A Report on

Addition of Portable Batch System as a cluster manager in Apache Spark

by

Utkarsh Maheshwari
2015A7PS0022G

at

Altair Engineering, Inc.
A Practice School II Station of

Birla Institute of Technology and Science, Pilani
December 5, 2018

A Report on

Addition of Portable Batch System as a cluster manager in Apache Spark

by

Utkarsh Maheshwari
2015A7PS0022G
Computer Science

Prepared in partial fulfillment of the
Practice School II Course

at

Altair Engineering, Inc.
A Practice School II Station of

Birla Institute of Technology and Science,
Pilani

December 5, 2018

Birla Institute of Technology and Science, Pilani (Rajasthan)

Practice School Division

Station: Altair Engineering, Inc.
Center: Bangalore, Karnataka.
Duration: 5 1/2 months
Date of Start: 4th July, 2018

Title: Addition of Portable Batch System as a cluster manager in Apache Spark

Student:
Utkarsh Maheshwari
2015A7PS0022G
Computer Science

Expert:
Subhasis Bhattacharya
Director - Software Development
Altair Engineering, Inc.

Faculty:
Mr. Srinivas Kota

Keywords: HPC; PBS Pro; Big Data; Apache Spark
Project Areas: Application Backend

Abstract:

This project looks at various solutions to allow using PBS Professional as a cluster manager in Apache Spark and implements a proof-of-concept prototype using one of them. The prototype allows any user to switch to using the PBS cluster of nodes in both client and cluster modes.

Signature of Student　　　　　　Signature of PS Faculty

Date　　　　　　Date

Acknowledgements

I would like to take this moment to thank **Subhasis Bhattacharya** of Altair Engineering, Inc. for accepting me in his team and trusting me with this project and guiding me towards the perfection that this project demands.

I would also like to extend my thanks to **Mr. Saksham Garg**, with whom I could have lengthy discussions about any problem I faced and **Mr. Manikanth**, to whom I could turn to anytime I needed to.

Secondly, I would like to thank my Practice School faculty in-charge **Mr. Srinivas Kota** for all the support and help.

Lastly, I would like to express my gratitude towards all my team members and colleagues at Altair Engineering, Inc. who made me feel comfortable from the very start and made my internship tenure a pleasant experience.

Contents

Introduction

About Apache Spark?

Spark is the leading frontrunner in Big Data Analytics and Machine Learning. A Spark application can run for days crunching on data and churning out results. This is why most Spark applications utilize a cluster of nodes to execute tasks to decrease the time. A Spark cluster can be set up using its own cluster manager known as Spark Standalone Cluster Manager or one of third party cluster managers like Apache Hadoop Yarn, Apache Mesos or Kubernetes.

Please see Appendix for Spark for more information on Spark's architecture.

About PBS

PBS is a workload management system which optimizes job scheduling in High Performance Computing environments — clusters, clouds and super computers. A PBS job can be anything from a batch script to a C/C++ application. It is an open sourced application with commercial support also available which can be run on Linux or Windows platforms.

Please see Appendix for PBS for more information on PBS's architecture.

Need for adding PBS as a cluster manager in Spark

There are many organizations which require running High Performance Computing jobs as well as Big Data jobs on a cluster. But none of the cluster managers which integrate with Spark are capable of running an High Performance Computing job, thus creating a need to add Portable Batch System as a pluggable scheduler in Spark. This project aims to do just that.

Project details

Problem statement

For both PBS and Spark, their own daemons for the cluster are required to be running on the nodes in the cluster and should have information about the node like number of CPU cores, memory available for use etc. Since they are two completely different programs, they have no way of knowing which node is in use by the other program at a given time and thus create problems when scheduling.

The current workaround for the said problem is to segregate the nodes cluster into two partitions — HPC cluster and Spark cluster. But this creates another problem. At any given time if the demand for running HPC jobs is high and the demand for Spark jobs is low, the Spark cluster will remain idle while HPC jobs may end up in queued state. This is a gross underutilization of the cluster for which the organization is paying but is unable to efficiently use.

Potential solutions

The potential solutions to this problem can be:

1. Running a Spark Executor on the PBS cluster endlessly. This can be done without modifying any code anywhere but it increases the need for human interaction. PBS jobs also have a walltime for all jobs. Running a Spark Executor endlessly also violates that condition.
2. Starting Spark Executors as PBS jobs on demand. This option requires knowing when a Spark job is submitted and how must resources are required for it.

