Big Data Analysis and Machine Learning at Scale with Oracle Cloud Infrastructure

JULY-AUGUST 2019

AUTHOR:
Michał Bień

Openlab Oracle collaboration

SUPERVISORS:
Riccardo Castellotti
Luca Canali

CERN openlab

ABSTRACT

This work has successfully deployed two different use cases of interest for High Energy Physics using cloud resources:

- **CMS Big data reduction:** This use case consists in running a data reduction workloads for physics data. The code and implementation has originally been developed by CERN openlab in collaboration with CMS and Intel in 2017-2018. It aims at demonstrating the scalability of a data reduction workflow, by processing ROOT files using Apache Spark
- **Spark DL Trigger:** This use case consists in the deployment of a full data preparation and machine learning pipeline, starting from data ingestion (4.5 TB of ROOT data), to the training of classifier using neural networks. This use case is implemented using Apache Spark and the Keras API, following previous work in collaboration with CERN openlab.

Resources for this work have been deployed using Oracle Cloud Infrastructure (OCI). In particular this project has allowed to complete:

- Setup of the project using Oracle Container Engine for Kubernetes and Oracle Cloud resources
- Troubleshooting of the oci-hdfs-connector to run Apache Spark at scale on OCI Object Storage
- Measurements of OCI Object Storage performance for the selected use cases
- Investigations and performance measurements of the resource utilisation on Oracle Container Engine for Kubernetes (OKE), when running the TensorFlow/Keras neural network model training at scale, using CPU resources, and when using GPU.

Notable results of this project:

- Produced several key improvements to the oci-hdfs-connector. The improvements are necessary to run the latest Spark version (Spark 2.4.x) on Oracle Cloud. The connector is distributed by Oracle with open source licensing, and the improvements will be fed back to Oracle.
- Improved instrumentation infrastructure for measuring Spark workloads on cloud resources, by streamlining the deployment of Spark performance dashboard on Kubernetes and developing a Helm chart
- Produced a solution for direct measurement of I/O latency for Spark workloads reading from OCI or S3 storage. The results are of general interest for Spark users, notably including the Spark service at CERN
- Developed methods to parallelize TensorFlow/Keras on Kubernetes using TensorFlow 2.0 new tf.distribute features. These are of general interest for ML practitioners, notably including the users of CERN cloud services.

TABLE OF CONTENTS

1. INTRODUCTION

Techniques and tools for data analysis of large datasets are key to the success of many projects in High Energy Physics and at CERN in general. HEP and CERN have developed many specialized toolsets [1] and processing workflows. Looking at the computing challenges for the future of LHC experiments [2], the interest is rising to investigate and adopt relevant tools and platforms from open source communities and industry. Deployments on cloud resources are getting common in HEP applications [3]. Responding to the CERN's need of scalable, elastic, on-demand data analysis, the project continues previous efforts [4], to deliver simple, scalable data analysis solution using Oracle Cloud Infrastructure, measure its performance and scalability, and compare it to CERN's in-house computing capabilities.

This work aims at collecting experience and insights on using cloud resources for use cases of interest for physics data analysis. For this scope it was decided to deploy two recently developed big data workloads and pipelines as a proof of concept, and measure their behaviour in different, public and private, cloud infrastructure environments. These environments have complex configuration that was tuned for this work. This work does not aim at benchmarking, but rather is an effort to provide meaningful insight that can be further explored by the interested communities.

2. CMS BIG DATA REDUCTION

CMS Big Data Reduction [5] runs an Apache Spark batch job written in Scala. It integrates with existing CERN architecture: the workload ingests ROOT files produced by CMS experiment and published in the CERN open data portal, and outputs histograms for di-muon events. The main challenge involved is to ingest and process data fast and efficiently. In the following experiments, a 22TB dataset was used, with 4TB and 800GB subsets selected to draw scalability curve.

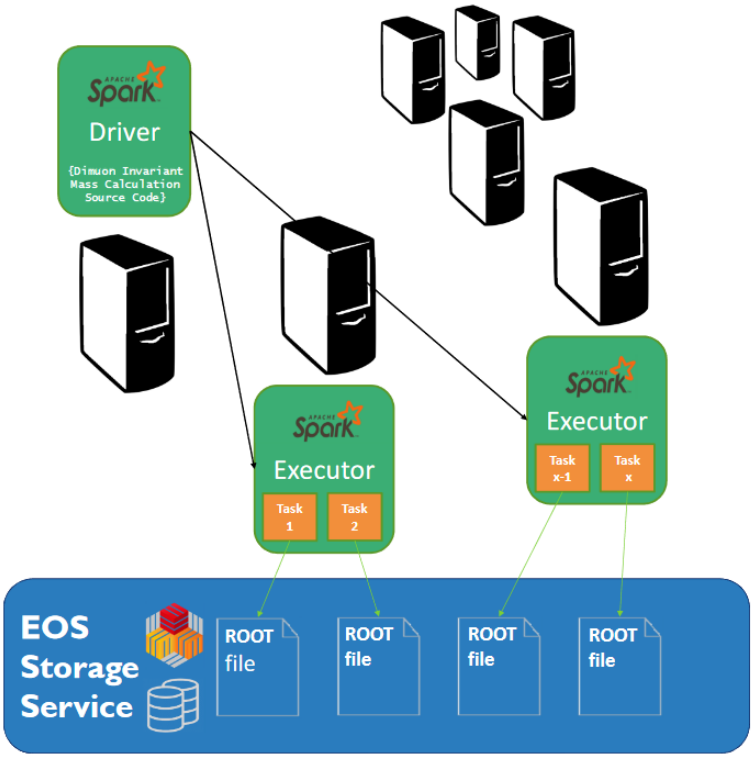


Figure 1: CMS Big Data Reduction workflow on CERN architecture

a. LIBRARIES COMPATIBILITY

Configuration of the library dependencies needed to run the job required significant work on the environment setup. Two main problems addressed by this work and their proposed solutions were:

