Lab 13: First Steps with Apache Spark

Instructor: Vivek Sarkar, Eric Allen

Course wiki : `https://wiki.rice.edu/confluence/display/PARPROG/COMP322`

Staff Email : comp322-staff@mailman.rice.edu

Importants tips and links

edX site : `https://edge.edx.org/courses/RiceX/COMP322/1T2014R`

Piazza site : `https://piazza.com/rice/spring2015/comp322/home`

Java 8 Download : `https://jdk8.java.net/download.html`

Maven Download : `http://maven.apache.org/download.cgi`

IntelliJ IDEA : `http://www.jetbrains.com/idea/download/`

HJ-lib Jar File : `http://www.cs.rice.edu/~vs3/hjlib/code/maven-repo/edu/rice/hjlib-cooperative/0.1.5-SNAPSHOT/hjlib-cooperative-0.1.5-SNAPSHOT.jar`

HJ-lib API Documentation : `https://wiki.rice.edu/confluence/display/PARPROG/API+Documentation`

HelloWorld Project : `https://wiki.rice.edu/confluence/pages/viewpage.action?pageId=14433124`

1 Acknowledgements

This lab presents the Spark implementation of word count described at `http://spark.apache.org`.

2 Overview

The purpose of this lab is to walk through an installation of Apache Spark, and test out two examples of Apache Spark applications.

Although Apache Spark is best applied to distributed computation, it can be be run in "local mode," where it will simply make use of the available cores on your computer. Using local mode, we can view Spark as another viable model of multicore parallel computing.

For the purposes of this lab, we will run in local mode. However, the commands you will execute are identical to what you would use to run a distributed computation on a cluster with Spark installed.

NOTE: There is no Maven repository for this lab!

3 Installing Spark

To install Spark on your computer, perform the following steps:

- First ensure you have a JVM installed on your machine by typing `java -version` at a command line
- Go to `http://spark.apache.org` and click on "Download"

- Select release 1.3.0
- Select "Prebuilt for Hadoop 2.4 and later"
- Select "Direct download"
- Click on the Download Spark link

4 Unpacking Spark

- In your home directory, create a subdirectory `spark` in which to store Spark
- Move the downloaded file into this directory
- Unpack the file. If you are on a Unix system (Linux, Mac OS) type the following at the command line: `tar xvf spark-1.3.0-bin-hadoop2.4.tgz`. If you are on Windows, you can download 7zip from `http://www.7zip.org` and use it to extract the contents.

5 Downloading a sample dataset

As part of today's lab, we will be processing a large file containing approximately 4 million words.

- Download a compressed version of that file from `http://jmg3.web.rice.edu/32big.zip` using either wget or by navigating to that URL in your browser.
- Extract the 32big.txt file from that ZIP using 7zip or the unzip utility. You should store 32big.txt under the new directory you just created beneath your home directory, `spark/spark-1.3.0-bin-hadoop2.4`.

6 Running Spark

- cd into the new directory: `cd spark/spark-1.3.0-bin-hadoop2.4`
- You can now start an interactive session with Spark by calling the Spark Shell `./bin/spark-shell`.

This will start a read-eval-print loop for Scala, similar to what you might have seen for many scripting languages, such as Python. You can try it out by evaluating some simple Scala expressions:

```
scala> 1 + 2
res0: Int = 3
```

As you type expressions at the prompt, the resulting values are displayed, along with new variables that they are bound to (in this case, `res0`), which you can use in subsequent expressions.

