Distributed Systems

15-440/640

Fall 2018

17 – Fault-tolerant in-memory computation

Announcements

Continued midsemester feedback (Yuvraj & Daniel)
Thursday 4-5pm in GHC 4124
Please come and talk to us
if lettergrade < C-
if midterm < 60
if you just want more feedback

Daniel’s OH today: noon to 1pm

HW3 released yesterday
Cluster Computing (Review)

MapReduce (Hadoop) Framework:

Key features: **fault tolerance** and **high throughput**

⇒ Simplified data analysis on large, unreliable clusters

Can you think of limitations of the MapReduce framework?
Limitations of MapReduce I

Implementation: store input/output after every step on disk

Effect on response time?

- L1 cache reference: 1ns
- Main memory reference: 100ns
- SSD disk write: 350μs
- Same-datacenter RTT: 500μs

I/O penalty makes interactive data analysis impossible

From Jeff Dean’s Latency Numbers Every Programmer Should Know
Limitations of MapReduce II

Real-world applications require iterating MapReduce steps

Each iteration steps is small.

But: we need many iterations

⇒ 90% spent on I/O to disks and over network
⇒ 10% spent computing actual results

Does not work for iterative applications
⇒ distributed machine learning
Limitations of MapReduce III

MapReduce abstraction not expressive enough

Explosion of specialized analytics systems

- Streaming analytics:
  [STORM, Kafka]
- Iterative ML algorithms:
  [GraphLab, Pregel]
- Graph/social data:
  [Giraph, Google Dremel]

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

Motivation

Going beyond MapReduce

In-memory computation (Spark)

Data sharing and fault tolerance

Distributed programming model

Distributed Machine Learning

The scalability challenge
In-Memory Computation

Berkeley Extensions to Hadoop (⇒ Apache Spark)

Key idea: keep and share data sets in main memory

Much faster response time (in practice: 10x-100x)

- L1 cache reference: 1ns
- Main memory reference: 100ns
- SSD disk write: 350μs
- Same-datacenter RTT: 500μs

Why didn’t we do that in the first place? Problems?
In-memory computation and data-sharing

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

💡 Fault tolerance techniques from lectures so far?

Traditional fault-tolerance approaches

- Logging to persistent storage
- Replicating data across nodes (ideally: also to persistent storage)
- Checkpointing (checkpoints need to be stored persistently)

⇒ Expensive (10-100x slowdown)

🤔 Idea: do we need fine-grained updates to data?
Spark Approach: RDDs and Lineage

**Resilient Distributed Datasets**

- Limit update interface to coarse-grained operations
  - Map, group-by, filter, sample, ...
- Efficient fault recovery using **lineage**
  - Data is partitioned and each operation is applied to every partition
  - Individual operations are cheap
  - Recompute lost partitions on failure

RDDs are Immutable Objects

What’s immutability? ⇒ Object’s state can’t be modified after creation

Examples: e.g., lists/vectors in SML, strings in Java (remember 15-150/210/15-214?)

Why immutability?

- Enables lineage
  - Recreate any RDD any time
  - More strictly: RDDs need to be deterministic functions of input
- Simplifies consistency
  - Caching and sharing RDDs across Spark nodes
- Compatibility with storage interface (HDFS)
  - HDFS chunks are append only

How to use RDDs?
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 interactively.

```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
```

Recall: fault recovery via lineage

Full-text search of Wikipedia in <1 sec (vs 20 sec for on-disk data).
Example 2: Regression Algorithms

Find a line (plane) that separates some data points

```
var w = Vector.random(D-1)
for (i <- 1 to ITERATIONS) {
    val gradient =
        data.map("math equation").reduce(_ + _)
    w -= gradient
}
```
Apache Spark Deployment

Master server ("driver")
- Lineage and scheduling

Cluster manager (not part of Spark)
- Resource allocation
- Mesos, YARN, K8S

Worker nodes
- Executors isolate concurrent tasks
- Caches persist RDDs

https://spark.apache.org/docs/latest/cluster-overview.html
Spark Pipeline and Scheduler

Support for directed graphs of RDD operations

Automatic pipelining of functions within a stage

Partitioning/Cache-aware scheduling to minimizes shuffles

![Diagram of Spark Pipeline and Scheduler]

- **A:**
- **B:**
  - `groupBy`
- **C:**
- **D:**
  - `map`
- **E:**
- **F:**
  - `union`
- **G:**

= cached data partition
Spark Real World Challenges

RDD Lineage
- What if need lineage grows really large?
  - manual checkpointing on HDFS

RDDs Immutability
- Deterministic functions of input
  - how to incorporate randomness?

Other design implications?

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

Two Goals

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: ease-of-use and generality
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 approximately
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)

⇒ “500 lines of code”

ML-lib and other distributed ML implementations
Should You Always Use Spark?

