UT Southwestern Medical Center
Lyda Hill Department of Bioinformatics

BioHPC

Introduction to Apache Spark

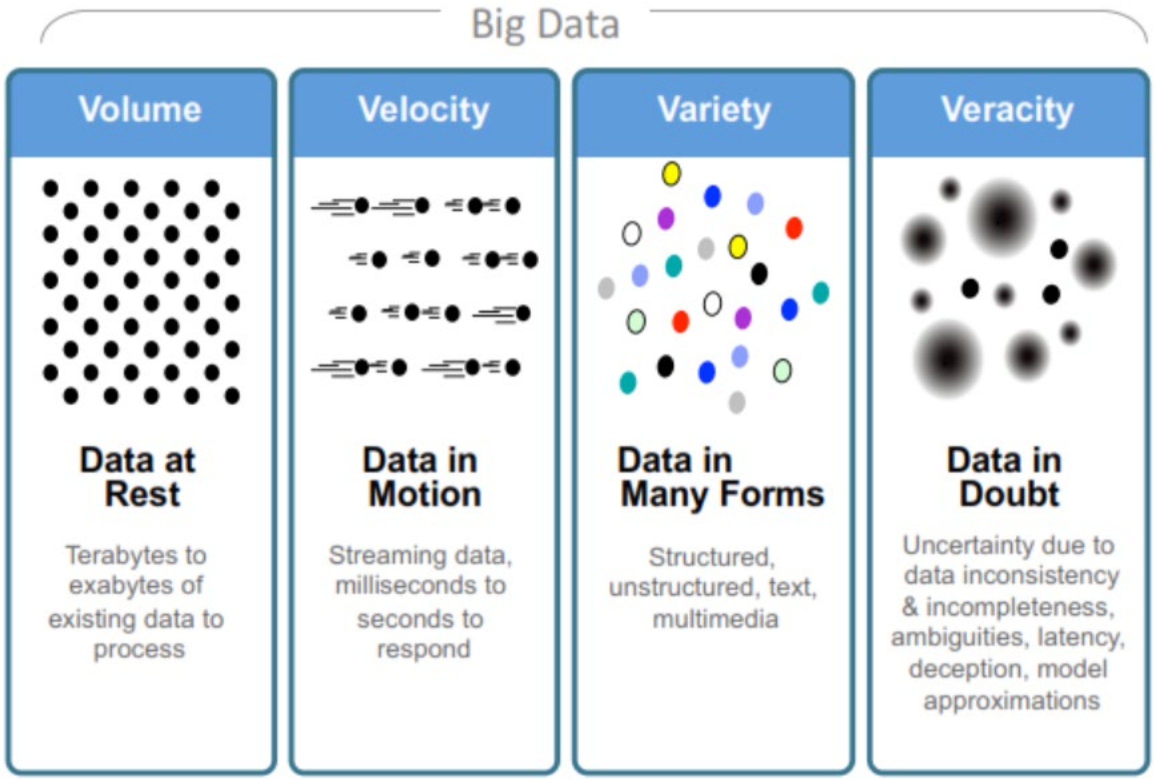
[web] portal.biohpc.swmed.edu
[email] biohpc-help@utsouthwestern.edu

Updated for 2021-05-12

Outline

- Overview of Spark platform and components
- How to submit a Spark job to the BioHPC cluster
- CPU and Memory considerations while submitting a Spark job to the cluster

Big data



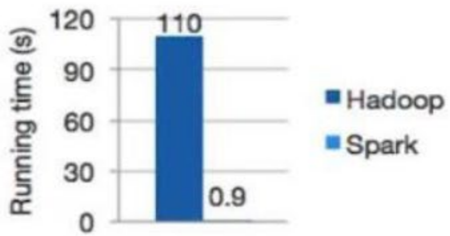
"**Big data**" is defined by IBM as any data that cannot be captured, managed and/or processed using traditional data management components and techniques.

A bit of history: Hadoop was introduced in 2006 as a general-purpose form of distributed processing. Then Spark initially developed in 2012.
Whereas Hadoop reads and writes files to HDFS, Spark processes data in RAM using a concept known as an RDD, Resilient Distributed Dataset.