Solution description

The solution which seems most feasible is to start Spark Executors on demand as HPC jobs. To do this, we must know how many executors or memory or CPU cores does the spark application require. This information can only be acquired if we hack into the Spark codebase.

Implementation

Submitting Spark applications on PBS cluster

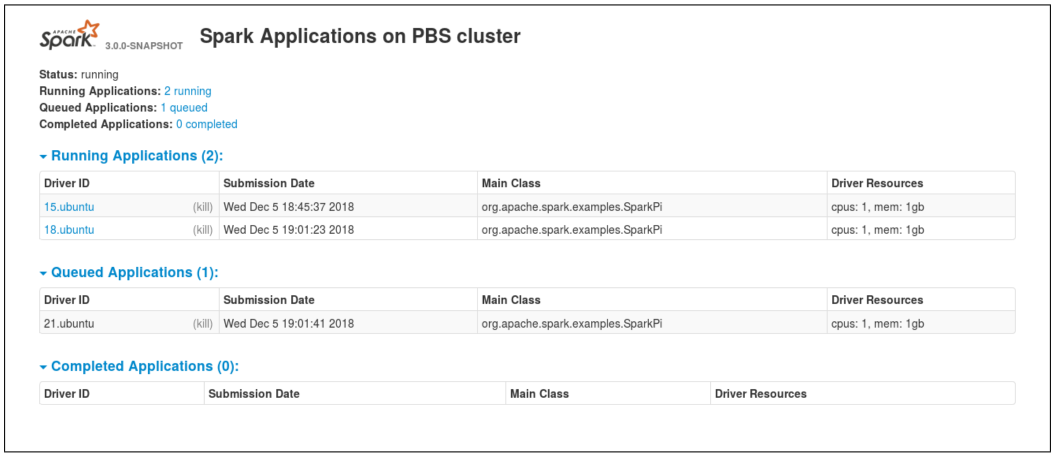
There is already support in Spark code-base for Apache Hadoop Yarn, Apache Mesos and Kubernetes. This is done by overriding the following classes in Spark:

ExternalClusterManager This class is responsible to create an instance of `SchedulerBackend` and `TaskScheduler`, and register them with the Spark application. It is discovered by the Spark build by the help of the `org.apache.spark.scheduler.ExternalClusterManager` file in `resource-managers/pbs/src/main/resources/META-INF/services/` directory.

SchedulerBackend It is the back-end for the Spark Driver, responsible for starting the Spark Executors in the Spark cluster. We override the `SchedulerBackend` so that instead of starting Spark Executors on a Spark Slave, it now submits the Spark Executors as jobs to PBS via `qsub`.

TaskScheduler It schedules the sub-tasks for each application and sends them to the Spark Executors registered to that application. We chose not to override this and keep using the Spark's implementation because we need not start a new Spark Executor for each task but keep using the already started Spark Executor. This is the Coarse Grained mode.

SparkApplication The above setup works well for a job where the Spark Driver is run locally. But to allow the Spark Driver to run remotely on a cluster, we must also be able to submit the Spark Driver as a job in the cluster. To do this, we also override `SparkApplication` which is instantiated whenever a Spark application is submitted, to start the Spark Driver as a PBS job in the cluster.



Spark Applications on PBS cluster

Status: running
Running Applications: 2 running
Queued Applications: 1 queued
Completed Applications: 0 completed

Running Applications (2):

Driver ID	Submission Date	Main Class	Driver Resources
15.ubuntu (kill)	Wed Dec 5 18:45:37 2018	org.apache.spark.examples.SparkPi	cpus: 1, mem: 1gb
18.ubuntu (kill)	Wed Dec 5 19:01:23 2018	org.apache.spark.examples.SparkPi	cpus: 1, mem: 1gb

Queued Applications (1):

Driver ID	Submission Date	Main Class	Driver Resources
21.ubuntu (kill)	Wed Dec 5 19:01:41 2018	org.apache.spark.examples.SparkPi	cpus: 1, mem: 1gb

Completed Applications (0):

