Prevalence of stragglers
⇒ need new programming model?
Relaxing BSP Consistency

Exploit robustness of ML convergence

Accept slightly stale state

Bounded-delay BSP variant

From: Eric Xing, Strategies & Principles for Distributed Machine Learning, Allen AI, 2016
Bounded-delay BSP for Distributed ML

Bound stale state by $N$ steps:

$\Rightarrow$ $N$-bounded delay BSP

Graph showing computing and waiting times for different bounded delays:

- Y-axis: time (hour)
  - 1.8
  - 1.35
  - 0.9
  - 0.45
  - 0.0
- X-axis: bounded delay
  - 0
  - 1
  - 2
  - 4
  - 8
  - 16

Diagram illustrating 1-bounded delay and 2-bounded delay.
Many Challenges Remain

Trade-Off:

Stale state -> throughput (iter / sec)

Misleading design decisions:

Throughput not the right metric

Higher throughput, but less progress / iteration

General distributed ML systems

⇒ Very active field
Summary: Spark and Distributed ML

Spark and in-memory computing

- Motivation: overcome disk i/o bottleneck
- Challenges: fault tolerance and generality/expressiveness
- Key ideas: RDDs, the BSP model, P2P architecture

Distributed Machine Learning

- Distributed variants highly complex and not always faster!
- Challenges: communication overhead and stragglers
- Key ideas: P2P+selective communication, bounded-delay BSP