THE UNIVERSITY OF CHICAGO | Research Computing Center

APACHE Spark

Introduction to Spark

Igor Yakushin
ivy2@uchicago.edu

December 27, 2019

How to get this tutorial

- Point your browser to:

  `https://git.rcc.uchicago.edu/ivy2/Spark`

- Once you log into midway, get the tutorial:

  ```
  git clone https://git.rcc.uchicago.edu/ivy2/Spark.git
  cd Spark
  ```

- The presentation is in docs/Spark.pdf
- The labs are in labs/

Login to midway: ssh

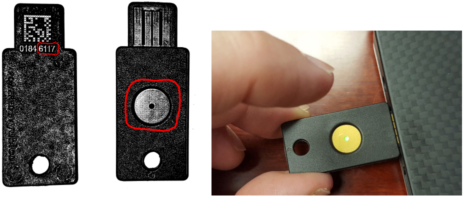
- ssh - secure shell.
- If you have an account on midway:

  ```
  ssh <cnetid>@midway2.rcc.uchicago.edu
  ```

- This assumes that you have ssh client on your laptop.
- If you have Mac or run Linux on your laptop, you should have it
- If you have MS Windows, you might need to install a client: for example putty or bitvise from `http://www.putty.org`.
- On Chromebook or any other OS you can use ssh extention to Chrome browser.

Login to midway: yubikey

- If you do not have an account on midway cluster, use yubikey:



- Use last 4 digits XXXX of yubikey as part of userid:

```
ssh rccguest<XXXX>@midway2.rcc.uchicago.edu
```

- Push the button when asked for password

Login to midway: ThinLinc

- There are two ways to connect with ThinLinc to midway:
    - Just use a web browser interface by pointing your browser to `https://midway2.rcc.uchicago.edu`. This might not work well for all browsers, you might have to try several: Chrome, Firefox, Safari...
    - For Linux, Mac and Windows there is a client you can download from `https://www.cendio.com/thinlinc/download`.
        - Configure the client to connect to `midway2.rcc.uchicago.edu`
        - You can specify the dimensions of the window in which it is running. I am usually using full screen.
- ThinLinc does not work with yubikeys, you need to have an account on midway

Introduction

- Apache Spark is a fast and general-purpose cluster computing system.
- It provides high-level APIs in Java, Scala, Python and R
- Typically all the functionality is available in Scala and Java while APIs in other languages might be behind. For example, Spark's GraphX library is only available in Scala.
- It also supports a rich set of higher-level tools including Spark SQL for SQL and structured data processing, MLlib for machine learning, GraphX for graph processing, and Spark Streaming.
- Spark can run within Hadoop cluster and knows how to deal with various Hadoop components (like HDFS, Hive, HBase, etc) and data formats.
- Spark can also run on a standalone computer or a regular HPC cluster like midway.
- Spark can utilize multiple nodes and multiple CPU cores in a node.

Introduction

- Python API, pyspark, can be used
  - in batch mode
  - interactively
    - in a python command prompt
    - in Jupyter notebook
- In this tutorial we shall use only Python API to Spark mostly on midway cluster without Hadoop.
- The simplest way to install pyspark is to use pip install pyspark.
- While MLlib - Spark's machine learning library - has some Neural Network routines, for more advanced Deep Learning framework consider using Spark in combination with BigDL library from Intel that knows how to work with Spark's RDDs and can also be installed with pip install bigdl.

Introduction

- Spark was initially started by Matei Zaharia at UC Berkeley's AMPLab in 2009, and open sourced in 2010 under a BSD license.
- In 2013, the project was donated to the Apache Software Foundation and switched its license to Apache 2.0.
- Michael Franklin, who co-founded and directed AMPLab when Spark was created, is now professor and chair at the Computer Science Department of the University of Chicago.
- The current version of Spark is 2.3.0.
- We are using version 2.3.0 on midway and 2.2.0 on Hadoop cluster.