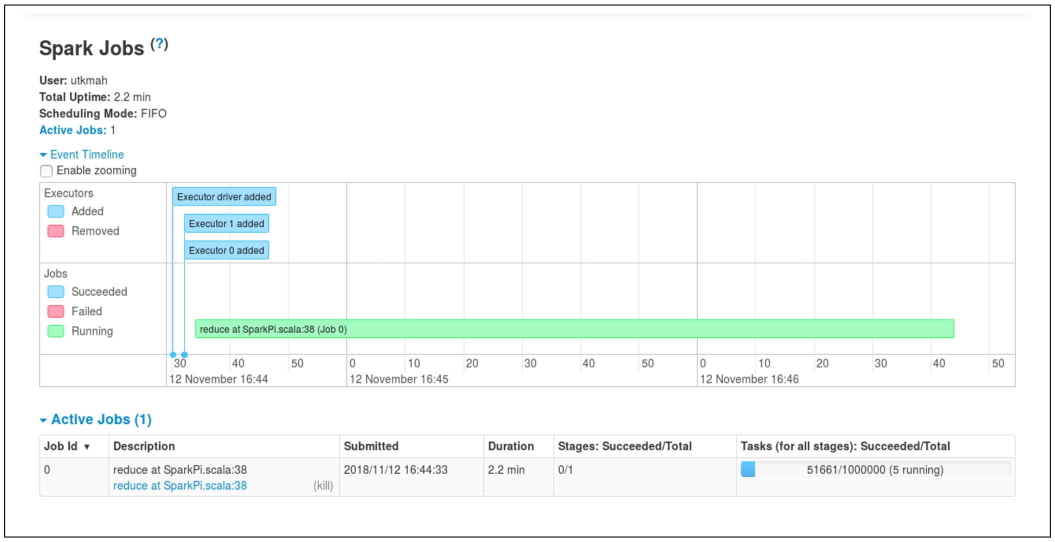
Driver ID	Submission Date	Main Class	Driver Resources

Figure 4.1: Spark Cluster UI page displaying jobs on PBS Cluster

Listing running Spark applications

Spark applications in qstat The Spark application driver running on the PBS cluster shows up in the `qstat` as "`sparkjob-`" followed by the class name of the application. The executors connected to this application show up as “`sparkexec-`” followed by the application name. The application driver does not show up when running in client mode which is expected.

Spark applications in Web UI The Spark applications running on the cluster get their own driver web UIs which are accessed on the `4040` port. There is also a master web UI page which is created by running the `org.apache.spark.deploy.pbs.ui.PbsClusterUI` class. This starts the cluster UI which lists all the Spark applications on the cluster. The user can also kill the applications by clicking the “kill” link on this page.



Spark Jobs (?)

User: utkmah
Total Uptime: 2.2 min
Scheduling Mode: FIFO
Active Jobs: 1

Event Timeline
☐ Enable zooming

Active Jobs (1)

Job Id ▾	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
0	reduce at SparkPi.scala:38 reduce at SparkPi.scala:38 (kill)	2018/11/12 16:44:33	2.2 min	0/1	51661/1000000 (5 running)

Figure 4.2: Spark Job page for each Spark Driver

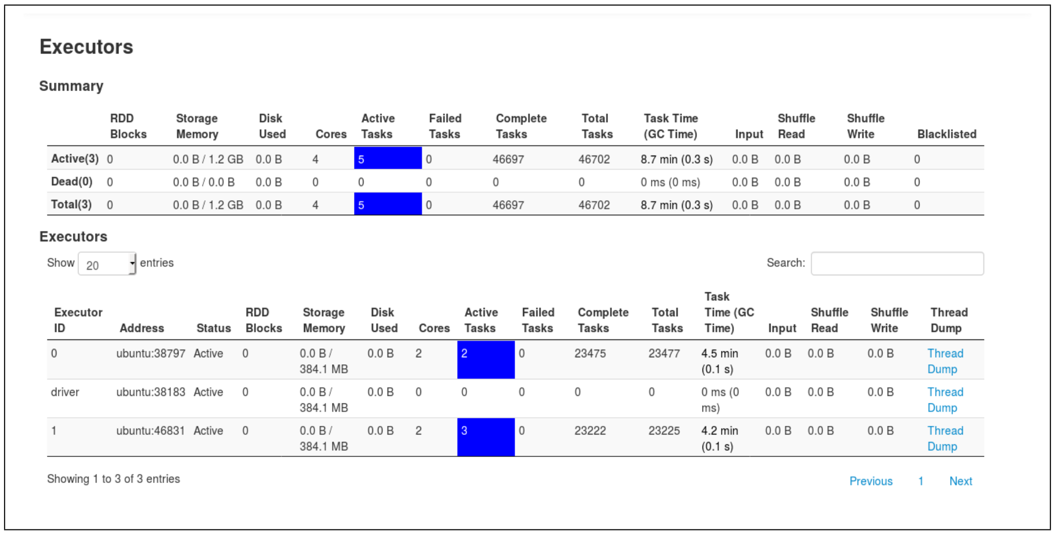
Viewing results of the Spark application

The results of the completed Spark application are stored as PBS job logs. The `stdout` and `stderr` are stored in separate files just like normal a PBS job. This can be improved later by making Web UI to read and display these logs to the user as a web page.

