Spark

Cluster Computing with Working Sets

Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, Ion Stoica

Motivation

- Cluster Computation has become the preferred way of computing on large amounts of data
- Existing solutions do not fare well for applications that **reuse** a particular data set across multiple parallel operations
  - Iterative algorithms
  - Interactive Applications

RDD

- Resilient Distributed Dataset
- Immutable collections of objects distributed across many machines
- Lineage: “Remembers” how it was created, so can rebuild itself if partition is lost
- Lazy evaluation of operations

Why not Shared Memory

- Distributed Shared Memory (DSM) allows for one giant address space across cluster
- Fine grained, aims to be invisible to programmer
- Difficult fault recovery
- Requires application to implement consistency
- In general expressivity are not worth performance tradeoffs

Creating RDDs

- Parallelize existing collection
- Transform an existing RDD
- From a file (eg hdfs file)
- Change persistence of existing RDD

Example Transformations/ Actions

Transformations	*map*(f : T ⇒ U)	RDD[T] ⇒ RDD[U]
	filter(f : T ⇒ Bool)	RDD[T] ⇒ RDD[T]
	flatMap(f : T ⇒ Seq[U])	RDD[T] ⇒ RDD[U]
	sample(*fraction* : Float)	RDD[T] ⇒ RDD[T] (Deterministic sampling)
	groupByKey()	RDD[(K, V)] ⇒ RDD[(K, Seq[V])]
	reduceByKey(f : (V, V) ⇒ 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 : V ⇒ W)	RDD[(K, V)] ⇒ RDD[(K, W)] (Preserves partitioning)
	sort(c : Comparator[K])	RDD[(K, V)] ⇒ RDD[(K, V)]
	partitionBy(p : Partitioner[K])	RDD[(K, V)] ⇒ RDD[(K, V)]
Actions	*count*()	RDD[T] ⇒ Long
	collect()	RDD[T] ⇒ Seq[T]
	reduce(f : (T, T) ⇒ T)	RDD[T] ⇒ T
	lookup(k : K)	RDD[(K, V)] ⇒ Seq[V] (On hash/range partitioned RDDs)
	save(*path* : String)	Outputs RDD to a storage system, *e.g.*, HDFS

Laziness

- All *transformations* are lazy (nothing happens when the *transformations* are called)
- When an *action* is executed (eg reduce, count, collect), the scheduler materializes the lineage DAG for the RDD and **executes** all the transformations.

Shared Variables

- Broadcast Variables
  - For large read only data
- Accumulators
  - Allow for an associative “add” operation

PageRank (example)

```
links = # RDD of (url, neighbors) pairs
ranks = # RDD of (url, rank) pairs

for i in range(NUM_ITERATIONS):
    def compute_contribs(pair):
        [url, [links, rank]] = pair  # split key-value pair
        return [(dest, rank/len(links)) for dest in links]

    contribs = links.join(ranks).flatMap(compute_contribs)
    ranks = contribs.reduceByKey(lambda x, y: x + y) \
                    .mapValues(lambda x: 0.15 + 0.85 * x)

ranks.saveAsTextFile(...)
```

Logistic Regression (example)

```
points = spark.textFile(...).map(parsePoint).cache()
w = numpy.random.ranf(size = D) # current separating plane
for i in range(ITERATIONS):
    gradient = points.map(
        lambda p: (1 / (1 + exp(-p.y*(w.dot(p.x)))) - 1) * p.y * p.x
    ).reduce(lambda a, b: a + b)
    w -= gradient
print "Final separating plane: %s" % w
```

Panacea?

- No.

What Spark can’t do (well)

- Small fine grained iterative updates to global state
- Fine grained debugging
- RDD’s limit expressivity somewhat
  - Still somewhat constrained by the map/reduce paradigm

Takeaways

- The RDD design for distributed objects showed that a little expressivity can be traded for performance and programmer productivity
- Brings large scale cluster computing one step closer to local computation
  - But we still aren't all the way there!