Introduction

- Before Spark 2.0, the main abstraction of Spark was the Resilient Distributed Dataset - RDD
- After Spark 2.0, RDD is replaced by DataFrame
- RDDs are still supported
- It is recommended now to use DataFrames instead of RDDs:
  - provides SQL interface to data - more convenient to program
  - much faster since a query optimization, similar the one used for SQL in RDBM, is applied to queries on DataFrames but not on RDDs
- We shall cover both since Spark is still in the state of transition. For example:
  - There are two streaming libraries:
    - Structured Streaming - based on DataFrames,
    - DStreams - based on RDDs
  - There are two APIs to machine learning library:
    - the old one based on RDD is in a maintenance state - only bug fixes are applied to it but no new features are introduced;
    - the new machine learning library based on DataFrames is still catching up with the functionality of the old library

RDD

- RDD - collection of records partitioned across the nodes and can be operated on in parallel
- RDD can be created from files in various supported formats or by transforming other RDDs
- One can ask Spark to cache RDD in memory or disk for fast reuse
- RDDs automatically recover from node failure
- RDD supports two types of operations:
  - Transformations - create a new dataset from an existing one. Lazy evaluation - evaluated only when required by action.
  - Actions - return a value to the driver program after running a computation on the dataset.

RDD: transformations

Examples of RDD transformations:

- map(func) - transform each element by a applying a function
- filter(func) - select records satisfying boolean function
- sample(withReplacement, fraction, seed) - Sample a fraction fraction of the data, with or without replacement, using a given random number generator seed.
- union(otherDataset), intersection(otherDataset)
- distinct([numTasks]))
- groupByKey([numTasks])
- sortByKey([ascending], [numTasks])
- pipe(command, [envVars])

RDD: actions

Examples of RDD actions:

- reduce(func) - Aggregate the elements of the dataset using a function func (which takes two arguments and returns one). The function should be commutative and associative so that it can be computed correctly in parallel.
- collect() - Return all the elements of the dataset as an array at the driver program.
- count()
- take(n) - Return first n elements of the results
- countByKey() - Only available on RDDs of type (K, V). Returns a hashmap of (K, Int) pairs with the count of each key.
- foreach(func) - Run a function func on each element of the dataset. This is usually done for side effects such as updating an accumulator variable (see below) or interacting with external storage systems.

RDD: Lab 1

```
from pyspark import SparkContext, SparkConf

conf = SparkConf()
sc = SparkContext(conf=conf)

print(sc)
inputData = sc.textFile("inputFile").cache()
print(type(inputData))

numAs = inputData.filter(lambda s: ’a’ in s).count()
numBs = inputData.filter(lambda s: ’b’ in s).count()

print("Lines with a: %i, lines with b: %i" % (numAs, numBs))
```

To run it:

```
make rdd
```

Shared variables

- Spark's second abstraction - shared variables
- By default variables are not shared between tasks
- Two kinds of shared variables supported:
  - Broadcast variables - can be used to cache a value in memory on all nodes
  - Accumulators - such as counters, sums; one can only "increment" those variables; can be used to store intermediate results of reduce operation

Shared variables: Lab 2

Here we use pyspark interpreter:

```
lines = sc.textFile("README.md")
l1 = lines.map(lambda line: len(line.split()))
l1.reduce(lambda a,b: a if (a>b) else b)

pairs = lines.flatMap(lambda s: s.split()).map(lambda w: (w,1))
result = pairs.reduceByKey(lambda a,b: a+b)
print("\n".join(map(lambda x: "{} -> {}".format(*x),
                    result.collect())))

distData = sc.parallelize(list(range(1000))

b = sc.broadcast(list(range(10)))
b.value

a = sc.accumulator(0)
sc.parallelize(list(range(5))).foreach(lambda x: a.add(x))
a.value
```

Spark SQL

- Spark SQL is a component on top of Spark that introduced a data abstraction called DataFrames, which provides support for structured and semi-structured data.
- DataFrame is like a table distributed over the cluster similar to RDD
- One can query DataFrame using
  - Spark language
  - SQL - language for queries on relational databases
- In both cases the query optimization is used that typically results in a better performance over RDDs
- One can create DataFrame by applying transformations to other DataFrames, from the same sources as RDD: RDD, text files, json files, arrays, files in various Hadoop formats like parquet.
- Spark SQL can be interfaced with relational databases via ODBC/JDBC
- Spark SQL can use distributed Thrift store via ODBC/JDBC
- Spark SQL can use Hive store

Spark SQL: Lab 1

```
from pyspark import SparkContext, SparkConf
from pyspark.sql import SparkSession

sc = SparkContext(conf=SparkConf())
spark = SparkSession(sc)

inputData = spark.read.text(inputFile).cache()

numAs = inputData.filter(inputData.value.contains(’a’)).count()
numBs = inputData.filter(inputData.value.contains(’b’)).count()

print("Lines with a: %i, lines with b: %i" % (numAs, numBs))
```