When invoking `spark-shell`, you can alter the number of cores and bytes of memory that Spark uses with the keyword `local`. For example, to invoke `spark-shell` with one core and 2GB of memory, we write:

```
./bin/spark-shell --master local[1] --driver-memory 2G
```

7 Interacting with Spark

If you are still in a Spark session, close it using Ctrl-D and create a new one with 1 CPU core and 2GB of memory:

```
./bin/spark-shell --master local[1] --driver-memory 2G
```

In your interactive session, you can interact with Spark via the global "spark context" variable `sc`. Try this out by creating a simple RDD from the text in the large 32big.txt file you downloaded:

```
scala> val textFile = sc.textFile("32big.txt")
```

You now have a handle on an RDD. The elements in the RDD correspond to the lines in the 32big.txt file. We can find out the number of lines using `count`:

```
scala> textFile.count()
```

As shown in class, we can also use the map/reduce pattern to count the number of occurences of each word in the RDD:

```
scala> val wordCounts = textFile.flatMap(line => line.split(" ")).
                                map(word => (word, 1)).
                                reduceByKey((a, b) => a + b)
```

We can then view the result of running our word count via the `collect` operation:

```
scala> wordCounts.collect()
```

How long does this operation take when you only use one core? Note that Spark reports execution times for all operations, there should be a line at the end of `collect`'s output that is labeled with "INFO DAGSchedule" and starts with "Job 0 finished" which reports a time. Take note of this time, as we will compare it to an execution with more than 1 core later.

If you run this `collect` operation repeatedly, i.e.:

```
scala> wordCounts.collect()
scala> wordCounts.collect()
scala> wordCounts.collect()
```

does the execution time change after the first execution? Can you explain this change?

You can also test the speedup with more cores. Kill your current Spark session by pressing Ctrl+D, and launch a new one that uses 4 cores (assuming your laptop has 4 or more cores):

```
./bin/spark-shell --master local[4] --driver-memory 2G
```

Now, rerun the previous commands:

```
scala> val textFile = sc.textFile("32big.txt")
scala> val wordCounts = textFile.flatMap(line => line.split(" ")).
                                map(word => (word, 1)).
                                reduceByKey((a, b) => a + b)
scala> wordCounts.collect()
```

How does execution time change, compared to the time you noted for a single core?

8 Selective Wordcount

Your next task is to alter our map/reduce operation on `textFile` so that only words of length 5 are counted, and then display the counts of all (and only) words of length 5.

Hints:

- Scala syntax for `if` expressions is:

  ```
  if testExpr thenExpr else elseExpr
  ```

  An `if` expression can be used in any context that an expression can be used, and returns the value returned by whichever branch of the `if` expression is executed.

- The length of a string `s` in Scala can be found by using the method `s.length`

- The elements of a pair can be retrieved using the accessors `_1` and `_2`. For example:

  ```
  scala> (1,2)._1
  res2: Int = 1
  ```

- Collections in Scala (including RDDs) have a method `filter` that takes a boolean test and returns a new RDD that contains only the elements for which the testing function passed. For example, the following application of filter to a list of ints returns a new list containing only the even elements:

  ```
  scala> List(1,2,3,4).filter(n => n % 2 == 0)
  res1: List[Int] = List(2, 4)
  ```

9 Estimating π

Exit the Spark Shell using Ctrl+D and then restart it with just one core:

```
./bin/spark-shell --master local[1] --driver-memory 2G
```

We can now walk through a Spark program to estimate π in parallel from random trials. We will alter the number of cores that Spark makes use of and observe the impact on performance.

At your read-eval-print loop, first define the number of random trials:

```
scala> val NUM_SAMPLES = 1000000000
```

Now we can estimate π with the following code snippet:

```
scala> val count = sc.parallelize(1 to NUM_SAMPLES).map{i =>
  val x = Math.random()
  val y = Math.random()
  if (x*x + y*y < 1) 1 else 0
}.reduce(_ + _)
```

You can then print out an estimate of π as follows:

```
println("Pi is roughly " + 4.0 * count / NUM_SAMPLES)
```

Now exit the Spark shell and restart with 2 cores, and then 4 cores. Do you observe a speedup?

10 Turning in your lab work

For each lab, you will need to turn in your work before leaving, as follows.

1. Show your work to an instructor or TA to get credit for this lab (as in COMP 215). Be prepared to explain the lab at a high level.
2. Unlike with previous labs, you are not required to submit code to the repository.