Result

Installation

The Spark application has to be patched to allow setting PBS as a scheduler. Apart from this, the Spark application must be compiled with the overridden classes. This whole compiled version of Spark must be installed on each execution node of the cluster as well as the client node. In addtion, to submit an application to Spark on PBS cluster, there must also be a PBS client distribution installed on the client machine.



Executors

Summary

	RDD Blocks	Storage Memory	Disk Used	Cores	Active Tasks	Failed Tasks	Complete Tasks	Total Tasks	Task Time (GC Time)	Input	Shuffle Read	Shuffle Write	Blacklisted
Active(3)	0	0.0 B / 1.2 GB	0.0 B	4	5	0	46697	46702	8.7 min (0.3 s)	0.0 B	0.0 B	0.0 B	0
Dead(0)	0	0.0 B / 0.0 B	0.0 B	0	0	0	0	0	0 ms (0 ms)	0.0 B	0.0 B	0.0 B	0
Total(3)	0	0.0 B / 1.2 GB	0.0 B	4	5	0	46697	46702	8.7 min (0.3 s)	0.0 B	0.0 B	0.0 B	0

Executors

Show 20 entries Search:

Executor ID	Address	Status	RDD Blocks	Storage Memory	Disk Used	Cores	Active Tasks	Failed Tasks	Complete Tasks	Total Tasks	Task Time (GC Time)	Input	Shuffle Read	Shuffle Write	Thread Dump
0	ubuntu:38797	Active	0	0.0 B / 384.1 MB	0.0 B	2	2	0	23475	23477	4.5 min (0.1 s)	0.0 B	0.0 B	0.0 B	Thread Dump
driver	ubuntu:38183	Active	0	0.0 B / 384.1 MB	0.0 B	0	0	0	0	0	0 ms (0 ms)	0.0 B	0.0 B	0.0 B	Thread Dump
1	ubuntu:46831	Active	0	0.0 B / 384.1 MB	0.0 B	2	3	0	23222	23225	4.2 min (0.1 s)	0.0 B	0.0 B	0.0 B	Thread Dump

Showing 1 to 3 of 3 entries

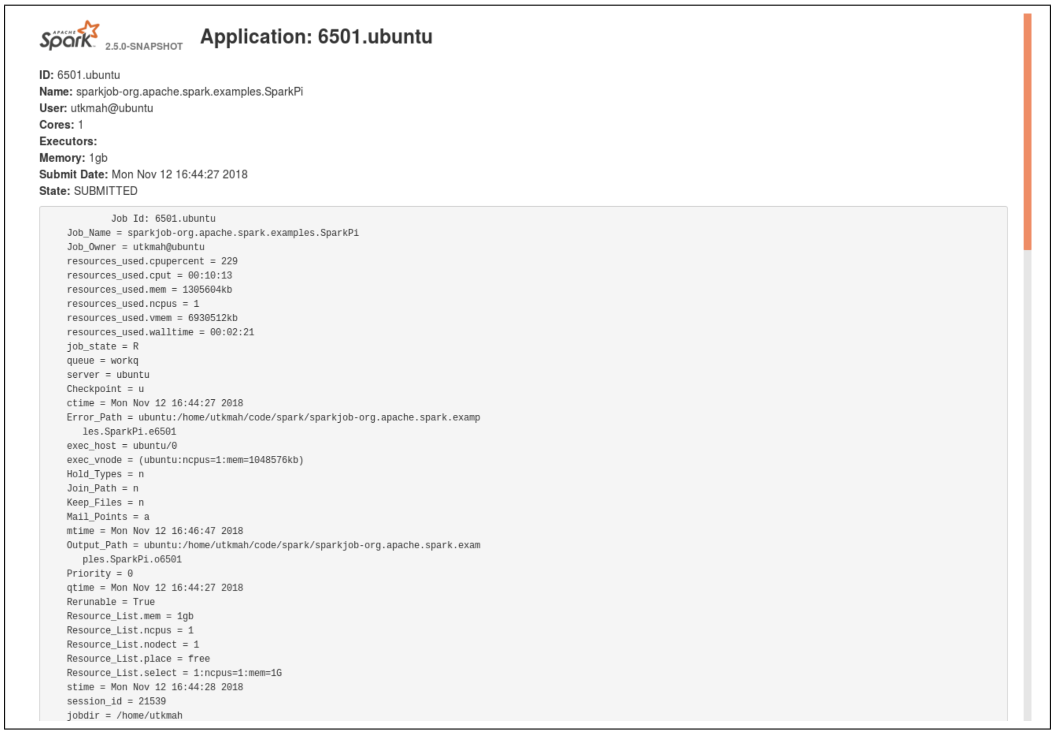
Figure 4.3: Job page listing executors which are running on PBS cluster nodes