Spark SQL: Lab 3: lab3.ipynb

We use jupyter notebook:

```
lines = spark.read.text("README.md")
from pyspark.sql.functions import *
z=lines.select(explode(split(lines.value,"\s+")).name("w"))
z.groupBy("w").count().take(10)
```

Spark SQL: Lab 4: sql.py

```
js = "data/people.json"
df = spark.read.json(js)
df.show()
df.printSchema()
df.select("name").show()
df.select(df[’name’], df[’age’] + 1).show()
df.filter(df[’age’] > 21).show()
df.groupBy("age").count().show()

df.createOrReplaceTempView("people")
sqlDF = spark.sql("SELECT * FROM people")
sqlDF.show()
```

Spark SQL: Lab 4: rdd2df.py

```
from pyspark.sql import SparkSession
from pyspark.sql import Row

spark = SparkSession.builder.getOrCreate()
sc = spark.sparkContext

lines = sc.textFile("data/people.txt")
parts = lines.map(lambda l: l.split(","))
people = parts.map(lambda p: Row(name=p[0], age=int(p[1])))

schemaPeople = spark.createDataFrame(people)
schemaPeople.createOrReplaceTempView("people")
teens = spark.sql("SELECT name FROM people
                   WHERE age >= 13 AND age <= 19")
teenNames = teens.rdd.map(lambda p: "Name: " + p.name).collect()
for name in teenNames:
    print(name)
```

Spark SQL: Lab 4: files.py

```
from pyspark.sql import SparkSession
from os.path import abspath
warehouse_location = abspath(’spark-warehouse’)
spark = SparkSession \
    .builder \
    .appName("Python Spark SQL Hive integration example") \
    .config("spark.sql.warehouse.dir", warehouse_location) \
    .enableHiveSupport() \
    .getOrCreate()
df = spark.read.load("data/users.parquet")
df.select("name", "favorite_color").write.save("output/t.parquet")

df = spark.sql("SELECT * FROM parquet.‘data/users.parquet‘")
print(df.show())

df.write.saveAsTable("users1")
df.write.option("path", "output/hive").saveAsTable("users")
```

Spark SQL: Lab 4: hive.py

```
warehouse_location = abspath(’spark-warehouse’)
spark = SparkSession \
    .builder \
    .appName("Python Spark SQL Hive integration example") \
    .config("spark.sql.warehouse.dir", warehouse_location) \
    .enableHiveSupport() \
    .getOrCreate()
spark.sql("CREATE TABLE IF NOT EXISTS
           src (key INT, value STRING) USING hive")
spark.sql("LOAD DATA LOCAL INPATH
           ’data/kv1.txt’ INTO TABLE src")
spark.sql("SELECT * FROM src").show()

spark.sql("SELECT * FROM users1").show()
```

Spark SQL: Lab 4: hive.sql

- hive.sql:

  ```
  select * from users1;
  ```

- spark-sql:

  ```
  spark-sql --master local[1] --name "spark-sql example"
            -f hive.sql 1>spark_sql.out 2>spark_sql.err
  ```

Machine Learning Library

- Spark Machine Learning Library - MLlib - is a distributed machine learning framework on top of Spark
- MLlib has RDD and DataFrame APIs
- RDD API is now in maintenance mode: bugs are fixed, no new features are added
- DataFrame API is catching up, once it does, RDD API will be removed
- Many common machine learning and statistical algorithms have been implemented and are shipped with MLlib which simplify large scale machine learning pipelines, including:
    - summary statistics, correlations, sampling, hypothesis testing
    - classification and regression: support vector machines, logistic regression, linear regression, decision trees, naive Bayes classification
    - cluster analysis methods including k-means
    - dimensionality reduction techniques such as SVD and PCA
    - feature extraction and transformation functions
    - optimization algorithms such as stochastic gradient descent

Lab 5: Logistic Regression

- In this lab we have a training set consisting of class label (0 or 1) and 692 numerical features
- The data is sparse and is given in libsvm format in `data/sample_libsvm_data.txt`
- Linear regression model tries to predict the class y based on features x_i as follows:

  $$y = \frac{1}{1 + e^{-(\sum_i w_i x_i + b)}}$$

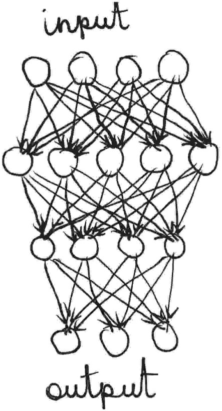
  where w_i and b are parameters that the model needs to learn to minimize error on the given training set.
- After the model is trained, the parameters are printed.
- Since most of them are zero, sparse format is used.

