Communication Patterns
Outline

Life of a Spark Program
The Patterns
Shuffling
Broadcasting
Other programming languages
Life of a Spark Program
Life of a Spark Program

1) Create some input RDDs from external data or parallelize a collection in your driver program.

2) Lazily *transform* them to define new RDDs using transformations like `filter()` or `map()`

3) Ask Spark to `cache()` any intermediate RDDs that will need to be reused.

4) Launch *actions* such as `count()` and `collect()` to kick off a parallel computation, which is then optimized and executed by Spark.
Example Transformations

- map()
- flatMap()
- filter()
- mapPartitions()
- mapPartitionsWithIndex()
- sample()
- union()
- intersection()
- distinct()
- groupByKey()
- reduceByKey()
- sortByKey()
- coalesce()
- pipe()
- repartition()
- partitionBy()
- join()
- cogroup()
Example Actions

- reduce()
- collect()
- count()
- first()
- take()
- takeSample()
- saveToCassandra()
- takeOrdered()
- saveAsTextFile()
- saveAsSequenceFile()
- saveAsObjectFile()
- countByKey()
- foreach()
- ...

...
Communication Patterns

None:
   Map, Filter (embarrassingly parallel)

All-to-one:
   reduce

One-to-all:
   broadcast

All-to-all:
   reduceByKey, groupByKey, Join
Communication Patterns

Narrow Dependencies:
- map, filter
- union

Wide Dependencies:
- groupByKey
- join with inputs not co-partitioned
Shipping code to the cluster
RDD → Stages → Tasks

RDD Objects
- rdd1.join(rdd2)
- .groupBy(...) .filter(...)
- build operator DAG

DAG Scheduler
- split graph into stages of tasks
- submit each stage as ready

Task Scheduler
- launch tasks via cluster manager
- retry failed or straggling tasks
- launch each stage as ready

Worker
- execute tasks
- store and serve blocks
Example Stages

- Stage 1: A: RDD, B: groupBy
  -Cached partition
- Stage 2: C: map, D: filter, E: RDD, F: RDD
  -Lost partition
- Stage 3: join

= RDD
= cached partition
= lost partition
Talking to Cluster Manager

Manager can be:

YARN
Mesos
Spark Standalone
Shuffling (everyday)
How would you do a reduceByKey on a cluster?

Sort! Decades of research has given us algorithms such as TimSort
Shuffle

Sort: use advances in sorting single-machine memory-disk operations for all-to-all communication
Sorting

Distribute Timsort, which is already well-adapted to respecting disk vs memory

Sample points to find good boundaries

Each machines sorts locally and builds an index
## Sorting (shuffle)

<table>
<thead>
<tr>
<th></th>
<th>Hadoop World Record</th>
<th>Spark 100 TB *</th>
<th>Spark 1 PB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Size</td>
<td>102.5 TB</td>
<td>100 TB</td>
<td>1000 TB</td>
</tr>
<tr>
<td>Elapsed Time</td>
<td>72 mins</td>
<td>23 mins</td>
<td>234 mins</td>
</tr>
<tr>
<td># Nodes</td>
<td>2100</td>
<td>206</td>
<td>190</td>
</tr>
<tr>
<td># Cores</td>
<td>50400</td>
<td>6592</td>
<td>6080</td>
</tr>
<tr>
<td># Reducers</td>
<td>10,000</td>
<td>29,000</td>
<td>250,000</td>
</tr>
<tr>
<td>Rate</td>
<td>1.42 TB/min</td>
<td>4.27 TB/min</td>
<td>4.27 TB/min</td>
</tr>
<tr>
<td>Rate/node</td>
<td>0.67 GB/min</td>
<td>20.7 GB/min</td>
<td>22.5 GB/min</td>
</tr>
<tr>
<td>Sort Benchmark</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Daytona Rules</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Environment</td>
<td>dedicated data center</td>
<td>EC2 (i2.8xlarge)</td>
<td>EC2 (i2.8xlarge)</td>
</tr>
</tbody>
</table>

Distributed TimSort
Example Join

// Load RDD of (URL, name) pairs
val pageNames = sc.textFile("pages.\"txt\")\.'map(...)

// Load RDD of (URL, visit) pairs
val visits = sc.textFile("visits.\"txt\")\.'map(...)

val joined = visits.join(pageNames)

Shuffles both pageNames and visits over network
Broadcasting
Broadcasting

Often needed to propagate current guess for optimization variables to all machines

The exact wrong way to do it is with “one machines feeds all” – use bit-torrent instead

Needs log(p) rounds of communication
Bit-torrent Broadcast
Broadcast Rules

Create with SparkContext.broadcast(initialVal)

Access with .value inside tasks (first task on each node to use it fetches the value)

Cannot be modified after creation
Replicated Join

```scala
val pageNames = sc.textFile("pages.txt").map(...)
val pageMap = pageNames.collect().toMap()
val bc = sc.broadcast(pageMap)

val visits = sc.textFile("visits.txt").map(...)

val joined = visits.map(v => (v._1, (bc.value(v._1), v._2)))
```

- The `pageMap` is broadcasted using `sc.broadcast(pageMap)`.
- The `visits.txt` file is read and mapped.
- The `joined` variable is created by mapping the `visits` file, where each element is transformed into a tuple `(v._1, (bc.value(v._1), v._2)).

**Diagram Description:**
- The `pages.txt` file is read and mapped.
- The `master` node broadcasts the `pageMap`.
- The broadcasted `pageMap` is accessed via `bc.value`.
- The `visits.txt` file is read and mapped.
- The result is a list of tuples, where each tuple contains the page name and its associated visit count.

**Key Points:**
- The `pageMap` is broadcasted to all nodes once.
- The `bc.value` is used to access the broadcasted `pageMap` value.

**Diagram Notes:**
- Only sends `pageMap` to each node **once**.
Spark for Python (PySpark)
PySpark and Pipes

Spark core is written in Scala

PySpark calls existing scheduler, cache and networking layer (2K-line wrapper)

No changes to Python