Distributed Data-Parallel Programming

Principles of Functional Programming

Heather Miller

Data-Parallel Programming

So far:

- Data parallelism on a single multicore/multi-processor machine.
- Parallel collections as an implementation of this paradigm.

Today:

- Data parallelism in a *distributed setting*.
- Distributed collections abstraction from Apache Spark as an implementation of this paradigm.

Distribution

Distribution introduces important concerns beyond what we had to worry about when dealing with parallelism in the shared memory case:

- *Partial failure:* crash failures of a subset of the machines involved in a distributed computation.
- *Latency:* certain operations have a much higher latency than other operations due to network communication.

Distribution

Distribution introduces important concerns beyond what we had to worry about when dealing with parallelism in the shared memory case:

- *Partial failure:* crash failures of a subset of the machines involved in a distributed computation.
- *Latency:* certain operations have a much higher latency than other operations due to network communication.

Latency cannot be masked completely; it will be an important aspect that also impacts the *programming model*.

Important Latency Numbers

Latency numbers "every programmer should know:"[1]

```
L1 cache reference ..................... 0.5 ns
Branch mispredict ...................... 5 ns
L2 cache reference ..................... 7 ns
Mutex lock/unlock ...................... 25 ns
Main memory reference .................. 100 ns
Compress 1K bytes with Zippy ............. 3,000 ns  =   3 µs
Send 2K bytes over 1 Gbps network ....... 20,000 ns  =  20 µs
SSD random read ........................ 150,000 ns  = 150 µs
Read 1 MB sequentially from memory ..... 250,000 ns  = 250 µs
```

(Assuming ~1GB/sec SSD.)

[1] https://gist.github.com/hellerbarde/2843375

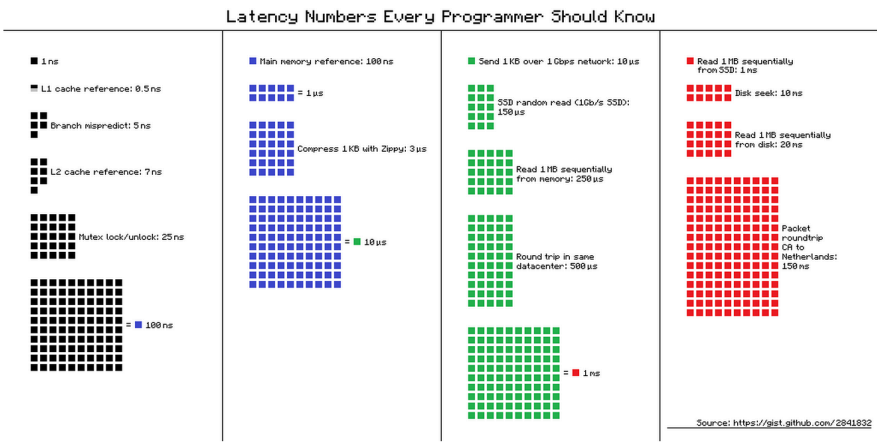
Important Latency Numbers

Latency numbers continued:

```
Round trip within same datacenter ...... 500,000 ns  = 0.5 ms
Read 1 MB sequentially from SSD* ..... 1,000,000 ns  =   1 ms
Disk seek ........................... 10,000,000 ns  =  10 ms
Read 1 MB sequentially from disk .... 20,000,000 ns  =  20 ms
Send packet CA->Netherlands->CA .... 150,000,000 ns  = 150 ms
```

(Assuming ~1GB/sec SSD.)

Latency Numbers Visually



Latency Numbers Intuitively

To get a better intuition about the *orders-of-magnitude differences* of these numbers, let's **humanize** these durations.

Method: multiply all these durations by a billion.

Then, we can map each latency number to a *human activity*.

Humanized Latency Numbers

Humanized durations grouped by magnitude:

Minute:

L1 cache reference	0.5 s	One heart beat (0.5 s)
Branch mispredict	5 s	Yawn
L2 cache reference	7 s	Long yawn
Mutex lock/unlock	25 s	Making a coffee

Hour:

Main memory reference	100 s	Brushing your teeth
Compress 1K bytes with Zippy	50 min	One episode of a TV show

Humanized Latency Numbers

Day:

Send 2K bytes over 1 Gbps network	5.5 hr	From lunch to end of work day

Week:

SSD random read	1.7 days	A normal weekend
Read 1 MB sequentially from memory	2.9 days	A long weekend
Round trip within same datacenter	5.8 days	A medium vacation
Read 1 MB sequentially from SSD	11.6 days	Waiting **for** almost 2 weeks **for** a delivery