- ***Apache Spark*** is a fast and general-purpose cluster computing system.
    - It provides high-level APIs in Java, Scala, Python and R, and an optimized engine that supports general execution graphs.
    - It also supports a rich set of higher-level tools including Spark SQL for SQL and structured data processing and Mllib for machine learning.
    - It eases developer to build an application to process and analyze big data in parallel on top of a cluster:
        - Image processing pipelines running on large datasets: ImageJ application for stitching microscopy images
        - Counting the words of an text file: work count example on page 15

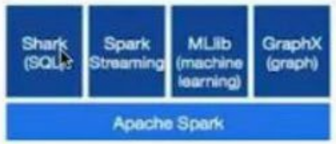
Why Spark?

Speed



Ease of use

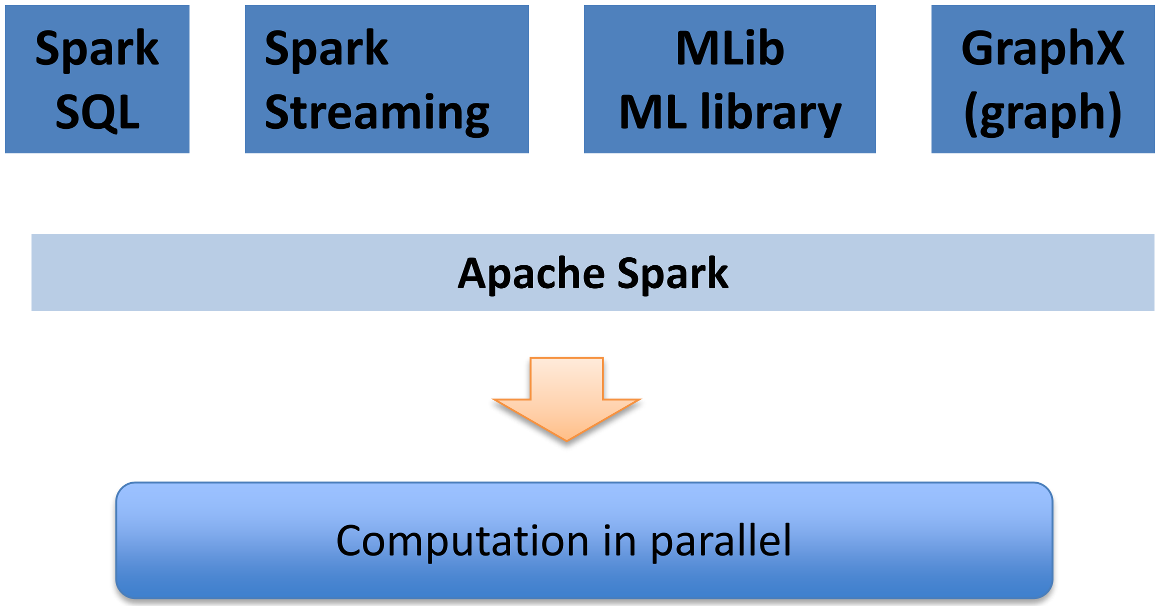
Generality



Ease of deployment

Spark Stack

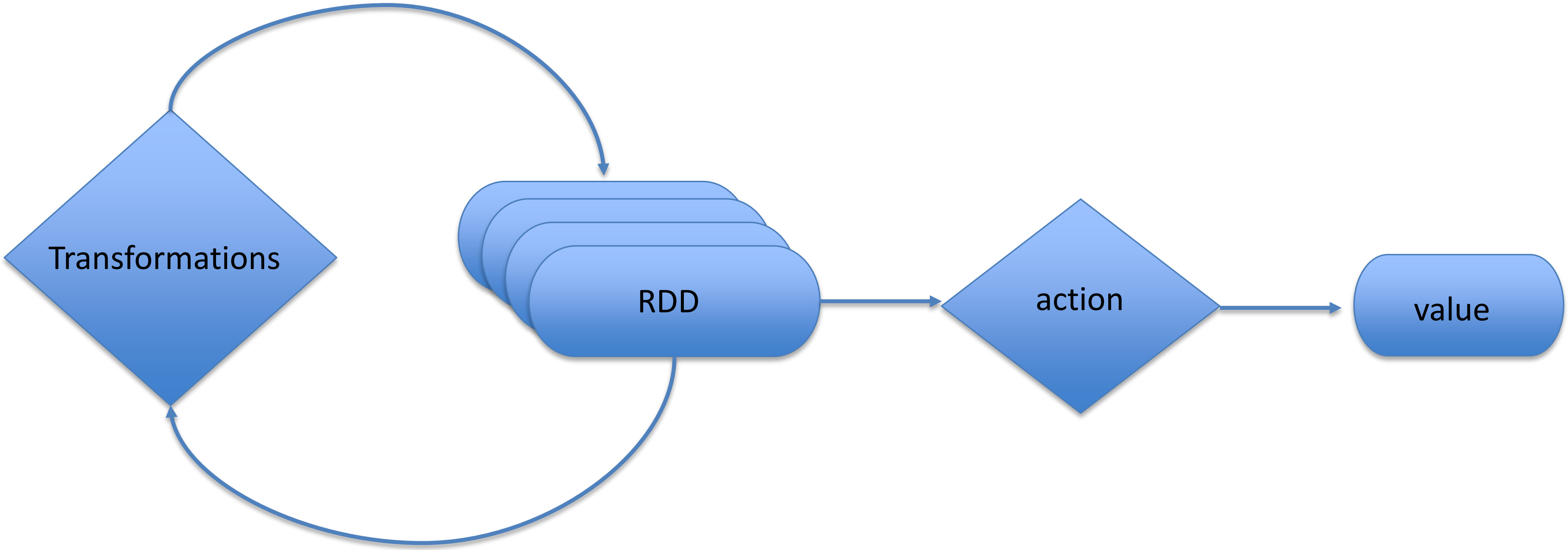
- ***Spark SQL***
  - For SQL and unstructured data processing
- ***Mllib***
  - Machine Learning Algorithms
- ***GraphX***
  - Graph Processing
- ***Spark Streaming***
  - stream processing of live data streams



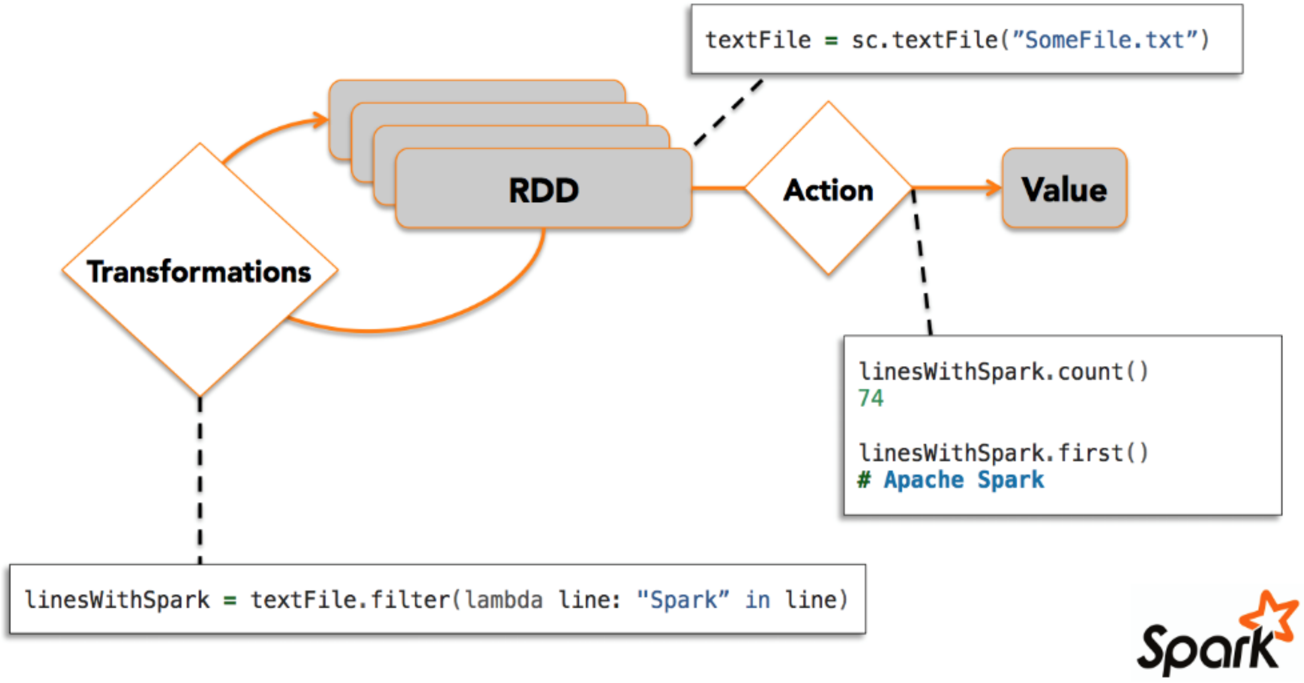
Resilient distributed dataset (RDD)