Lab 5: Logistic Regression

```
from pyspark.sql import SparkSession
from pyspark.ml.classification import LogisticRegression

spark = SparkSession.builder.getOrCreate()

# Load training data
training = spark.read.format("libsvm").\
                 load("data/sample_libsvm_data.txt")
lr = LogisticRegression(maxIter=10,
                   regParam=0.3, elasticNetParam=0.8)
# Fit the model
lrModel = lr.fit(training)

# Print the coefficients and intercept for logistic regression
print("Coefficients: " + str(lrModel.coefficients))
print("Intercept: " + str(lrModel.intercept))
```

Lab 5: Perceptron

- The data has
  3 classes and 4 features and is given in libsvm format
  `data/sample_multiclass_classification_data.txt`
- The model is a fully connected neural network with 4 layers.
- 60% of data is used as training set and 40% as validation set.
- 100 epochs is used for training, mini-batch size is 128.
- Prediction accuracy on the validation set is printed: 90%.



Lab 5: Perceptron

```
from pyspark.ml.classification
     import MultilayerPerceptronClassifier as mlp
from pyspark.ml.evaluation
     import MulticlassClassificationEvaluator as mce
spark = SparkSession.builder.getOrCreate()
data = spark.read.format("libsvm")\
    .load("data/sample_multiclass_classification_data.txt")
splits = data.randomSplit([0.6, 0.4], 1234)
train = splits[0]; test = splits[1]
layers = [4, 5, 4, 3]
trainer = mlp(maxIter=100, layers=layers, blockSize=128, seed=1234)
model = trainer.fit(train)
result = model.transform(test)
predictionAndLabels = result.select("prediction", "label")
evaluator = mce(metricName="accuracy")
print(evaluator.evaluate(predictionAndLabels)))
```

Streaming

- Streaming allows to process continuously arriving data
- There are two streaming interfaces in Spark:
  - Structured Streaming - DataFrame API
  - Spark Streaming (DStreams) - RDD API
- We shall only consider Structured Streaming.
- Structured Streaming is a scalable and fault-tolerant stream processing engine built on the Spark SQL engine.
- One can express streaming computation the same way one would express a batch computation on static data.
- The Spark SQL engine will take care of running it incrementally and continuously and updating the final result as streaming data continues to arrive
- Structured Streaming can use as sources: files, Kafka, socket (for testing)

Streaming: Lab 6: word_count.py

```
from pyspark.sql.functions import explode, split
import sys
port = int(sys.argv[1]); host = sys.argv[2]
spark = SparkSession.builder.getOrCreate()
lines = spark.readStream.format("socket") \
    .option("host",host).option("port",port).load()
words = lines.select(
   explode(
       split(lines.value, " ")
   ).alias("word")
)
wordCounts = words.groupBy("word").count()
query = wordCounts \
    .writeStream \
    .outputMode("complete") \
    .format("console") \
    .start()
query.awaitTermination()
```

Streaming: Lab 6: word_count.py

- Generate random port number from 9000 to 10000 and write the corresponding commands used below into the files mync.sh and mystream.sh.
- Open the second terminal on the same host and run netcat in it:

  ```
  source env.sh
  make nc
  ```

- netcat will connect to `localhost:<port>`
- In the first terminal, start pyspark streaming program:

  ```
  make word_count
  ```

- Now pyspark listens to input on `localhost:<port>`
- In the second terminal type some words. Once you hit Enter, all the words including the current ones are reprocessed by pyspark. Keep entering new lines and observe the output in the first terminal.
- To stop both processes, type Ctrl+C in the window with pyspark.