Usage

The Spark application can be submitted just like it used to be submitted before the PBS integration with the commands following the similar pattern:

- Submit Spark application on PBS cluster:
  `bin/spark-shell --master pbs`
- Submit application with Spark Driver running locally:
  `bin/spark-submit --master pbs --deploy-mode client`
- Submit application with Spark Driver running on the cluster:
  `bin/spark-submit --master pbs --deploy-mode cluster`

In this prototype, the result for the Spark application has to be checked using the PBS job logs as of now but that can be improved once the UI is in place. The master UI can read the logs of the job and display them on the application page in the webUI.



2.5.0-SNAPSHOT **Application: 6501.ubuntu**

ID: 6501.ubuntu
Name: sparkjob-org.apache.spark.examples.SparkPi
User: utkmah@ubuntu
Cores: 1
Executors:
Memory: 1gb
Submit Date: Mon Nov 12 16:44:27 2018
State: SUBMITTED

```
        Job Id: 6501.ubuntu
    Job_Name = sparkjob-org.apache.spark.examples.SparkPi
    Job_Owner = utkmah@ubuntu
    resources_used.cpupercent = 229
    resources_used.cput = 00:10:13
    resources_used.mem = 1305604kb
    resources_used.ncpus = 1
    resources_used.vmem = 6930512kb
    resources_used.walltime = 00:02:21
    job_state = R
    queue = workq
    server = ubuntu
    Checkpoint = u
    ctime = Mon Nov 12 16:44:27 2018
    Error_Path = ubuntu:/home/utkmah/code/spark/sparkjob-org.apache.spark.examp
       les.SparkPi.e6501
    exec_host = ubuntu/0
    exec_vnode = (ubuntu:ncpus=1:mem=1048576kb)
    Hold_Types = n
    Join_Path = n
    Keep_Files = n
    Mail_Points = a
    mtime = Mon Nov 12 16:46:47 2018
    Output_Path = ubuntu:/home/utkmah/code/spark/sparkjob-org.apache.spark.exam
       ples.SparkPi.o6501
    Priority = 0
    qtime = Mon Nov 12 16:44:27 2018
    Rerunable = True
    Resource_List.mem = 1gb
    Resource_List.ncpus = 1
    Resource_List.nodect = 1
    Resource_List.place = free
    Resource_List.select = 1:ncpus=1:mem=1G
    stime = Mon Nov 12 16:44:28 2018
    session_id = 21539
    jobdir = /home/utkmah
```

Figure 4.4: Spark Application page displaying PBS Job properties

Conclusions

Final product

The final working prototype was able to submit a job, execute jobs, list running jobs and show the result of the jobs successfully on a PBS cluster. This prototype can be built with the latest Spark (currently version `2.4.0`) and the latest Scala (currently version `2.12.7`).

Change for clients

Job submission The client see no change in submission of a Spark job on a PBS cluster except adding `--master pbs` as a command-line option while submission.

Job monitoring The jobs can be seen on a webpage similar to Spark's Master UI webpage. The users have the ability to kill jobs from this page itself.

Job result and logs The job result and logs are currently accessed as PBS job logs. This is a regression from what Spark users are used to. But this issue can be solved when the Web user interface is complete, which can fetch data from the job logs and display on a web page. This behavior is similar to Spark's way for displaying job logs.

Issues and further scope of improvement

Security In the current prototype, any user who can access the web UI is able to kill user jobs. This is because the kill request is sent as user who started the UI. Since the UI is started on the server by the user with job deletion rights and all kill requests are sent with that user, there is no way of differentiating between different users.

Usability The current prototype can be optimized to prevent deadlocks in the cluster by allocating one big chunk of resources at the same time rather than in smaller, per-executor chunks. Currently, the user also has to manually `scp` the resources to the server when submitting a job. This can be improved by using the stage-in utility of PBS.