- In the **oci-hdfs-connector** version current at the time of this work, classpath resolution by JVM left modules loaded manually (--jars and --packages) unprivileged. This in turn caused some jars to miss dependencies, or not be loaded at all by Spark (starting from version 2.3.0). To overcome this, it was decided to produce shaded (independent, conflict-free) JAR files for the extensions used by the project. **One of the shaded JARs - oci-hdfs-connector - was then submitted to the package maintainers as a solution potentially interesting for other Spark users on OCI.**
- In **spark-root** and **root4j** (libraries used to read ROOT files using Spark), the Hadoop configuration would vanish or fail to propagate into the executor pods. A partial workaround - passing hardcoded core-site.xml configuration to the workers - made it work in some cases, but not in all. After an investigation, a bug was discovered in root4j library that prevented use of loaded Hadoop configuration for all use cases involving .root files parsing. **The bug was then examined and fixed. The solution was included upstream, leading to the release of two patched versions of root4j and spark-root (version 0.1.7 and 0.1.18 respectively)**.

b. RUNNING SPARK JOBS USING THE KUBERNETES OPERATOR FOR APACHE SPARK

Running batch Spark jobs is often done using the spark-submit command included in Spark. However for this work there was an additional requirement to use cloud resources and Kubernetes. For this reason, it was decided to use **spark-on-k8s-operator** [6] to facilitate job deployment and control. The operator was installed using helm, and provided the CRD (Custom Resource Definition) for Spark applications. The resource file for deployment was then created and sent to Kubernetes cluster that ran the task using spark-operator and provided configuration. For this work, Spark version 2.4.3, the latest production Spark version at the time, was used. The operator was configured to use custom Docker images, based on spark-base-image [7] and enriched with libraries and files crucial for running this workflow.

c. EXPERIMENTAL RESULTS

The first experiments with CMS Big Data analysis workflow on OCI used a small 6-node Kubernetes cluster. Several configurations were tested using input data of increasing size: 800GB, 4TB and 22TB.

First runs were executed with one Spark task assigned to each physical CPU core. However, using OCI instrumentation it was found that overall CPU utilisation on the VMs was low. To improve on this metric, the adopted solution was to increase the number of parallel tasks per core. By experimentation, the optimum value was found and set to 4 parallel task per executor/core.

The next step was to fine-tune the oci-hdfs-connector configuration. It was discovered that file downloads were failing due to dynamic change of OCI Object Storage DNS record. For this with adjusted the Java TTL value, following recommendations in the documentation [8].

Finally, it was discovered that the specific workload of the data reduction use cases uses random access extensively. The object store that was carrying the big data is optimized for sequential file reading. Therefore, the files processing performance was much lower than that of disk-based volumes used in CERN. To overcome the problem at least partially, **an experimental look ahead feature was implemented in oci-hdfs-connector**. The feature prevented connector from re-establishing connection to storage every time a seek operation is performed on current streamed file. However, it didn't resolve the problem of backward file seeks.

At that point, the fine-tuning stopped and measurement of the parallelization speedups and efficiency was done for different cluster sizes.

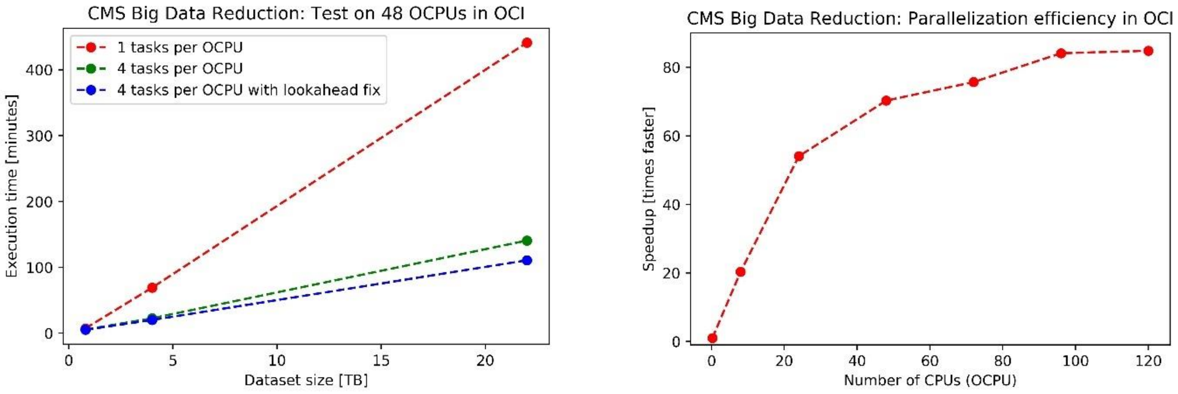


Figure 2: Measurements of the CMS Bigdata data reduction workload scalability using OCI resources. (Left) The workload scales linearly with the increase of dataset size. (Right) The workload scales linearly up to about 20 CPUs on OCI containers (OCPU), when allocating more CPU resources, the onset of saturation was noticed, which is very evident at the regime of 80 to 120 CPU.

The results of scalability benchmarks show linear scalability of the execution time as a function of dataset size, as expected, since the workload is a simple data reduction without aggregations nor joins. Speedup measurements as a function of the number of CPU cores, show almost linear scalability for low CPU utilisation, up to about 20 CPUs. Deviations from linear scalability appear clearly in the speedup graph for loads of about 20 concurrent CPUs and higher, finally saturating when more than 80 concurrent CPUs are allocated. The workflow was previously measured [9] to scale up almost linearly up to thousands of cores, therefore an effort to understand the bottleneck started.

