Introduction to Spark

Kenny Lu
School of Information Technology
Nanyang Polytechnic

October 10, 2017

Learning Outcomes

- Discuss and identify the differences between Parallel Programming and Concurrent Programming during system design and analysis
- Discuss and Identify the differences between data parallelism and task parallelism during system design and analysis.
- Understand and conceptualize data paralellism using MapReduce
- Comprehend and develop script in Spark for data transformation

What is a parallel program?

What is a parallel program?

A parallel program is one that uses a *multiplicity of computational hardware* (e.g., several processor cores or several server nodes) to *perform a computation more quickly*. The aim is to arrive at the answer earlier, by delegating different parts of the computation to different processors that execute at the same time.

Parallelism vs Concurrency

People often compare and confuse parallelism with concurrency.

What is a concurrent program?

By contrast, concurrency is *a program-structuring technique* in which there are multiple threads of control. Conceptually, the threads of control execute "at the same time"; that is, the user sees *their effects interleaved*. Whether they actually execute at the same time or not is an implementation detail; a concurrent program can execute on a single processor through interleaved execution or on multiple physical processors.

Parallelism vs Concurrency

	Parallelism	Concurrency
Area of Focus	Efficiency	Structural and Modularity
Number of Goals	One	One or more
Program Semantic	Deterministic	Non-deterministic
Single processor Exec	Sequential	Interleaving

Parallelism vs Concurrency

Examples of Parallelism

- A sodoku solver uses multiple CPU cores
- A parallelized database query that retrieves and aggregates records from a cluster of replica of database.
- A K-means analyses running over a Hadoop cluster

Examples of Concurrency

- A web application that handles multiple clients HTTP requests and interacting with the databases.
- A User Interface of the a mobile phone handles user's touch screen input and exchanging data via the 4G network

Different Types of Hardware Parallelism

- Single processor, e.g.
  - Bit-level parallelism
  - Instruction Pipelining
- GPU, e.g.
  - Matrix operations parallelisation
- Multiple processors in a single machine
  - Shared Memory
    - Multi-core computer executes multi-threaded program. Memory are shared among different threads.
  - Distributed Memory
    - Multi-GPU parallelisation.
- Multiple hosts (machines)
  - A grid computing
  - MapReduce cluster, Hadoop

Different Types of Software Parallelism

Software Parallelism

- Task parallelism (Dataflow parallelism)
- Data parallelism

Different Types of Software Parallelism

Task parallelism

```
def cook_beef_bolognese(beef,spaghetti) = {
  val sauce = cook_sauce(beef) // task A
  val pasta = cook_pasta(spaghetti) //  task B
  mix(sauce,pasta)
}
```

Note that task A and task B can be executed in parallel, e.g. `cook_sauce(beef)` is executed in Core 1, `cook_pasta(spaghetti)` is executed in Core 2.

Different Types of Software Parallelism

Data parallelism

```
def factorial(n) = ...
def main() {
   val inputs = List( 10, 100, 200, ...)
   val results = List()
   for ( i <- inputs )
   {
       results.append(factorial(i))
   }
   results
}
```

Different Types of Software Parallelism

Data parallelism

```
def factorial(n) = ...
def main() {
   val inputs = List( 10, 100, 200, ...)
  // task C
   val results = inputs.map( i => factorial( i ) )
   results
}
```

Note that each task C can be executed in parallel, e.g. `factorial(10)` is executed in Core 1, `factorial(100)` is executed in Core 2, `factorial(200)` is executed in Core 3, ...

Different Types of Software Parallelism

In most of situations, data parallelism is more scalable than task parallelism.

MapReduce and Google

- MapReduce was first introduced and developed by the development in Google [**?**].
- MapReduce is a formed of data parallelism.
- MapReduce is popularized by Hadoop.