- RDD
  - Resilient-if data is lost, data can be recreated
  - Distributed-stored in nodes among the cluster
  - Dataset-initial data comes from a file or can be created programmatically
- Partitioned collection of records
- Spread across the cluster
- Read-only
- Caching dataset in memory–different storage levels available

How basic operations works



Working with RDDs



Source: http://10minbasics.com/what-is-apache-spark/

Two ways creating an RDD:

- Initialize a collection of values
  - `val rdd= sc.parallelize(Seq(1,2,3,4))`
- Load data file(s) from fileSystem, HDFS, etc.
  - `val rdd= sc.textFile("/home2/s183990/hello_world.txt")`

UTSouthwestern Medical Center
Lyda Hill Department of Bioinformatics
BioHPC

Type of operations

- Transformations
  - map
  - filter
  - union, etc.
- Actions
  - Collect
  - Count
  - foreach, etc.

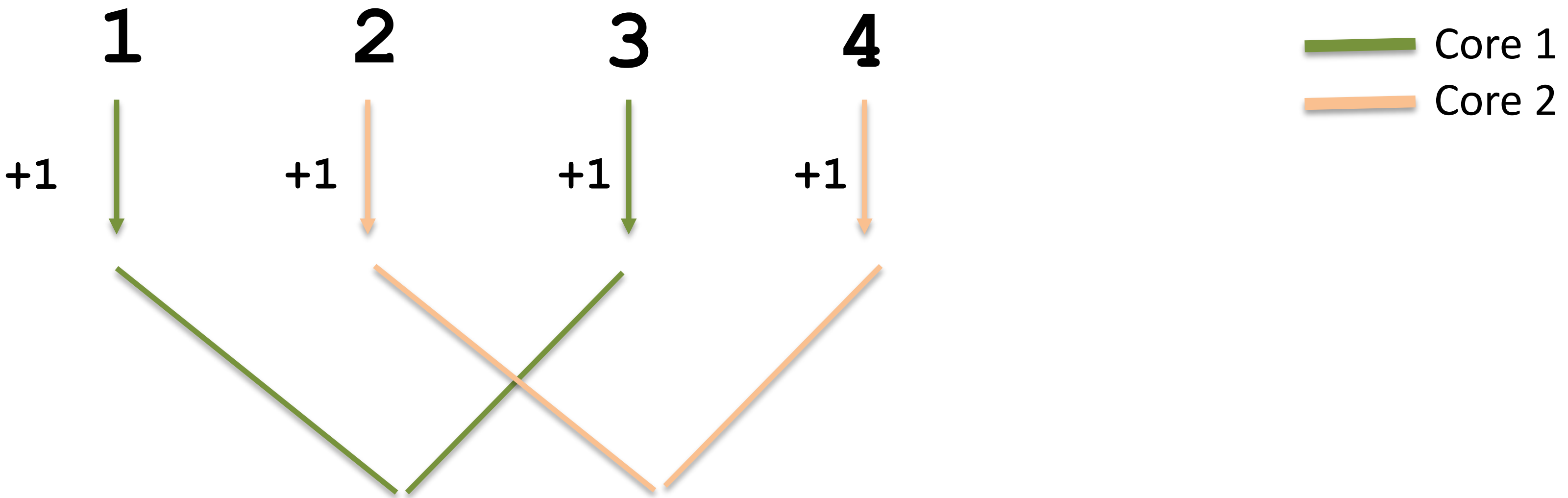
Suppose we are working with a standalone node with 4 cores.