More Humanized Latency Numbers

Year:

Disk seek	16.5 weeks	A semester in university
Read 1 MB sequentially from disk	7.8 months	Almost producing a **new** human being
The above 2 together	1 year	

Decade:

Send packet CA->Netherlands->CA	4.8 years	Average time it takes to complete a bachelor's degree

(Humanized) Durations: Shared Memory vs Distribution

Shared Memory

Seconds

L1 cache reference.....0.5s

L2 cache reference.....7s

Mutex lock/unlock.....25s

Minutes

Main memory reference.....1m 40s

Distributed

Days

Roundtrip within
same datacenter.....5.8 days

Years

Send packet
CA->Netherlands->CA....4.8 years

Data-Parallel to **Distributed** Data-Parallel

What does **distributed** data-parallel look like?

Data-Parallel to **Distributed** Data-Parallel

What does **distributed** data-parallel look like?

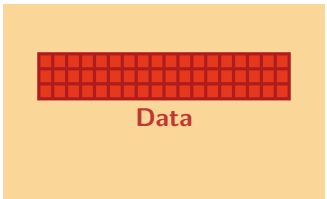
Shared memory:

Distributed:

Data-Parallel to **Distributed** Data-Parallel

What does **distributed** data-parallel look like?

Shared memory:

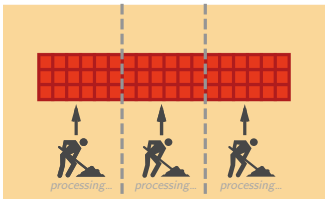


Distributed:

Data-Parallel to **Distributed** Data-Parallel

What does **distributed** data-parallel look like?

Shared memory:

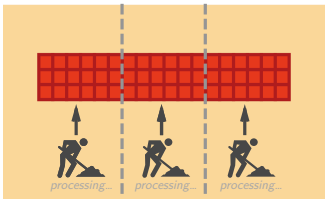


Distributed:

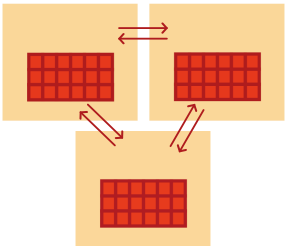
Data-Parallel to **Distributed** Data-Parallel

What does **distributed** data-parallel look like?

Shared memory:



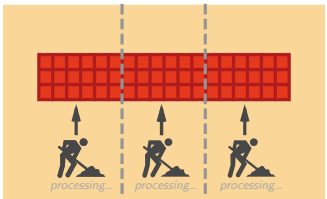
Distributed:



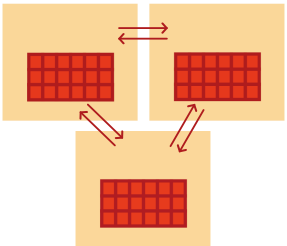
Data-Parallel to **Distributed** Data-Parallel

What does **distributed** data-parallel look like?

Shared memory:



Distributed:



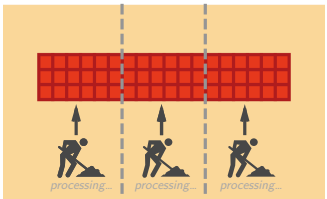
Shared memory case: Data-parallel programming model. Data partitioned in memory and operated upon in parallel.

Distributed case: Data-parallel programming model. Data partitioned between machines, network in between, operated upon in parallel.

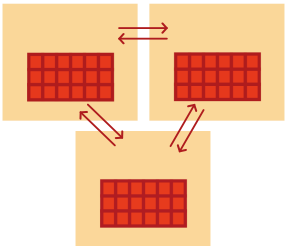
Data-Parallel to **Distributed** Data-Parallel

What does **distributed** data-parallel look like?

Shared memory:



Distributed:



Overall, most all properties we learned about related to shared memory data- parallel collections can be applied to their distributed counterparts.
E.g., watch out for non-associative reduction operations!

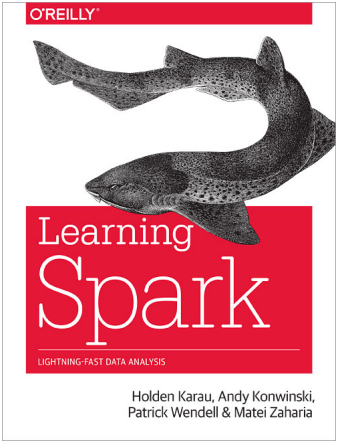
However, must now consider **latency** when using our model.

