CSC2/458 Parallel and Distributed Systems Distribute Computing – Other Programming Models

Sreepathi Pai

April 03, 2018

URCS

Outline

Abstractions for Distributed Computing

Spark's Abstractions

The Spark Runtime

Layering on top of Spark

Outline

Abstractions for Distributed Computing

Spark's Abstractions

The Spark Runtime

Layering on top of Spark

Abstractions for Shared Memory Programming

- Shared address space
  - One Address, One Value
- Shared memory
  - Disk
  - RAM
- Coherent Caches
  - Disk Cache (OS)
  - Processor Cache (Processor)
- Locking
  - Libraries
  - Hardware Atomics
- Resilience
  - ECC (Hardware)

Abstractions for Distributed Computing

- Distributed Name Space
  - e.g. ?
- Distributed Shared Memory
- Distributed File Systems
- Distributed Caching
  - e.g. memcached
- Distributed Concurrency Control
  - i.e. locking and consistency
- Data Distribution and Marshalling
  - e.g. `ntoh`, `hton`
- Distributed Execution
- Resilience
  - e.g. ?

Provided by most OS

- Sockets

Provided by MPI

- Distributed Name Space (Ranks)
- Send/Recv
- Communication Primitives
- Distributed Execution (SPMD)
- No:
  - distributed shared memory
  - distributed file system
  - caching
  - locking and consistency
  - resilience
  - marshalling

Erlang/Elixir

What is Erlang?

Maybe, now, we should ask what is Elixir?

What is Erlang? Part I

Verbatim from the Erlang FAQ: Introduction

- Erlang provides a simple and powerful model for error containment and fault tolerance (supervised processes).
- Concurrency and message passing are a fundamental to the language. Applications written in Erlang are often composed of hundreds or thousands of lightweight processes. Context switching between Erlang processes is typically one or two orders of magnitude cheaper than switching between threads in a C program.

What is Erlang? Part II

Verbatim from the Erlang FAQ: Introduction

- Writing applications which are made of parts which execute on different machines (i.e. distributed applications) is easy. Erlang's distribution mechanisms are transparent: programs need not be aware that they are distributed.
- The Erlang runtime environment (a virtual machine, much like the Java virtual machine) means that code compiled on one architecture runs anywhere. The runtime system also allows code in a running system to be updated without interrupting the program.

Hadoop

What is Hadoop?

What does Hadoop give us?

- Distributed File System
  - HDFS
- MapReduce programming model
  - distributed execution
  - marshalling
  - caching
- Resilience
  - Always writes to stable storage
  - Reruns failed jobs
- No need for:
  - distributed name space
  - distributed shared memory
  - concurrency control

Perspective

MapReduce

?

Erlang/Elixir

MPI

Sockets

Outline

Abstractions for Distributed Computing

Spark's Abstractions

The Spark Runtime

Layering on top of Spark

A Spark of an idea

Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing

Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, Ion Stoica
University of California, Berkeley

- Apache Spark
  - "fast and *general* engine for large-scale data processing."
- Translation: Write more than MapReduce programs *easily*
  - Compared to MPI
- 100x faster than Hadoop
  - "In-memory"
- You can write your own data processing engine on top of Spark

What Spark Provides

- Distributed File system
  - Reuses HDFS
- Distributed Execution
- Data Partitioning
- Marshalling
- Resilience
- Distributed Caching
  - No coherence required (or supported)
- No:
  - Distributed Concurrency Control (not supported)
  - Distributed Shared Memory (fine-grained)

Spark Programming Model: 10000ft overview

- Spark is a limited programming model
- Built on observation that:
  - "Many parallel applications naturally apply the same operations to multiple data items"
  - i.e. data-parallel model, e.g. SIMD, SPMD, etc.
- Provides a distributed *data structure*
  - Resilient Distributed Datasets (RDDs)
  - Like a huge table, but could be anything really
- Programs (you write) operate on RDDs in a *coarse-grained* fashion
  - They always operate on entire RDDs
  - I.e. on *all* elements in a RDD
  - Constrast with DSM which allows fine-grained accesses
- Java/Python/Scala

Resilient Distributed Datasets (RDD)

- A RDD is a
  - "read-only, partitioned set of records"
- Can only be built by "deterministic operations" (*transformations*) on:
  - data in stable storage
  - or other RDDs
- RDDs "remember" the operations that were used to create them
  - paper calls this "lineage"
- RDDs exist "lazily" in memory
  - the operations are only applied when needed (database lingo: materialized)
  - can be stored on disk too
- Why are these properties important?

Comparison to DSM

Aspect	RDDs	Distr. Shared Mem.
Reads	Coarse- or fine-grained	Fine-grained
Writes	Coarse-grained	Fine-grained
Consistency	Trivial (immutable)	Up to app / runtime
Fault recovery	Fine-grained and low-overhead using lineage	Requires checkpoints and program rollback
Straggler mitigation	Possible using backup tasks	Difficult
Work placement	Automatic based on data locality	Up to app (runtimes aim for transparency)
Behavior if not enough RAM	Similar to existing data flow systems	Poor performance (swapping?)

Table 1: Comparison of RDDs with distributed shared memory.