Streaming: Lab 6: age.py

```
from pyspark.sql import SparkSession
from pyspark.sql.types import StructType
from pyspark.sql.functions import avg

spark = SparkSession.builder.getOrCreate()

userSchema = StructType().add("name", "string")\
                         .add("age", "integer")
csvDF = spark.readStream.option("sep", ";") \
    .schema(userSchema).csv("input_csv")

averageAge = csvDF.select(avg(csvDF.age))

query = averageAge.writeStream.outputMode("complete")\
                              .format("console").start()
query.awaitTermination()
```

Streaming: Lab 6: age.py

- In this lab data is processed as files are added into `input_csv` directory
- First start pyspark by running

  ```
  make age
  ```

- In the second terminal

  ```
  mv tmp/1.csv input_csv/
  ```

- Observe average age computed in the first terminal
- Next

  ```
  mv tmp/2.csv input_csv/
  ```

- Notice: it is important to use mv because files for Spark Streaming are supposed to be immutable. mv simply renames a file instantaneously while something like cp might take time. As a result Spark might process the first part of the file it notices and ignore the rest.

Spark on midway cluster: sinteractive

- So far we have seen:
  - How to submit Spark job to a local node with spark-submit or spark-sql
  - How to run pyspark locally in python interpreter
  - How to run pyspark locally in jupyter notebook
- If you are doing some heavy computations on login nodes your jobs might get killed since login nodes are not meant for it
- Instead you might want to grab a compute node and do the same processing there. For example:

  ```
  sinteractive -p broadwl --exclusive --time=3:00:00
  ```

- The above will give you all the cores (28) and memory (64G) on one compute node from broadwl partition for 3 hours and would allow you to work interactively.
- You can even run jupyter notebook on the compute nodes. For more details, see References section.

Spark on midway cluster: Lab 7: single node in batch

- If you want to submit your job to a queue and run it on a single node instead of working interactively, create a batch file, single.batch

```
#!/bin/bash

#SBATCH --job-name=single
#SBATCH --exclusive
#SBATCH --nodes=1
#SBATCH --time=00:10:00
#SBATCH --partition=broadwl
#SBATCH --output=single_%j.out
#SBATCH --error=single_%j.err
####SBATCH --account=rcc-guest

module load spark/2.3.0
export MASTER="local[*]"
spark-submit --master $MASTER perceptron.py
```

Spark on midway cluster: Lab 7: single node in batch

- Notice: we are reusing perceptron example from Lab 5.
- To submit a job to midway

  ```
  sbatch single.batch
  ```
- To monitor the job, use

  ```
  squeue -j <jobid>
  ```

  or

  ```
  squeue -u <username>
  ```
- To cancel the job

  ```
  scancel <jobid>
  ```

Spark on midway cluster: Lab 7: multiple nodes in batch

- If you want to take advantage of using multiple nodes, you have to start master Spark server on one node and slave Spark servers on the other nodes, possibly including the master node. Slaves should know the address of the master.
- The job is submitted to the master Spark server.
- To automate this procedure, start-spark-slurm.sh and stop-spark-slurm.sh scripts are used.

```
...
module load spark/2.3.0

start-spark-slurm.sh

export MASTER=spark://$HOSTNAME:7077
spark-submit --master $MASTER perceptron.py

stop-spark-slurm.sh
```

Spark on Hadoop cluster: batch

- To submit a job to a Hadoop cluster you simply need to use `spark2-submit --master yarn <your program>.py` on the login node of the Hadoop cluster
- Hadoop takes care of distributing the job across the nodes. You can overwrite default number of executors in command line arguments to `spark-submit`
- Under Hadoop, Spark expects its input in HDFS and puts the output into HDFS if you use I/O-related command on RDD or DataFrame. You can still get input from local file system by using the full path with `file:///` in front.
- To run `perceptron.py` from Lab 7 inside Hadoop:

```
hdfs dfs -put data
make hadoop
```

Spark on Hadoop cluster: jupyter

- Point your browser to
  `https://hadoop.rcc.uchicago.edu/`
- Login using your midway credentials.
- Browse into `Spark/labs/3` and open `lab3.ipynb`
- Change the kernel to `pySpark 2.2.0`:
  `Kernel -> Change Kernel -> pySpark 2.2.0`
- One can execute a cell in the notebook with Shift+Enter.
- The main thing to remember: unless you shut down the notebook, it continues running and using Hadoop resources even if you close the browser and turn off your computer!!! As a result, the next user might not be able to get access to Spark either from jupyter or even in batch. This happens especially often at the end of the semester when students are doing the final project. There is a script running killing pyspark jobs that have been running for more than 3 hours.