Apache Spark

Throughout this part of the course we will use the **Apache Spark** framework for distributed data-parallel programming.

Spark implements a distributed data parallel model called **Resilient Distributed Datasets (RDDs)**

Book



Learning Spark by Holden Karau, Andy Konwinski, Patrick Wendell & Matei Zaharia. O'Reilly, February 2015.

Resilient Distributed Datasets (RDDs)

RDDs look just like ***immutable*** sequential or parallel Scala collections.

Resilient Distributed Datasets (RDDs)

RDDs look just like ***immutable*** sequential or parallel Scala collections.

Combinators on Scala parallel/sequential collections:

```
map
flatMap
filter
reduce
fold
aggregate
```

Combinators on RDDs:

```
map
flatMap
filter
reduce
fold
aggregate
```

Resilient Distributed Datasets (RDDs)

While their signatures differ a bit, their semantics (macroscopically) are the same:

```
map[B](f: A => B): List[B] // Scala List
map[B](f: A => B): RDD[B] // Spark RDD

flatMap[B](f: A => TraversableOnce[B]): List[B] // Scala List
flatMap[B](f: A => TraversableOnce[B]): RDD[B] // Spark RDD

filter(pred: A => Boolean): List[A] // Scala List
filter(pred: A => Boolean): RDD[A] // Spark RDD
```

Resilient Distributed Datasets (RDDs)

While their signatures differ a bit, their semantics (macroscopically) are the same:

```
reduce(op: (A, A) => A): A // Scala List
reduce(op: (A, A) => A): A // Spark RDD

fold(z: A)(op: (A, A) => A): A // Scala List
fold(z: A)(op: (A, A) => A): A // Spark RDD

aggregate[B](z: => B)(seqop: (B, A) => B, combop: (B, B) => B): B // Scala
aggregate[B](z: B)(seqop: (B, A) => B, combop: (B, B) => B): B // Spark RDD
```

Resilient Distributed Datasets (RDDs)

Using RDDs in Spark feels a lot like normal Scala sequential/parallel collections, with the added knowledge that your data is distributed across several machines.

Example:

Given, `val encyclopedia: RDD[String]`, say we want to search all of `encyclopedia` for mentions of EPFL, and count the number of pages that mention EPFL.

Resilient Distributed Datasets (RDDs)

Using RDDs in Spark feels a lot like normal Scala sequential/parallel collections, with the added knowledge that your data is distributed across several machines.

Example:

Given, `val encyclopedia: RDD[String]`, say we want to search all of `encyclopedia` for mentions of EPFL, and count the number of pages that mention EPFL.

```
val result = encyclopedia.filter(page => page.contains("EPFL"))
                         .count()
```

Example: Word Count

The “Hello, World!” of programming with large-scale data.

```
// Create an RDD
val rdd = spark.textFile(”hdfs://...”)

val count = ???
```

Example: Word Count

The “Hello, World!” of programming with large-scale data.

```
// Create an RDD
val rdd = spark.textFile(”hdfs://...”)

val count = rdd.flatMap(line => line.split(” ”)) // separate lines into words
```

Example: Word Count

The “Hello, World!” of programming with large-scale data.

```
// Create an RDD
val rdd = spark.textFile(”hdfs://...”)

val count = rdd.flatMap(line => line.split(” ”)) // separate lines into words
               .map(word => (word, 1))           // include something to count
```

Example: Word Count

The “Hello, World!” of programming with large-scale data.

```
// Create an RDD
val rdd = spark.textFile(”hdfs://...”)

val count = rdd.flatMap(line => line.split(” ”)) // separate lines into words
               .map(word => (word, 1))           // include something to count
               .reduceByKey(_ + _)               // sum up the 1s in the pairs
```

That’s it.

Transformations and Actions

Recall ***transformers*** and ***accessors*** from Scala sequential and parallel collections.

Transformations and Actions

Recall ***transformers*** and ***accessors*** from Scala sequential and parallel collections.

Transformers. Return new collections as results. (Not single values.)
Examples: `map`, `filter`, `flatMap`, `groupBy`

```
map(f: A => B): Traversable[B]
```

Transformations and Actions

Recall ***transformers*** and ***accessors*** from Scala sequential and parallel collections.

Transformers. Return new collections as results. (Not single values.)
Examples: `map`, `filter`, `flatMap`, `groupBy`

```
map(f: A => B): Traversable[B]
```

Accessors: Return single values as results. (Not collections.)
Examples: `reduce`, `fold`, `aggregate`.

```
reduce(op: (A, A) => A): A
```

Transformations and Actions

Similarly, Spark defines ***transformations*** and ***actions*** on RDDs.