- We want to increment by 1 each element in a list/array.
- We are running the job in sequential mode.

```
val data = Seq(1,2,3,4)
res[] = {}
for(i<-1 to 4){
    res[i] = data[i]+1
}
res = (2,3,4,5)
```

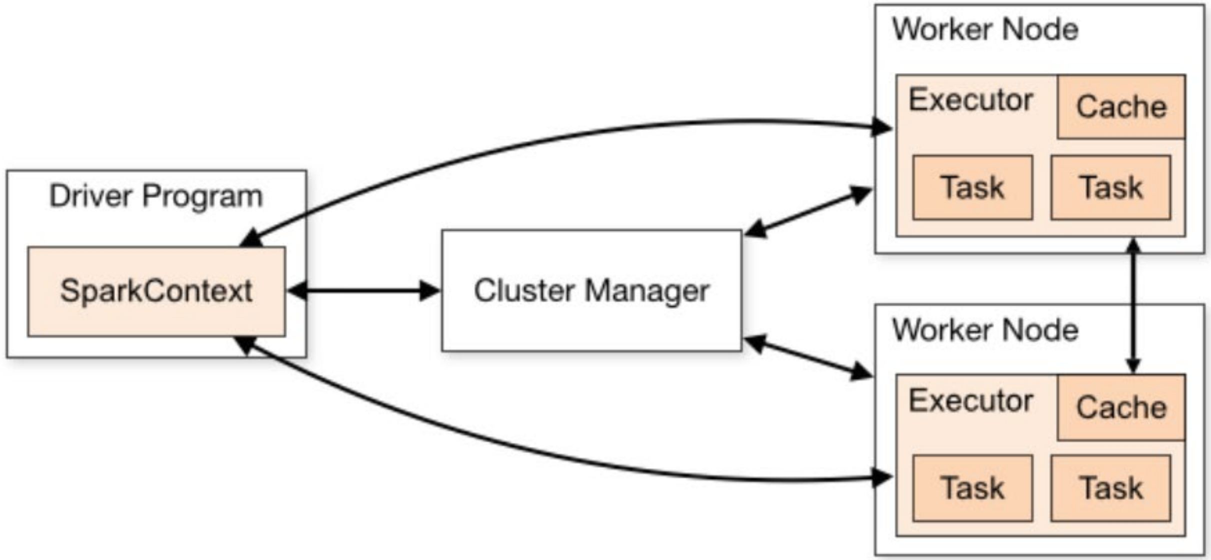
How the parallelization works (cont.)

```
val rdd= sc.parallelize(Seq(1,2,3,4))
val rddAdd= rdd.map(i=> i+1)
```




```
res: rddAdd= (2,3,4,5)
```

Execution flow of Spark



Source: http://spark.apache.org/docs/latest/cluster-overview.html

Spark application simple example: word count

```
("Gutenberg's", 1),
("Alice's", 1),
("Adventures", 1),
("in", 1),
("Wonderland", 1),
("Project", 1),
("Gutenberg's", 1),
("Adventures", 1),
("in", 1),
("Wonderland", 1),
("Project", 1),
("Gutenberg's", 1))
```

```
from pyspark.sql import SparkSession

# initialization of spark context
conf = SparkConf().setAppName(appName).setMaster(master)
sc = SparkSession\
        .builder\
        .appName("PythonWordCount")\
        .config(conf=conf)\
        .getOrCreate()

# read data from FS, as a result we get RDD of lines
linesRDD = sc.textFile("/project/biohpcadmin/s183990/hello_world.txt")

# from RDD of lines create RDD of lists of words
wordsRDD = linesRDD.flatMap(lambda line: line.split(" ")

# from RDD of lists of words make RDD of words tuples where
# the first element is a word and the second is counter, at the
# beginning it should be 1
wordCountRDD= wordsRDD.map(lambda word: (word, 1))

# combine elements with the same word value
resultRDD = wordCountRDD.reduceByKey(lambda a, b: a + b)

# write it back to FS
resultRDD.saveAsTextFile("/project/biohpcadmin/s183990/hello_world_out.txt)
spark.stop()
```

```
('Project', 2)
('Gutenberg's', 3)
('Alice's', 1)
('in', 2)
('Adventures', 2)
('Wonderland', 2)
```