User Interface The user interface of the current prototype can be improved to provide an interface very similar to Spark with application logs and such. There is a work-in-progress pull request on the prototype directory which will resolves this issue.

More issues can be found in the issues tab of the GitHub repository on the prototype. See references.

Appendices

Portable Batch System (PBS)

PBS is a workload management system that optimizes job scheduling based on computing resources like number of CPU cores, memory, etc.

Architecture

The PBS architecture consists of **Server**, **Scheduler**, **Comm** and **Machine Oriented Mini-server** or MOM. In a typical PBS cluster, there can be only one Server, one Comm, multiple Schedulers for different queues (based on priority, resources etc.) and multiple MOMs (one for each node in the cluster).

A PBS Server is responsible for taking a job from the client and sending it to a scheduler. The scheduler then either puts the job in queue or sends it to a MOM. The MOM is responsible to execute the job and return the result back to the Server. The Comm is responsible for communication between Server, Scheduler and MOMs.

Job submission in PBS

The job submission in PBS is done via `qsub` command. The parameters of a typical job submission include the number of CPU cores, the amount of memory and the walltime for the job.

The jobs running at any given time can be checked via the command `qstat`.

Apache Spark

Spark is an engine for writing Big Data applications to produce faster results on large quantities of data.

Architecture

The Spark architecture consists of **Master** and **Slave** daemons in the Spark Standalone Cluster Manager and **Spark Driver** and **Spark Executor** which are two processes which can run on the mentioned daemons.

Master and Slave Daemons These are two daemons which are required to be started before any application can be executed on a Spark Cluster. There can be only one Master for each Spark Cluster and multiple Slaves (one for each node in the cluster). These daemons are responsible for scheduling applications on different nodes in the cluster. The Spark Applications are submitted to the Master and run on Slaves.

Spark Driver and Spark Executor These are two types of jobs which can run on a Spark cluster. Every Spark application has a Spark Driver and multiple Spark Executors. A Driver is responsible for dividing an application into several sub-tasks and scheduling these tasks on the Executors. The Executors connect to the Driver, receive tasks from it, execute the tasks and return the result back to it.

Job submission in Spark

A Spark job is a program written in either Java, Scala, Python or R. This program can be run interactively or in the background. The application itself which is responsible for breaking the program into subtasks and later assembling the result of the subtasks is called the Spark Driver. When the program is started in interactive mode (by specifying `--deploy-mode client` or starting a shell), the driver is started on the client node itself. When the application is submitted in non-interactive mode (by specifying `--deploy-mode cluster`), the driver is started on a node in the cluster itself.

The Spark Executors are required to register to the Spark Driver to receive tasks and return results. This is achieved by creating an RPC endpoint for each Spark Driver to which the Spark Executors connect to. There are two ways in which an Spark Executor can associate with a Spark Driver:

Coarse grained mode In this mode, the Spark Executor which is started once is only stopped once the Spark Driver ends or if it is explicitly closed.

Fine grained mode In this mode, the Spark Executor is started and stopped dynamically for each task.

References

PBS home page https://www.pbspro.org

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

PBS contributor's portal https://pbspro.atlassian.net/wiki/spaces/PBSPro/overview

Work repository https://github.com/UtkarshMe/Spark-PBSPro

Spark's bug-tracker ticket https://issues.apache.org/jira/browse/SPARK-25678

Pull request to Spark https://github.com/apache/spark/pull/22822

Glossary

Apache Hadoop Yarn A resource manager running on a Hadoop cluster.

Apache Mesos A cluster manager initially developed at UC, Berkley and now owned by Apache.

High Performance Computing High-performance computing is the use of parallel processing for running advanced application programs efficiently, reliably and quickly.

job A computer program comprising of tasks/steps that performs some work. It can be a shell script, a C program, a python script and so on.

Kubernetes A container-orchestration system for automating deployment, scaling and management of containerized applications.

Portable Batch System A computer software to manage job scheduling efficiently on a large number of compute nodes. See Appendix for more information.

Spark A data analytics engine owned by Apache for parallelizing data processing. See Appendix for more information.

Spark Driver A process running the main() function of the application and creating the SparkContext.

Spark Executor A process launched for an application on a worker node, that runs tasks and keeps data in memory or disk storage across them. Each application has its own executors.

walltime The actual time taken from start to the end of a job.