They seem similar to ~~transformers~~ and ~~accessors~~, but there are some important differences.

Transformations. Return new ~~collections~~ RDDs as results.

Actions. Compute a result based on an RDD, and either returned or saved to an external storage system (e.g., HDFS).

Transformations and Actions

Similarly, Spark defines ***transformations*** and ***actions*** on RDDs.

They seem similar to ~~transformers~~ and ~~accessors~~, but there are some important differences.

Transformations. Return new ~~collections~~ RDDs as results.

They are lazy, their result RDD is not immediately computed.

Actions. Compute a result based on an RDD, and either returned or saved to an external storage system (e.g., HDFS).

They are eager, their result is immediately computed.

Transformations and Actions

Similarly, Spark defines ***transformations*** and ***actions*** on RDDs.

They seem similar to ~~transformers~~ and ~~accessors~~, but there are some important differences.

Transformations. Return new ~~collections~~ RDDs as results.

They are lazy, their result RDD is not immediately computed.

Actions. Compute a result based on an RDD, and either returned or saved to an external storage system (e.g., HDFS).

They are eager, their result is immediately computed.

Laziness/eagerness is how we can limit network communication using the programming model.

Example

Consider the following simple example:

```
val largeList: List[String] = ...
val wordsRdd = sc.parallelize(largeList)
val lengthsRdd = wordsRdd.map(_.length)
```

What has happened on the cluster at this point?

Example

Consider the following simple example:

```
val largeList: List[String] = ...
val wordsRdd = sc.parallelize(largeList)
val lengthsRdd = wordsRdd.map(_.length)
```

What has happened on the cluster at this point?

Nothing. Execution of `map` (a transformation) is deferred.

To kick off the computation and wait for its result...

Example

Consider the following simple example:

```
val largeList: List[String] = ...
val wordsRdd = sc.parallelize(largeList)
val lengthsRdd = wordsRdd.map(_.length)
val totalChars = lengthsRdd.reduce(_ + _)
```

...we can add an action

Cluster Topology Matters

If you perform an action on an RDD, on what machine is its result "returned" to?

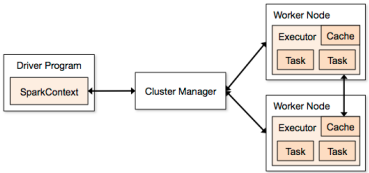
Example

```
val people: RDD[Person] = ...
val first10 = people.take(10)
```

Where will the `Array[Person]` representing `first10` end up?

Execution of Spark Programs

A Spark application is run using a set of processes on a cluster. All these processes are coordinated by the ***driver program***.



1. The driver program runs the Spark application, which creates a `SparkContext` upon start-up.
2. The `SparkContext` connects to a cluster manager (e.g., Mesos/YARN) which allocates resources.
3. Spark acquires ***executors*** on nodes in the cluster, which are processes that run computations and store data for your application.
4. Next, driver program sends your application code to the executors.
5. Finally, `SparkContext` sends ***tasks*** for the executors to run.

Cluster Topology Matters

If you perform an action on an RDD, on what machine is its result "returned" to?

Example

```
val people: RDD[Person] = ...
val first10 = people.take(10)
```

Where will the `Array[Person]` representing `first10` end up?

Cluster Topology Matters

If you perform an action on an RDD, on what machine is its result "returned" to?

Example

```
val people: RDD[Person] = ...
val first10 = people.take(10)
```

Where will the `Array[Person]` representing `first10` end up?

The driver program.

In general, executing an action involves communication between worker nodes and the node running the driver program.

Benefits of Laziness for Large-Scale Data

Spark computes RDDs the first time they are used in an action.

This helps when processing large amounts of data.

Example:

```
val lastYearsLogs: RDD[String] = ...
val firstLogsWithErrors = lastYearsLogs.filter(_.contains("ERROR")).take(10)
```

The execution of `filter` is deferred until the `take` action is applied.

Spark leverages this by analyzing and optimizing the **chain of operations** before executing it.

Spark will not compute intermediate RDDs. Instead, as soon as 10 elements of the filtered RDD have been computed, `firstLogsWithErrors` is done. At this point Spark stops working, saving time and space computing elements of the unused result of `filter`.

Caching and Persistence

By default, RDDs are recomputed each time you run an action on them. This can be expensive (in time) if you need to traverse a dataset more than once.

Spark allows you to control what is cached in memory.

Caching and Persistence

By default, RDDs are recomputed each time you run an action on them. This can be expensive (in time) if you need to traverse a dataset more than once.