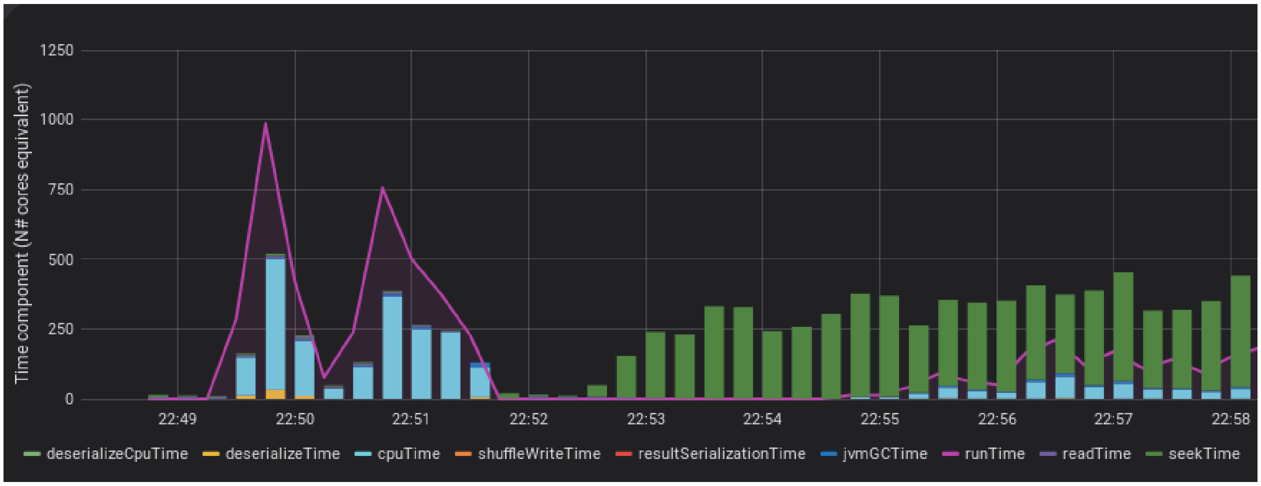


Figure 3: Significant percentage of seek operations in overall processing time

The investigation have shown that file seek times do not stay constant with increasing number of nodes, but instead they increase, making it inefficient to scale up above 6 nodes in the setup. The results point to a possible I/O bottleneck. However, due to lack of access to additional metrics on OCI side, the research in this area has been stopped and will be investigated in the future.

It was decided to compare results and times achieved in OCI environment to two cluster solutions available at CERN: Kubernetes and YARN. Custom cluster configurations (reported in section 2b) were used to run the jobs. The results achieved have clearly shown that the bottleneck is not present in the CERN infrastructure.

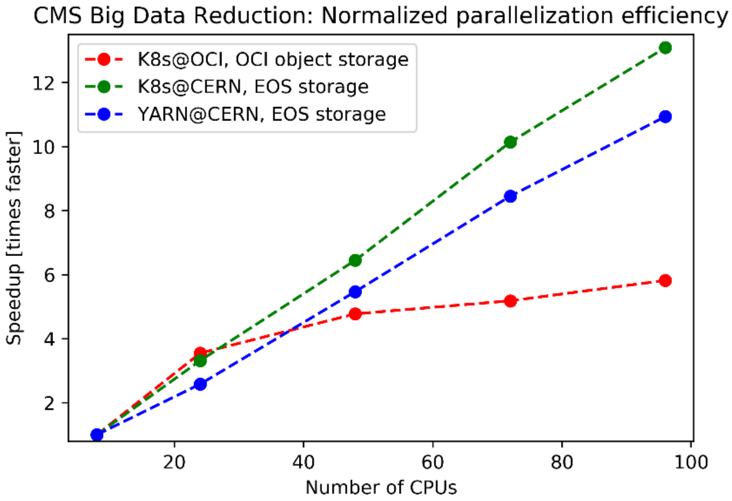


Figure 4: Speedup comparison in OCI and CERN Cloud for the CMS Bigdata data reduction workload.

While running the experiments on the CERN architecture, two additional observations were made:

- Usage of EOS storage leads to faster execution time compared to HDFS for this workload
- Kubernetes and YARN clusters scaled comparably. However, both of them suffered from minor issues that needed to be addressed:
    - The Kubernetes clusters suffered from issues related to how executor pods were deployed by Spark into the cluster in batches. This was tuned for the workload using the Spark parameter: *spark.kubernetes.allocation.batch.size.* Allocation batch size equal to the number of desired executors was used in all cases: this resulted in all executors being allocated at the application startup.
    - The YARN cluster used had nodes of different CPU and RAM capacity, and overall was "noisy" due to the presence of additional jobs and workloads. Due to these specifics, some of tasks and executors ran at a slower pace, affecting the overall processing performance. The problem was mitigated by enabling *spark.speculation*

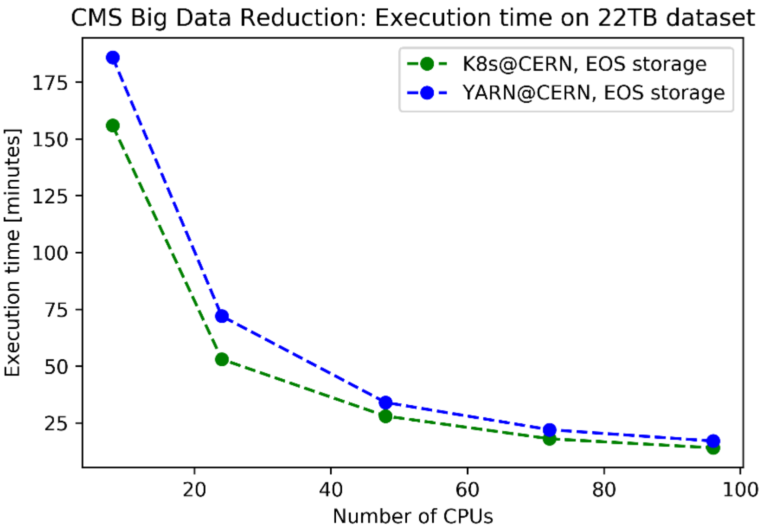


Figure 5: YARN and Kubernetes performance comparison in CERN Cloud

3. MACHINE LEARNING PIPELINE WITH SPARK ON PHYSICS DATA

The effective deployment at scale of complex machine learning (ML) techniques for HEP use cases poses several technological challenges, most importantly on the actual implementation of dedicated end-to-end data pipelines.