Submitting a simple spark application to the cluster (live demo)

```
import sys
from random import random
from operator import add

from pyspark.sql import SparkSession


if __name__ == "__main__":
    """
        Usage: pi [partitions]
    """
    spark = SparkSession\
        .builder\
        .appName("PythonPi")\
        .getOrCreate()

    partitions = int(sys.argv[1]) if len(sys.argv) > 1 else 2
    n = 100000 * partitions

    def f(_):
        x = random() * 2 - 1
        y = random() * 2 - 1
        return 1 if x ** 2 + y ** 2 <= 1 else 0

    count = spark.sparkContext.parallelize(range(1, n + 1), partitions).map(f).reduce(add)
    print("Pi is roughly %f" % (4.0 * count / n))

    spark.stop()
```

```
#!/bin/bash

#SBATCH --partition=32GB
#SBATCH --nodes=2
#SBATCH --mem-per-cpu=4G
#SBATCH --cpus-per-task=8
#SBATCH --ntasks-per-node=2
#SBATCH --output=sparkjob-%j.out
```

https://portal.biohpc.swmed.edu/content/training/training-slides/

SLURM submission script: Preparation

```
# load the Spark module
module load spark/2.2.2

echo $SLURM_NTASKS
echo $SLURM_CPUS_PER_TASK

SPARK_DIR="/work/biohpcadmin/s183990"

# identify the Spark cluster with the Slurm jobid
export SPARK_IDENT_STRING=$SLURM_JOBID

# prepare directories
export SPARK_WORKER_DIR=${SPARK_WORKER_DIR:-$SPARK_DIR/.spark/worker}
export SPARK_LOG_DIR=${SPARK_LOG_DIR:-$SPARK_DIR/.spark/logs}
export SPARK_LOCAL_DIRS=${SPARK_LOCAL_DIRS:-/tmp/spark}
mkdir -p $SPARK_LOG_DIR $SPARK_WORKER_DIR
```

SLURM submission script: Start the worker nodes

```
# get the resource details from the Slurm job
export SPARK_WORKER_CORES=${SLURM_CPUS_PER_TASK:-1}
export SPARK_MEM=$(( ${SLURM_MEM_PER_CPU:-4096} * ${SLURM_CPUS_PER_TASK:-1} ))M
export SPARK_EXECUTOR_MEMORY=$SPARK_MEM

# start the workers on each node allocated to the job
srun  --output=$SPARK_LOG_DIR/spark-%j-workers.out --label \
      start-slave.sh ${MASTER_URL} &
```

SLURM submission script: Submit the job and clean-up after job is done

```
spark-submit --master ${MASTER_URL} \
             --total-executor-cores $((SLURM_NTASKS * SLURM_CPUS_PER_TASK)) \
             $SPARK_HOME/examples/src/main/python/pi.py 10000
```

```
# stop the workers
scancel ${SLURM_JOBID}.0

# stop the master
stop-master.sh
```

Managing CPU resources:

```
--total-executor-cores
```

Best practice: adjust the - -total-executor-cores parameter to be equal to the number of nodes times the number of tasks per node allocated for application by Slurm,

For instance:

```
#SBATCH -N 5
#SBATCH --ntasks-per-node 10
```

```
total-executor-cores = 100
(assuming 2 cores per node)
```

Managing memory resources:

- Memory used by objects
- Cost of accessing those objects
- Overhead of garbage collection

Example

Resources: 6 nodes available on a cluster with 16 core nodes and 32 GB memory per node.

Non-optimal:

```
--num-executors 6
--executor-cores 15
--executor-memory 31G
```

Optimal:

```
#SBATCH --partition=32GB
#SBATCH -N 6
#SBATCH --ntasks-per-node=4
#SBATCH --cpus-per-task=4
--executor-memory 8G
```

Thank you!

Please let us know if you have any questions regarding using spark on BioHPC by submitting a ticket to BioHPC ticket system.

biohpc-help@utsouthwestern.edu