Spark Example

```
lines = spark.textFile("hdfs://...")
errors = lines.filter(_.startsWith("ERROR"))
errors.persist()

errors.count()
```

Spark Transformations

$map(f : \mathrm{T} \Rightarrow \mathrm{U})$	:	RDD[T] ⇒ RDD[U]
$filter(f : \mathrm{T} \Rightarrow \mathrm{Bool})$	:	RDD[T] ⇒ RDD[T]
$flatMap(f : \mathrm{T} \Rightarrow \mathrm{Seq[U]})$	:	RDD[T] ⇒ RDD[U]
$sample(fraction : \mathrm{Float})$	:	RDD[T] ⇒ RDD[T] (Deterministic sampling)
$groupByKey()$	:	RDD[(K, V)] ⇒ RDD[(K, Seq[V])]
$reduceByKey(f : (\mathrm{V}, \mathrm{V}) \Rightarrow \mathrm{V})$	:	RDD[(K, V)] ⇒ RDD[(K, V)]
$union()$	:	(RDD[T], RDD[T]) ⇒ RDD[T]
$join()$	:	(RDD[(K, V)], RDD[(K, W)]) ⇒ RDD[(K, (V, W))]
$cogroup()$	:	(RDD[(K, V)], RDD[(K, W)]) ⇒ RDD[(K, (Seq[V], Seq[W]))]
$crossProduct()$	:	(RDD[T], RDD[U]) ⇒ RDD[(T, U)]
$mapValues(f : \mathrm{V} \Rightarrow \mathrm{W})$	:	RDD[(K, V)] ⇒ RDD[(K, W)] (Preserves partitioning)
$sort(c : \mathrm{Comparator[K]})$	:	RDD[(K, V)] ⇒ RDD[(K, V)]
$partitionBy(p : \mathrm{Partitioner[K]})$	:	RDD[(K, V)] ⇒ RDD[(K, V)]

Spark Actions

$count()$	:	RDD[T] $\Rightarrow$ Long
$collect()$	:	RDD[T] $\Rightarrow$ Seq[T]
$reduce(f : (T, T) \Rightarrow T)$	:	RDD[T] $\Rightarrow$ T
$lookup(k : K)$	:	RDD[(K, V)] $\Rightarrow$ Seq[V] (On hash/range partitioned RDDs)
$save(path : String)$	:	Outputs RDD to a storage system, *e.g.*, HDFS

Outline

Abstractions for Distributed Computing

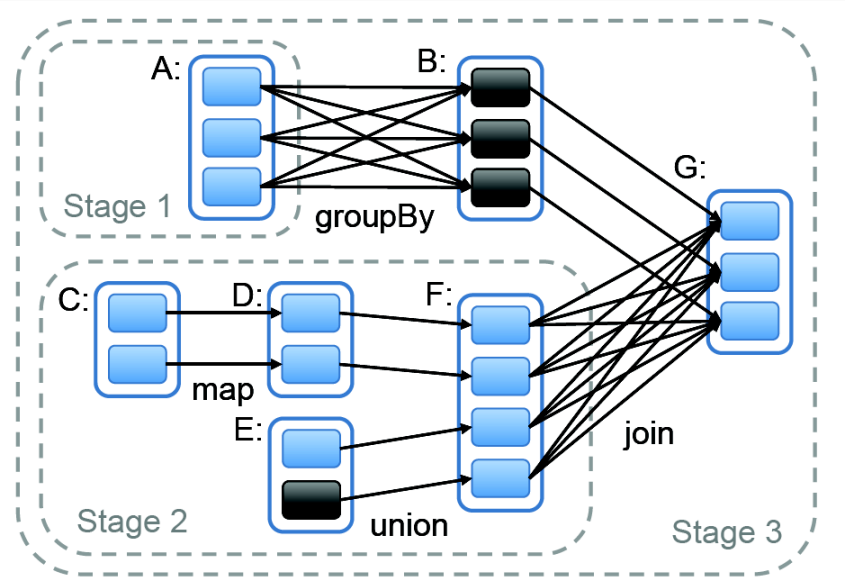
Spark's Abstractions

The Spark Runtime

Layering on top of Spark

Spark Scheduler

- Spark Programs are DAGS/dependence graphs
  - directed acyclic graphs
  - nodes are RDDs
  - edges are operations
- Scheduler executes in order of dependencies
  - prioritizes edges whose inputs already in memory



Scheduler Optimizations

- Narrow operations pipelined
  - i.e. loop coalescing

```
for(r in RDD)
   out1 = op1(r)

for(r in RDD)
   out2 = op2(r)
```

```
for(r in RDD) {
    out1 = op1(r)
    out2 = op2(r)
}
```

- Operations scheduled on machines based on locality
  - similar to "owner-computes"

Handling Failure

- Each RDD knows how to recreate itself
  - Ultimately from stable storage
- Recreation may use RDDs from other machines
  - "Wide" operations
  - e.g. join
- Or from the same machine
  - "Narrow" operations
  - e.g. map
- Can run in parallel
  - RDDs are immutable

Outline

Abstractions for Distributed Computing

Spark's Abstractions

The Spark Runtime

Layering on top of Spark

MapReduce

- RDD.map()
- RDD.reduceByKey()

DryadLINQ and SQL

- RDD.select()
- RDD.groupby()
- etc.

Pregel

- Google Pregel is a graph query engine
  - operates on graphs: vertices and edges
- Each operation is applied to a vertex in parallel
- Each vertex can send messages to other vertices
- Example Pregel in Spark:
  - RDD.flatMap()
  - RDD.join()

Your programming model here

- You need to implement transformations
- And actions
- Spark will take care of the rest...

Conclusion

- Spark provides a somewhat general distributed computing programming model
- Operations on immutable, partitioned datasets
- Partitioning, scheduling, marshalling, resilience, etc. for free
- Immensely popular programming model