MapReduce's Origin

- MapReduce was inspired by two combinators coming from Functional Programming world, i.e. `map` and `reduce` (some times `reduce` is named as `fold`)
- MapReduce exploits the concept of purity fro FP world to achieve parallelism.

map

In FP languages, `map(f,l)` takes two formal arguments, a *higher order* function `f` and a list `l`, and applies `f` to every element in `l`. For instance, let `incr(x)` be a function that returns the result of adding 1 to `x`.

incr

```
def incr(x) = x + 1
```

We apply it with `map` as follows

`map` in action

```
map(incr, [1,2,3,4])
```

evaluates to

```
[incr(1),incr(2),incr(3),incr(4)]
```

yields

```
[2,3,4,5]
```

Given that `f` is a *pure* function, (i.e. it does not modify its external state when it is executed,) it is guaranteed that `map(f,l)` can be parallelized by applying `f` to every element in `l` in parallel.
For instance, assuming we have processors A, B, C and D.

map in parallelized mode

```
map(incr, [1,2,3,4])
```

evaluates to

```
[incr(1),incr(2),incr(3),incr(4)]
```

yields

```
[2,3,4,5]
```


More on Pure function

A full defintiion of Pure function from Wikipedia.
A function may be described as a pure function if both these statements about the function hold:

1. The function always evaluates the same result value given the same argument value(s). The function result value cannot depend on any hidden information or state that may change as program execution proceeds or between different executions of the program, nor can it depend on any external input from I/O devices.
2. Evaluation of the result does not cause any semantically observable side effect or output, such as mutation of mutable objects or output to I/O devices

reduce

`reduce(f,l)` takes a function `f` and a list `l`, it aggregates the elemeents in `l` with `f`.
For instance, let `add(x,y)` be a function that returns the result of adding `x` to `y`.

add

```
def add(x,y) = x + y
```

We apply it with `reduce` as follows

`reduce` in action

```
reduce(add, [2,3,4,5])
```

evaluates to

```
2+3+4+5
```

yields

```
14
```


reduce

Given `f` is pure and *associative*, (`f` is associative iff `f(x,f(y,z)) == f(f(x,y),z)`), it is guaranteed that `reduce(f,l)` can be parallelized by partitioning elements in `l` into segments, aggregating each segment with `f` and aggregating the segment results into the final result.

For instance, assuming we have processors A and B.

reduce in parallelized mode

```
reduce(add, [2,3,4,5])
```

evaluates to

2+3+4+5

yields

5+9

yields 14.

Tweaking the partitions in `reduce`

Suppose instead of summing up all integers in the `reduce` step, we would like to sum up all the even numbers and odd numbers separately.

For instance, assuming we have processors A, B, C and D.

`map` in parallelized mode with partition keys

```
map(incr, [1,2,3,4])
```

evaluates to

```
[(0,incr(1)),(1,incr(2)),(0,incr(3)),(1,incr(4))]
```

yields

```
[(0,2),(1,3),(0,4),(1,5)]
```

Note that there are two possible values appearing as the first component of the pairs. `0` indicates that the following value is an even number and `1` denotes the following number is an odd

Tweaking the partitions in `reduce`

Now we can split the intermediate results into two different lists/arrays based on the partition ids (either `0` or `1`).

```
(0,[2,4])
```

and

```
(1,[3,5])
```

Tweaking the partitions in `reduce`

The two partitions can be processed as two independent `reduce` tasks.

`reduce` the even numbers in processor A

```
(0, reduce([2,4]))
```

and

`reduce` the odd numbers in processor B

```
(1, reduce([3,5]))
```

If we need to compute the final sum, we can run `reduce` over the results from the two tasks.