The use case addressed in this work consists in deploying a data pipeline for a machine learning task of interest in high energy physics: building a particle classifier to improve event selection at the particle detectors. The pipeline is built using tools from the "Big Data ecosystem", notably Apache Spark, BigDL with Analytics Zoo, and TensorFlow. The original research work by Nguyen et al. [10], on which this pipeline is based, has identified the potential benefits of using a learned particle classifier to improve the efficiency of trigger systems for LHC experiments. Further work in CERN openlab [11] has shown how tools from the Big Data ecosystem can be used to implement data preparation and deep learning at scale and has produced a set of notebooks to implement the pipeline.

The pipeline consists of: data ingestion, feature preparation, model building, and training steps. During the experiments conducted in this work, the whole pipeline was tested and run in cloud environment.

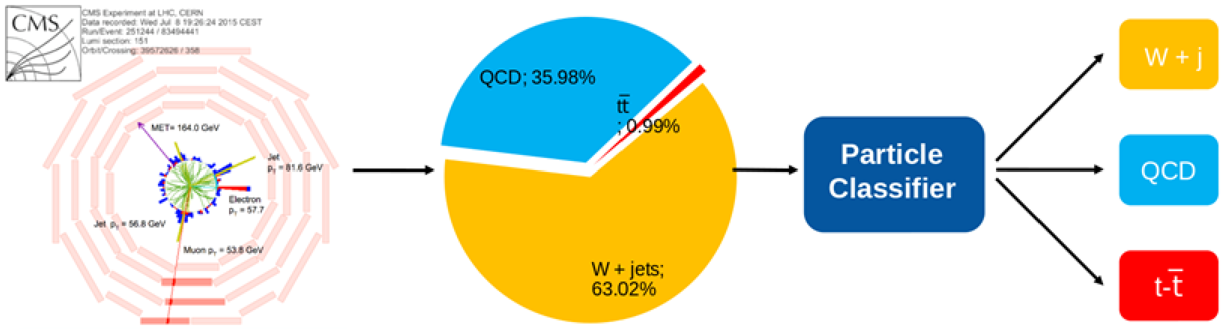


Figure 6: CMS Deep Learning Topology Classifier

a. DATA PREPARATION PIPELINE

The data preparation pipeline consists of data ingestion and feature preparation steps.

The data ingestion step acquires the original ROOT experimental data, selects the values and columns used by the workflow, and generates Parquet files as an output and starting point for the next pipeline. The process is CPU-intensive, although it also generates high traffic from and to data storage. This step was run once as a proof that it can be run on OCI. The processing time of 12h 56m 41s was measured for 24 spark executors running on total number of 12 OCPUs. During the process the data, originally 4.5TB of ROOT files residing in OCI Object Storage, was reduced into 941GB of Parquet files, and saved to the same storage.

The feature preparation step loads the data generated by the previous pipeline and prepares the dataset for deep learning classifier training. Two key processes are performed: data undersampling and shuffling into train and test sets. As an effect of the undersampling operation, which decreases representation of event classes to prevent class imbalance, the dataset shrunk to 317GB. Data shuffling process relies highly on RAM memory capacity and generates very intense traffic inside the cluster, as it needs to shuffle the data between nodes, while the whole dataset is stored in cluster memory. The shuffle implementation was also affected by bug in Spark version 2.4.3, which lead to memory overflow in driver node (see Section 3b). The whole process took 1h 26m 17s on test set (20% of data) and respectively longer on training set.

b. INTERACTIVE PYSPARK JOBS ON KUBERNETES

The code implementing the workflows was made available in Python with Jupyter Notebooks. Two Docker images were created [7] to run this code on the available cloud resources and Kubernetes:

- spark-executor-image: based on spark-base image, with added Python, PySpark support, and needed integrations (AWS, OCI…)
- spark-driver-image: based on spark-executor-image, with added Jupyter notebook installation, customized with an entry point able to serve a Jupyter notebook server on start.

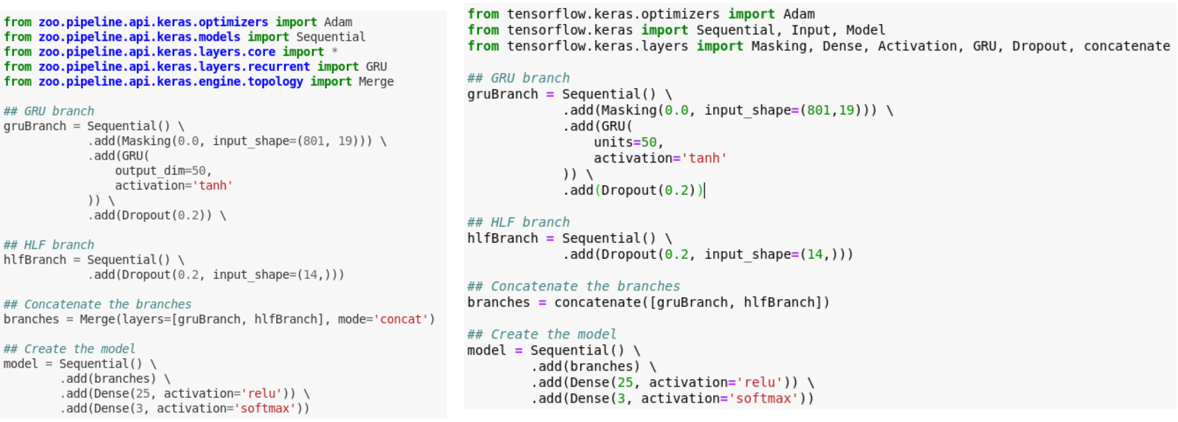
The Docker image was run manually on a host computer first, and then connected to a remote Kubernetes cluster using the local Jupyter server. To facilitate running remote interactive jobs, a helm chart was created. This method allowed running Spark DL Trigger charts in a scalable, safe, and reproducible environment.