Spark is not a good fit for

- Non-batch workloads
  - Applications with fine-grained updates to shared state
- Datasets that don’t fit into memory
- If you need high efficiency, SIMD/GPU

“Apache Spark does not provide out-of-the-box GPU integration.” (databricks.com)
Topics Today

Motivation

Going beyond MapReduce

In-memory computation (Spark)

Data sharing and fault tolerance

Distributed programming model

Distributed Machine Learning

Scaling out DML

Two challenges and current solution approaches

Open challenges
Machine Learning

The ML hype

Enabled by huge leap in parallelization

Often: fast enough to build a powerful machine, lots of GPUs
Is There a Case for Distributed ML?

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)

Ads make one case for **Distributed Machine Learning.**
What Do ML Algorithms look like?

We’ve already seen some examples:

- Page Rank
- Regression

Other examples: Bayes, K-means, NNs...

Common feature when computing these algorithms?

ML algorithms are **iterative** in nature

Three key challenges:

1) lots of data
2) lots of parameters
3) lots of iterations

---

Eric Xing, Strategies & Principles for Distributed Machine Learning, Allen AI, 2016
Scaling Out Distributed Machine Learning

10-100s nodes enough for data/model

Scale out for **throughput**
Goal: more iterations / sec

Best case: 100x speedup from 1000 machines

Worst case: 50% slowdown from 1000 machines

Can you think of reasons?
Challenge of Communication Overhead

Communication overhead scales badly with #machines (N)

e.g., for Netflix-like recommender systems

Challenge of Communication Overhead

Communication overhead scales badly with #machines (N)

e.g., for Netflix-like recommender systems

Solution approaches:

- **Centralized server**
  - 2N overhead (instead of N!)
  - Limited scalability of central server

- **P2P file sharing (BitTorrent-like)**
  - Higher network overhead
  - Better scalability
  - Needs to be data center aware

Challenge of Synchronization Overhead

BSP model:
- No computation during barrier
- No communication during computation

Fundamental limitation in BSP model
Constantly waiting for **stragglers**

Do we need a new programming model?
Relaxing BSP Consistency

Idea: nodes can accept slightly stale state

ML algorithms are robust
⇒ converge even with some stale state

How can we incorporate stale state into the BSP model?

From: Eric Xing, Strategies & Principles for Distributed Machine Learning, Allen AI, 2016
Opposite Extreme: No Synchronization

What if we fully remove BSP’s synchronization barriers?

Asynchronous communication:
● no communication at all, or
● communication at any time

Observation through experiments:
Iterative algorithms won’t converge

How can we get diverging algorithms under control?
Bounded-delay BSP for Distributed ML

Bound stale state by N steps:
⇒ N-bounded delay BSP

From: Li et al, Scaling Distributed Machine Learning with the Parameter Server OSDI 2014
Many Challenges Remain

Trade-Off:

Stale state -> throughput (iter / sec)

Misleading design decisions:

Higher throughput

Less progress / iteration

Many open challenges

Automatic model partitioning

How to schedule many parallel jobs on ML clusters

How to build a framework for interactive ML applications

⇒ Very active field of research
Many open challenges

Automatic model partitioning
e.g., how to partition natural graphs

How to schedule many parallel jobs on ML clusters
e.g., how many nodes should be assigned to each analysis job, given a set of speed-up functions?

How to build a framework for interactive ML applications
e.g., how to get latency of queries to distributed ML systems below 10s? 1? 1ms?
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
- Many more challenges remain
Overview of Cache Update Propagation Mechanisms

(Please read before next lecture if not covered on 10/30.)
Cache Update Propagation Techniques

Ideal World: One-Copy Semantics

- S1
- X (Master Copy)
- S2
- S3

write X

Caching Reality

- S1
- X (cached)
- S2
- write X

update propagation?

"cached" copies
Cache Update Propagation Techniques

1. Enforce Read-Only (Immutable Objects)
2. Broadcast Invalidations
3. Check on Use
4. Callbacks
5. TTLs (“Faith-based Caching”)

All of these approximate one-copy-semantics
- how little can you give up, and still remain scalable?
- how complex is the implementation?
2. Broadcast Invalidations

Every potential caching site notified on every update

- No check to verify caching site actually contains object
- Notification includes specific object being invalidated
- Effectively broadcast of address being modified
- At each cache site, next reference to object will cause a miss

Usage: e.g., in CDNs (next lecture)

+ Simple to implement
+ No race conditions (with blocking writes)

+ Wasted traffic if no readers
+ Limited scalability (in blocking implementation)
3. Check On Use

Reader checks master copy before each use

- conditional fetch, if cache copy stale
- has to be done at coarse granularity (e.g. entire file)
- otherwise every read is slowed down excessively

Usage: e.g., AFS-1, HTTP (Cache-control: must-revalidate)

- Simple to implement
- No server state (no need to know caching node)

- Wasted traffic if no updates
- Very slow if high latency
- High load
5. TTLs ("Faith-based Caching")

Assume cached data is valid for a while

- Check after timer expires: Time-to-Live (TTL) field
- No communication during trust (TTL) period

Usage: e.g., CDNs, DNS, HTTP (Cache-control: max-age=30)

+ Simple to implement
- No server state (no need to know caching node)

+ Use visible inconsistency
- Less efficient than callback-based schemes