MapReduce in Hadoop

Two basic components in MapReduce

- Mapper
- Reducer

Mapper

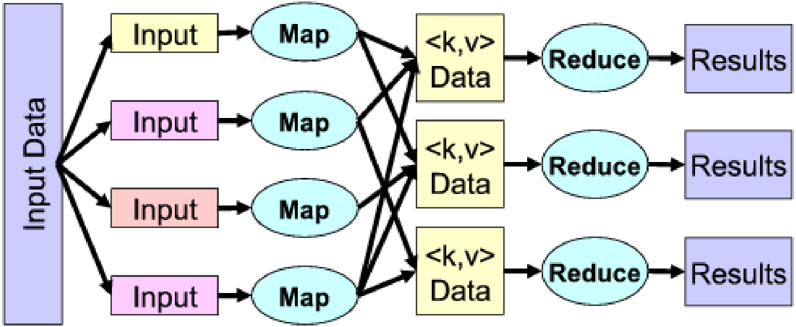
- Input: A list/array of key-value pairs.
- Output: A list/array of key-value pairs.

MapReduce in Hadoop

Reducer

- Input: A key and a list/array of values
- Output: A key and a value

MapReduce in Hadoop



An Example

WordCount Not using Hadoop

```
val counts = new Map()
val lines = scala.io.Source.fromFile("file.txt").mkString

for (line <- lines) {
    words = line.split(" ")
    for (word <- words) {
        counts.get(word) match
        { case None => counts.add(word,1)
          case Some(count) => counts.update(word,count + 1)
        }
    }
}
for ( (word,count) <- counts.iterator() ) {
    println(s"$word \t $count")
}
```

An Example

Let's say the input file is as follows

```
hey diddle diddle
the cat and the fiddle
the cow jumped over the
 moon
the little dog laughed
to see such sport
and the dish ran away
 with the spoon
```

An Example

We expect the output as

```
hey           1
diddle           2
the           7
cat           1
and           2
fiddle           1
cow           1
jumped           1
over           1
moon           1
little           1
dog           1
laughed           1
to           1
see           1
such           1
sport           1
dish           1
ran           1
away           1
with           1
spoon           1
```

An Example

The Mapper for WordCount

```
class WordCountMapper extends Mapper[Object,Text,Text,IntWritable] {
  val one = new IntWritable(1)
  val word = new Text

  override
  def map(key:Object, value:Text, context:
        Mapper[Object,Text,Text,IntWritable]#Context) = {
    for (t <-  value.toString().split("\\s")) {
      word.set(t)
      context.write(word, one)
    }
  }
}
```

An Example

The Reducer for WordCount

```
class WordCountReducer extends Reducer[Text,IntWritable,Text,IntWritable] {
  override
  def reduce(key:Text, values:java.lang.Iterable[IntWritable],
        context:Reducer[Text,IntWritable,Text,IntWritable]#Context) = {
    val sum = values.foldLeft(0) { (t,i) => t + i.get }
    context.write(key, new IntWritable(sum))
  }
}
```

An Example

Main for WordCount

```
object WordCount {

  def main(args:Array[String]):Int = {
    val conf = new Configuration()
    val otherArgs = new GenericOptionsParser(conf, args).getRemainingArgs
    if (otherArgs.length != 2) {
      println("Usage: wordcount <in> <out>")
      return 2
    }
    val job = Job.getInstance(conf, "wordcount");
    job.setJarByClass(classOf[WordCountMapper])
    job.setMapperClass(classOf[WordCountMapper])
    job.setCombinerClass(classOf[WordCountReducer])
    job.setReducerClass(classOf[WordCountReducer])
    job.setOutputKeyClass(classOf[Text])
    job.setOutputValueClass(classOf[IntWritable])
    FileInputFormat.addInputPath(job, new Path(args(0)))
    FileOutputFormat.setOutputPath(job, new Path((args(1))))
    if (job.waitForCompletion(true)) 0 else 1
  }
}
// yarn jar yourJar.jar WordCount /input/ /output/
```

An Example

Let's say the input file is as follows

```
hey diddle diddle
the cat and the fiddle
the cow jumped over the
 moon
the little dog laughed
to see such sport
and the dish ran away
 with the spoon
```

An Example

The mappers go through line by line

```
[("hey",1), ("diddle",1), ("diddle",1)]
the cat and the fiddle
the cow jumped over the
 moon
the little dog laughed
to see such sport
and the dish ran away
 with the spoon
```