The feature preparation step made use of data sampling and triggered Spark shuffle operations. However, due to a bug in Spark 2.4.3 mentioned above, the shuffle operations were causing significant increase of RAM memory use in driver node, effectively causing memory overflow in testing environment. To overcome the problem, feature preparation step was run on Spark-3.0.0-SNAPSHOT (August 2019). Unfortunately, the amount of cluster memory was too small to fit the data needed by shuffle operation. The problem had two possible solution: scaling cluster up, or increasing the number of shuffle partitions. The latter, allows to decrease the size of data processed at a given time and it was our choice. We used it by increasing the *spark.sql.shuffle.partitions* parameter. With this configuration, the feature preparation process completed successfully.

c. DISTRIBUTED TRAINING BIG DL AND ANALYTICS

The classification model architecture tested in this project – an "Inclusive Classifier" - consists of two dataflow branches. The first one is a recurrent neural network featuring GRU units that trains on the original input data. The second branch is an input layer that features 14 handcrafted high level features, giving the model additional insight about the data. The two branches are combined and fed to the dense layers which return three possible classes with probabilities calculated using the softmax function.

This project makes use of BigDL combined with Analytics Zoo, and handles the model architecture using only the Keras API. Thanks to the compatibility layer, Keras API objects and directives are interpreted and run as BigDL workflow, without the need of any previous user expertise in BigDL.



```
from zoo.pipeline.api.keras.optimizers import Adam
from zoo.pipeline.api.keras.models import Sequential
from zoo.pipeline.api.keras.layers.core import *
from zoo.pipeline.api.keras.layers.recurrent import GRU
from zoo.pipeline.api.keras.engine.topology import Merge

## GRU branch
gruBranch = Sequential() \
            .add(Masking(0.0, input_shape=(801, 19))) \
            .add(GRU(
                output_dim=50,
                activation='tanh'
            )) \
            .add(Dropout(0.2)) \

## HLF branch
hlfBranch = Sequential() \
            .add(Dropout(0.2, input_shape=(14,)))

## Concatenate the branches
branches = Merge(layers=[gruBranch, hlfBranch], mode='concat')

## Create the model
model = Sequential() \
        .add(branches) \
        .add(Dense(25, activation='relu')) \
        .add(Dense(3, activation='softmax'))
```

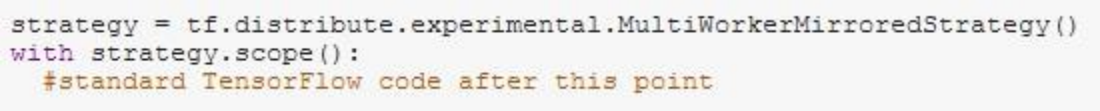
```
from tensorflow.keras.optimizers import Adam
from tensorflow.keras import Sequential, Input, Model
from tensorflow.keras.layers import Masking, Dense, Activation, GRU, Dropout, concatenate

## GRU branch
gruBranch = Sequential() \
            .add(Masking(0.0, input_shape=(801,19))) \
            .add(GRU(
                units=50,
                activation='tanh'
            )) \
            .add(Dropout(0.2))

## HLF branch
hlfBranch = Sequential() \
            .add(Dropout(0.2, input_shape=(14,)))

## Concatenate the branches
branches = concatenate([gruBranch, hlfBranch])

## Create the model
model = Sequential() \
        .add(branches) \
        .add(Dense(25, activation='relu')) \
        .add(Dense(3, activation='softmax'))
```

Figure 7: Comparison of the classifier model written with Analytics Zoo Keras API (on the left) and TensorFlow 2.0.0-rc0 (on the right). The code is nearly the same, while the underlying interpretation and implementation in machine learning framework is completely different

d. TENSORFLOW DISTRIBUTED USING CPU RESOURCES

BigDL training pipeline approach, which is very straightforward to run on Spark, is not the only possible way to run distributed training on big data. For comparison, a TensorFlow experimental API – tf.distribute with Multi-Worker Mirrored Strategy was tested.

Multi-worker strategy is a new TensorFlow runtime strategy, introduced in TensorFlow 1.14. It allows running training in a distributed cluster of machines with minimal changes to the single-node Keras models. It requires additional configuration and in particular care to run the same script on all the participating nodes. This idea, while very powerful, leads to a need of manual configuration and execution of jobs on worker machines. To automate the deployment, and execute training in scalable, yet controlled, distributed cluster environment, **an automatic configuration and deployment script for Kubernetes was created** [12]**.**



```
strategy = tf.distribute.experimental.MultiWorkerMirroredStrategy()
with strategy.scope():
  #standard TensorFlow code after this point
```

Figure 8: Python code applying tf.distribute to existing code

Another issue was related to the training data format used by TensorFlow. Currently, TFData API, used as a data fetching engine, is only capable of reading a limited number of data formats and in particular its recommended file format is tfrecord. For this reason, the input files of for the training and test dataset have been converted from Apache Parquet to TFRecord. This was done using Spark and a data source capable of saving in tfrecord format:**spark-tensorflow-connector,**made available by the TensorFlow project..

```
df.coalesce(numPartitions)

           .selectExpr("toArray(HLF_input) as HLF_input",

                   flatten(GRU_input) as GRU_input",

                   "toArray(encoded_label) as encoded_label")

           .write.format("tfrecords")

           .save(outputPath+"testUndersampled.tfrecord")
```