Spark on Hadoop cluster: jupyter

Two ways to shut down a jupyter notebook:

- When inside the notebook: `File -> Close and Halt`
- When in the file browser outside of the notebook: select “Running” tab and press the yellow “Shutdown” button near any running notebook.

BigDL

- BigDL is a library for Deep Learning from Intel
- It is fully integrated with Spark and can work directly with RDDs
- While MLlib has some machine learning algorithms, including fully connected Neural Networks, it does not currently implement Deep Learning models
- BigDL runs on CPU only and relies on MKL for multithreading
- The installation of python version of BigDL is as simple as

  ```
  pip install BigDL
  ```
- BigDL API appears to be very similar to Keras

BigDL: Lab 8

- In this lab we use the models supplied with BigDL
- `local_lenet.py` is a standalone (no Spark) BigDL program to train LeNet model for handwritten digits recognition
- Run it on a single compute node with

```
make lenet_local
```

- lenet5.py is using both Spark and BigDL.
- Run it on a single compute node with

```
make lenet5
```

BigDL: Lab 8

- LeNet has been used for decades by USPS to recognize handwritten zip codes
- Here is how LeNet model is built in BigDL:

```
def build_model(class_num):
    model = Sequential()
    model.add(Reshape([1, 28, 28]))
    model.add(SpatialConvolution(1, 6, 5, 5))
    model.add(Tanh())
    model.add(SpatialMaxPooling(2, 2, 2, 2))
    model.add(Tanh())
    model.add(SpatialConvolution(6, 12, 5, 5))
    model.add(SpatialMaxPooling(2, 2, 2, 2))
    model.add(Reshape([12 * 4 * 4]))
    model.add(Linear(12 * 4 * 4, 100))
    model.add(Tanh())
    model.add(Linear(100, class_num))
    model.add(LogSoftMax())
    return model
```

BigDL: Lab 8

- MNIST data is loaded into RDD and is preprocessed in Spark before going into BigDL:

```
train_data = get_mnist(sc, "train", options.dataPath)\
      .map(lambda rec_tuple: (normalizer(rec_tuple[0],\
       mnist.TRAIN_MEAN, mnist.TRAIN_STD), rec_tuple[1]))\
      .map(lambda t: Sample.from_ndarray(t[0], t[1]))
```

Conclusion

- In this tutorial we covered Python API to Spark:
  - Spark Core: RDDs, shared variables
  - Spark SQL: DataFrames
  - Spark Machine Learning Library
  - Spark Streaming
- We also briefly looked into integrating Spark with BigDL library for Deep Learning
- We have not looked into Scala, Java and R Spark APIs
- We also have not discussed GraphX library since so far it is only implemented in Scala
- We learned how to run Spark on a local computer, on general purpose cluster under Slurm and on Hadoop cluster under Yarn
- We learned how to use Spark in batch or interactively in python shell or jupyter notebook

References

Apache Spark home page:
https://spark.apache.org

How Companies are Using Spark, and Where the Edge in Big Data Will Be
by Matei Zaharia
https://conferences.oreilly.com/strata/strata2014/public/schedule/detail/33057

Deep Learning Tutorials on Apache Spark using BigDL
https://github.com/intel-analytics/BigDL-Tutorials

Running Jupyter on a midway compute node
https://git.rcc.uchicago.edu/ivy2/Jupyter_on_compute_nodes

Troubleshooting JupyterHub and jupyter notebook
https://git.rcc.uchicago.edu/ivy2/Troubleshooting_JupyterHub

Introduction to Hadoop
https://git.rcc.uchicago.edu/ivy2/Graham_Introduction_to_Hadoop

Tutorials on demand

- I accept requests for doing tutorials on demand, outside of regular RCC Workshops schedule
- The tutorials can also be integrated into regular courses. For example, last year I did
  - "Distributed MPI Programming in Python" as part of the "Computational Astrophysics" course for graduate students
  - "Deep Learning with Keras" as part of the "Computational Physics" course for undergraduate students
- Currently the following tutorials are available at any time:
  - Distributed MPI programming in C/C++/Python
  - Introduction to Linux and RCC
  - Introduction to Python
  - Introduction to Hadoop
  - Deep Learning with Keras and TensorFlow
  - Deep Learning with Theano
  - Singularity Container
  - Introduction to Spark
- More are coming soon...