Spark allows you to control what is cached in memory.

```
val lastYearsLogs: RDD[String] = ...
val logsWithErrors = lastYearsLogs.filter(_.contains("ERROR")).persist()
val firstLogsWithErrors = logsWithErrors.take(10)
```

Here, we *cache* `logsWithErrors` in memory.

After `firstLogsWithErrors` is computed, Spark will store the contents of `logsWithErrors` for faster access in future operations if we would like to reuse it.

Caching and Persistence

By default, RDDs are recomputed each time you run an action on them. This can be expensive (in time) if you need to traverse a dataset more than once.

Spark allows you to control what is cached in memory.

```
val lastYearsLogs: RDD[String] = ...
val logsWithErrors = lastYearsLogs.filter(_.contains("ERROR")).persist()
val firstLogsWithErrors = logsWithErrors.take(10)
val numErrors = logsWithErrors.count() // faster
```

Now, computing the `count` on `logsWithErrors` is much faster.

Caching and Persistence

Persistence levels. Other ways to control how Spark stores objects.

Level	*Space used*	*CPU time*	*In memory*	*On disk*
`MEMORY_ONLY`	High	Low	Y	N
`MEMORY_ONLY_SER`	Low	High	Y	N
`MEMORY_AND_DISK`*	High	Medium	Some	Some
`MEMORY_AND_DISK_SER`†	Low	High	Some	Some
`DISK_ONLY`	Low	High	N	Y

* Spills to disk if there is too much data to fit in memory

† Spills to disk if there is too much data to fit in memory. Stores serialized representation in memory.

Caching and Persistence

Persistence levels. Other ways to control how Spark stores objects.

Level	*Space used*	*CPU time*	*In memory*	*On disk*
`MEMORY_ONLY`	**High**	**Low**	**Y**	**N**
`MEMORY_ONLY_SER`	Low	High	Y	N
`MEMORY_AND_DISK`*	High	Medium	Some	Some
`MEMORY_AND_DISK_SER`†	Low	High	Some	Some
`DISK_ONLY`	Low	High	N	Y

Default

* Spills to disk if there is too much data to fit in memory

† Spills to disk if there is too much data to fit in memory. Stores serialized representation in memory.

Other Important RDD Transformations

Beyond the transformer-like combinators you may be familiar with from Scala collections, RDDs introduce a number of other important transformations.

`sample`	Sample a fraction fraction of the data, with or without replacement, using a given random number generator seed.
`union`	Return a new dataset that contains the union of the elements in the source dataset and the argument. Pseudo-set operations (duplicates remain).
`intersection`	Return a new RDD that contains the intersection of elements in the source dataset and the argument. Pseudo-set operations (duplicates remain).

Other Important RDD Transformations (2)

Beyond the transformer-like combinators you may be familiar with from Scala collections, RDDs introduce a number of other important transformations.

`distinct`	Return a new dataset that contains the distinct elements of the source dataset.
`coalesce`	Decrease the number of partitions in the RDD to numPartitions. Useful for running operations more efficiently after filtering down a large dataset.
`repartition`	Reshuffle the data in the RDD randomly to create either more or fewer partitions and balance it across them. This always shuffles all data over the network.

Other Important RDD Actions

RDDs also contain other important actions which are useful when dealing with distributed data.

`collect`	Return all the elements of the dataset as an array at the driver program. This is usually useful after a filter or other operation that returns a sufficiently small subset of the data.
`count`	Return the number of elements in the dataset.
`foreach`	Run a function func on each element of the dataset. This is usually done for side effects such as interacting with external storage systems.
`saveAsTextFile`	Write the elements of the dataset as a text file (or set of text files) in a given directory in the local filesystem, HDFS or any other Hadoop-supported file system. Spark will call toString on each element to convert it to a line of text in the file.

Pair RDDs

Often when working with distributed data, it's useful to organize data into **key-value pairs**. In Spark, these are Pair RDDs.

Useful because: Pair RDDs allow you to act on each key in parallel or regroup data across the network.

Spark provides powerful extension methods for RDDs containing pairs (*e.g.*, `RDD[(K, V)]`). Some of the most important extension methods are:

```
def groupByKey(): RDD[(K, Iterable[V])]
def reduceByKey(func: (V, V) => V): RDD[(K, V)]
def join[W](other: RDD[(K, W)]): RDD[(K, (V, W))]
```

Depending on the operation, data in an RDD may have to be **shuffled** among worker nodes, using worker-worker communication.

This is often the case for many operations Pair RDDs!