Figure 9: Scala code transforming parquet files into tfrecord

e. TENSORFLOW ON GPU

After running both BigDL and TensorFlow on CPU, it seemed interesting to compare this two results to the most standard TensorFlow training case – on one node, with one GPU. For that, the GPU-enabled node on OCI was used (*VM.GPU2.1*), which was deployed with NVIDIA Tesla P100 GPU. The instance was configured with the corresponding NVIDIA drivers, CUDA and cuDNN to ensure training optimization.

f. EXPERIMENTAL RESULTS

The results of the tests with Spark and BigDL and with TensorFlow distributed show very good parallelization efficiency. Near linear scalability was found in both cases up to about 200 cores when running distributed training of the neural network, for the case of the “Inclusive classifier”.

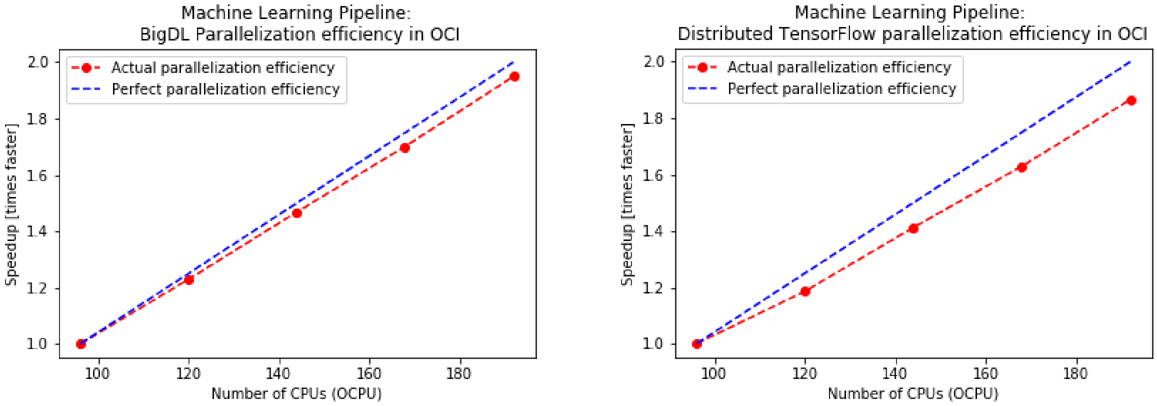


Figure 10: Speedup comparison for BigDL and TensorFlow Distributed (CPU-only)

TensorFlow distributed tests were run using resources from an Oracle OKE cluster. The testing environment used TensorFlow 2.0.0-rc0 and variable number of nodes (12 to 24). The number of TensorFlow threads (Kubernetes pods) per node was experimentally tuned and set to 2. The batch size was set to 128 both for TensorFlow and BigDL tests.

solution\nodes	12	15	18	21	24
BigDL	14.25	16.33	18.92	22.60	27.78
tf.distribute	12.30	14.08	16.25	19.35	22.95

Figure 11: Comparison of distributed training solutions - processing time in minutes, time to first epoch

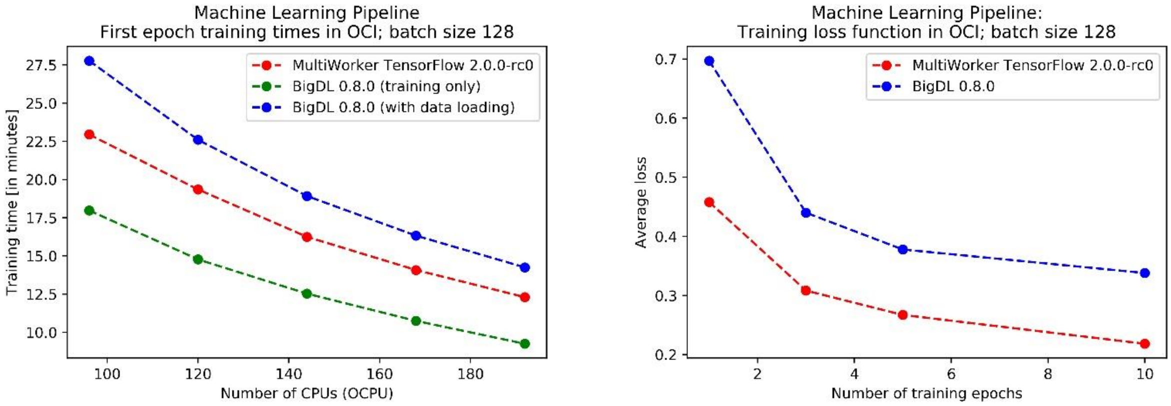


Figure 12: Comparison of BigDL and Distributed TensorFlow runs – times and losses

It's clearly visible, that MultiWorker TensorFlow and BigDL training times are comparable (slightly in favour of BigDL, as it downloads all the data before training so first-epoch training is a worst-case measure). However, during the experiments, significant difference in training performance was encountered – the TensorFlow loss function value was decreasing much faster, and was plateauing slower. The two frameworks use very different backend. **The results of these tests have been fed back to the BigDL developer team for further understanding of the root causes.**