An Example

The mappers go through line by line

```
[("hey",1), ("diddle",1), ("diddle",1)]
[("the",1), ("cat",1), ("and",1), ("the",1), ("fiddle", 1)]
the cow jumped over the
 moon
the little dog laughed
to see such sport
and the dish ran away with
 the spoon
```

An Example

The mappers go through line by line

```
[("hey",1), ("diddle",1), ("diddle",1)]
[("the",1), ("cat",1), ("and",1), ("the",1), ("fiddle", 1)]
[("the",1), ("cow",1), ("jumped",1), ("over",1), ("the", 1),
 ("moon", 1)]
[("the",1), ("little",1), ("dog",1), ("laughed", 1)]
[("to",1), ("see",1), ("such",1), ("sport",1)]
[("and",1), ("the",1), ("dish",1), ("ran",1), ("away",1),
 ("with",1), ("the",1), ("spoon",1)]
```

An Example

The output from the mappers are grouped by keys

```
[("hey",[1]),
 ("diddle",[1,1]),
 ("the",[1,1,1,1,1,1,1]),
 ("cat",[1]),
 ("and",[1,1]),
 ("fiddle", [1]),
 ("cow",[1]),
 ("jumped",[1]),
 ("over",[1]),
 ("moon", [1]),
 ("little",[1]),
 ("dog",[1]),
 ("laughed", [1]),
 ("to",[1]),
 ("see",[1]),
 ("such",[1]),
 ("sport",[1])
 ("dish",[1]),
 ("ran",[1]),
 ("away",[1]),
 ("with",[1]),
 ("spoon",[1])]
```

An Example

The reducers sum up the values for each key

```
[("hey",1),
 ("diddle",2),
 ("the",7),
 ("cat",1),
 ("and",2),
 ("fiddle", 1),
 ("cow",1),
 ("jumped",1),
 ("over",1),
 ("moon", 1),
 ("little",1),
 ("dog",1),
 ("laughed", 1),
 ("to",1),
 ("see",1),
 ("such",1),
 ("sport",1)
 ("dish",1),
 ("ran",1),
 ("away",1),
 ("with",1),
 ("spoon",1)]
```

What's wrong with Hadoop?

- All the mapper and reducers are communicating via the HDFS
- Reducers tend to be the bottle neck and its loads hardly re-distribute!

What is Spark?

- A distrbute cluster computing system favoring in-memory computation.
- It was developed intially for batch processing computation like MapReduce

Why Spark?

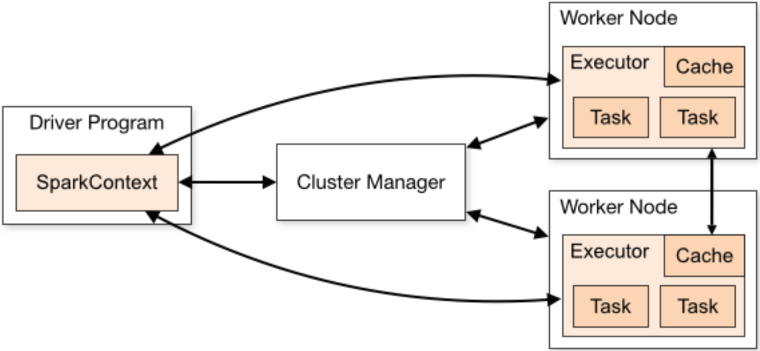
What's wrong with MapReduce?

- it was designed for moderate CPU and low memory systems.
- it relies on disk I/O operations at each intermediate steps.
- Its performance is capped by the disk I/O performance, and symmetric distribution of the Reduce jobs.

Spark comes in assuming our machines are in general more powerful, and RAMs are cheaper.

- it favors in memory computations. Data are loaded from disk and stay in memory as long as possible.
- it uses resillent distributed datasets (RDD) as the abstract data collections.
- it performs better than MapReduce if we have sufficient RAM in the cluster.