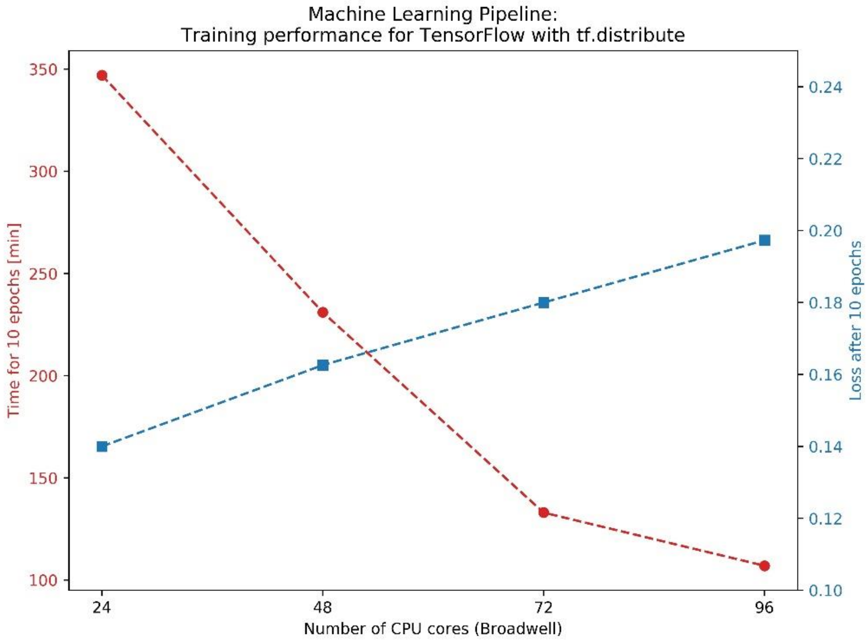


Figure 13: Time and Loss performance metrics measured for distributed training with TensorFlow on Kubernetes (CPU clusters), plotted as a function of the number of cores used for the test.

The last test that was performed was a single GPU training with vanilla TensorFlow on one node. It was prepared to compare and get an insight about the scale of difference between performance of ML training on GPU and distributed training on CPU. The tests were run on a single VM instance running in OCI (section 3d).

During the test on TensorFlow 1.14, the GPU initially appeared to have very bad performance while performing GRU training. After analysis, it became clear that the performance was lower than expected and that the GPU was underutilized. The issue was solved after updating to TensorFlow-2.0.0-rc0. TensorFlow 2.0 release notes confirm that the implementation of GRU models for NVIDIA GPUs and cuDNN library has been improved by an order of magnitude in TensorFlow release 2.0 compared to TensorFlow 1.x. The training speed increased more than 10 times after the version was upgraded.

At that moment, another test was run: the same TensorFlow training on GPU was ran in CERN using local NVIDIA GPUs with an effort to reproduce similar run environment and compare public cloud GPU training performance to utilisation of local resources. The results were very different: single GPU training took about 50 min/epoch on cloud GPU ingesting data from OCI Object Storage, while single machine with dataset available locally was able to finish the same calculations in 30 min/epoch. This result shows how distinct environment, storage and parameters setup can affect the training performance, even when the same training accelerator is in use.

In addition to comparing time for the GPU-based training and distributed CPU training on TensorFlow, the loss function decrease velocity was also analysed, as a proxy for model convergence speed. It was found that single GPU training converges faster than distributed CPU, which may make models trained on GPU better, even if smaller number of training epochs was ran.

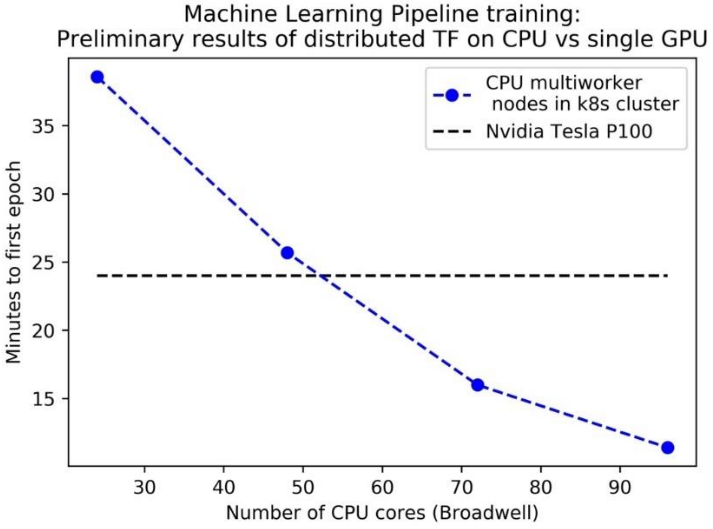


Figure 14:Comparison of training time on distributed CPUs and GPU

4. INFRASTRUCTURE

a. SOFTWARE

The software used for experiment runs can be briefly introduced as follows:

- **Docker** - container solution that provides runtime environment isolation and reproducibility
- **Kubernetes -** container orchestrator, configured as cluster of the machines, allows automatic node assignment and scheduling
- **Terraform** - Infrastructure as a Code engine that allows user to seamlessly spawn cloud infrastructure following the provided scripts describing expected final state.
- **Apache Spark (2.4.3, 3.0.0-SNAPSHOT)** - in-memory analytics engine for large-scale data processing
- **Apache Hadoop** - framework for distributed processing of large datasets. Integrates with Spark, and serves as its data ingestion engine in many situations
- **Apache Hadoop YARN -** job scheduler and cluster resource manager. In the workflows considered as alternative to Kubernetes
- **hadoop-aws -** connector for Hadoop framework, allowing ingestion of data from object storages that offer APIs compatible with those of AWS S3.
- **oci-hdfs-connector** [8] - official connector for Hadoop framework, allowing ingestion of data using native OCI Object Storage APIs.
- **root4j** and **spark-root** - libraries that enable Apache Spark to parse ROOT file format, that is used by CERN experiments.
- **xrootd** and **hadoop-xrootd** - libraries that allow high performance, direct access to CERN experiments data written in ROOT file format.
- **Intel BigDL (0.8.0)** [13] - distributed deep learning framework optimized to run on Intel CPUs in the cloud. It integrates with Spark using Intel Analytics Zoo framework.
- **TensorFlow (2.0.0rc1)** - one of the most popular machine learning libraries, recently updated with distributed training feature

i. SPARK ON KUBERNETES

Spark on Kubernetes is a relatively new feature, which is showing increasing adoption and maturity in recent Spark versions. Introduced in Spark version 2.3 and greatly extended in 2.4, Kubernetes support joins YARN, Mesos and standalone cluster, as a cluster manager for Spark, and it is the main solution for cloud native environments. Due to its wide adoption in many areas of cloud industry, Kubernetes support for Spark is expected to further gain attention and improved maturity in the upcoming next version of Spark (Spark 3.0).