Spark Architecture



A SparkContext is an interface between the Spark Driver Program (application) and the Spark runtime-system

WordCount Example in Spark

Wordcount in Scala

```
val lines = sc.textFile("hdfs://127.0.0.1:9000/input/")
val counts = lines.flatMap(line => line.split(" "))
    .map(word => (word, 1))
    .reduceByKey(_ + _)
counts.saveAsTextFile("hdfs://127.0.0.1:9000/output/")
```

Wordcount in Python

```
lines = sc.textFile("hdfs://127.0.0.1:9000/input/")
counts = lines.flatMap(lambda x: x.split(' ')) \
              .map(lambda x: (x, 1)) \
              .reduceByKey(add)
couts.saveAsTextFile("hdfs://127.0.0.1:9000/output/")
```

sc denotes SparkContext

WordCount Example in Spark

Wordcount in Scala

```
val lines:RDD[String] =
    sc.textFile("hdfs://127.0.0.1:9000/input/")
val counts:RDD[(String,Long)] =
    lines.flatMap(line => line.split(" "))
    .map(word => (word, 1))
    .reduceByKey(_ + _)
counts.saveAsTextFile("hdfs://127.0.0.1:9000/output/")
```

Recall in Scala `List(1,2,3).map( v => v + 1)` yields `List(2,3,4)`
and `List(List(1),List(2),List(3)).flatMap( l => l )` yields `List(1,2,3)`
An RDD can be seen as a distributed list.

Resilient Distributed Dataset

- RDD is an abstraction over a collection of data set being distributed and partitioned across a cluster of worker machines, mostly in memory.
- Programmers are not required to manage or to coordinate that distributed and partitioned. RDD is fault tolerant.
- RDDs are initialized and managed by the SparkContext.

RDD transformations are pure

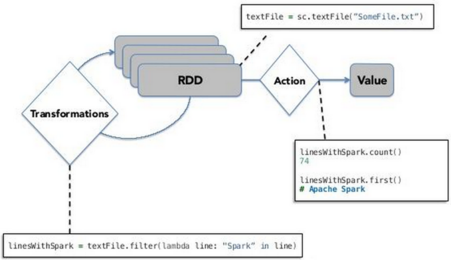


Image adapted from `http://www.hadooptpoint.com`

RDD transformations are pure

Let `r` denotes an RDD,

- `r.map(f)` and `r.flatMap(f)` applies `f` to elements in `r`.
- `r.filter(f)` filters away elements `x` in `r` which `f(x)` yields `false`.
- assuming `r` is a collection of key-value pairs, `r.reduceByKey(f)` will shuffle the pairs and group them by keys. The values grouped under the same key will be reduced by `f`. Data locality is exploit when possible.

RDD transformations are lazy

- Computations do not take place unless the results are needed.
- In memory cache are explicitly created.

Wordcount in Scala

```
val lines:RDD[String] =
    sc.textFile("hdfs://127.0.0.1:9000/input/")
val counts:RDD[(String,Long)] =
    lines.flatMap(line => line.split(" "))
    .map(word => (word, 1))
    .reduceByKey(_ + _)
counts.persist() // caching
counts.saveAsTextFile("hdfs://127.0.0.1:9000/output/")
val somethingelse = counts.map( ... )
```

RDD transformations are resilient to node failure

Since computations are pure, hence they are deterministic. Final results and intermediate results can be always recomputed.

How to run it?

First start the cluster

```
$ /opt/spark-1.4.1-bin-hadoop2.6/sbin/start-all.sh
```

Run it in the REPL

```
$ /opt/spark-1.4.1-bin-hadoop2.6/bin/spark-shell
 scala> :load Wordcount.scala
```

Or we can type the code in line by line.

Submit to the cluster

Scala

```
$ /opt/spark-1.4.1-bin-hadoop2.6/bin/spark-submit
 Wordcount.jar
```

Python

```
$ /opt/spark-1.4.1-bin-hadoop2.6/bin/spark-submit
wordcount.py
```

It supports R too.