Spark makes use of the Kubernetes API to connect to the service, and create the requested number of worker pods. User is expected to provide a Docker image for worker deployment. Jobs can be run in cluster or client mode: client mode runs the Spark driver on the same machine that started Spark job. Cluster mode runs driver as a separate Kubernetes pod. For this reason, the decision was made to use client mode for interactive jobs, and cluster mode for the batch ones. Spark 2.4 provides several configurable setup flags, the following table lists the main ones were used in tests:

Parameter name	**Default value (if not specified otherwise)**
spark.kubernetes.namespace	*(Kubernetes namespace name)*
spark.kubernetes.allocation.batch.size	*(equal to spark.executor.instances)*
spark.kubernetes.container.image	*(one of custom Docker images)*
spark.kubernetes.container.image.pullPolicy	Always
spark.kubernetes.pyspark.pythonVersion	3
spark.master	*(k8s://http://127.0.0.1:8001 with kubectl proxy running)*

b. HARDWARE – CLOUD RESOURCES

i. OCI CONTAINER ENGINE FOR KUBERNETES

The cloud solution used to create the cluster environment was Oracle OCI Container Engine for Kubernetes. The solution allows users to spawn Kubernetes cluster, with managed master nodes, which can then be populated with worker nodes organized in specific topologies. To deploy the cluster in an organized manner, a terraform script - oke-terraform [14] - created by Oracle, was customized and used.

Oracle Cloud Infrastructure allows its users to create flexible topologies and setups of nodes, which then can be scaled up and down easily. All this setup resides in specific tenancy and compartment. A tenancy is an isolated partition of Oracle Cloud that is specifically dedicated to an organization. All the OCI endpoints are identified by both their name and the tenancy id. The tenancies are fully independent - resources names used inside a given tenancy don't need to be globally unique. Tenancy can contain multiple compartments. Each compartment is a setup of OCI services, dedicated logically to specific group of services, and provided with distinct service limits, user privileges and services running. Compartments are organized in a tree schema. Each cluster and instance is therefore identified by its ID, tenancy, and compartment. The default oke-terraform setup allows user to spawn a Kubernetes cluster along with the desired amount of node pools. A node pool is a logical set of running instances that are meant to share the same state and configuration setup.

The OCI Frankfurt-1 region that was used for the deployment consists of 3 availability domains (AD). Each AD is a logically independent part of the Oracle datacentre. For this reason, the default setup of instances used by oke-terraform consists of multiples of 3, where each 3 machines have different AD, for better redundancy and performance concerns. Instances in the same AD are organized in subnets, one for each AD. For this project, it was decided to create OKE cluster with 1 node pool consisting of 3 subnets, variable number of instances (1 to 8) each. For the deployment, VM.Standard2.8 machine flavour was used, providing 8 physical (16 logical) Intel Xeon cores and 120GB of RAM memory [15].

In this work, the Spark was run over Kubernetes on OKE, using Spark's cluster mode for batch jobs, and client mode for interactive ones. All jobs were run with a 100GB memory threshold specified, with both *spark.dynamicAllocation* and *spark.speculation* features disabled.

ii. INFRASTRUCTURE AT CERN

CERN datacenter provides user with infrastructure, both virtualized and bare-metal, that can be organized in many different types of clusters. In this case, two solutions, YARN and Kubernetes, were compared and tested. The clusters deployed at CERN reside in different parts of datacentre - it was previously discovered [4] that deploying large scale workflows reading EOS storage on Kubernetes might cause network bandwidth bottleneck at around 20Gb/s, while connection to YARN cluster profits of network configurations with higher bandwidth to EOS storage. For these runs, cluster sizes were experimentally adjusted not to hit the bottleneck threshold on Kubernetes, while still showing full scalability potential.

- YARN cluster at CERN consists of 47 virtual machines serving as worker nodes, manually configured and collaborating together to provide boost on analytics workflows. The cluster offers total of 14.23TB of RAM memory and 1642 logical CPU cores
- Spark@K8s - a vanilla Kubernetes cluster, deployed with Openstack Magnum architecture provisioner that serve computing power for Spark on Kubernetes workflows. The part of cluster in use for this experiment offered a total of 256GB of RAM memory and 128 logical cores.

c. DOCKER IMAGES FOR KUBERNETES

As Kubernetes is an just an orchestrator for Docker containers, all of the code and configuration needed for running the user application, has to be built-in or otherwise accessible from within the container. This was a major problem when porting the existing workflows that strongly relayed on statically deployed, feature-packed, Hadoop YARN cluster nodes. It was difficult to create lightweight, yet powerful Docker image that supports all the features and libraries needed by the workflow. The official Spark image, based on Alpine, which uses musl-libc, a libc implementation, was tested, but this was not compatible with BigDL, a key library for the distributed learning part of the pipeline. Containers based on Debian were also tested, but in this case Debian had no repository support for CERN specific software like xrootd. Finally, the workflow was deployed using CERN CentOS 7 image as base Spark executor image [7]. This image was later used as a base for all the other custom Spark container images.

5. CONCLUSIONS

This project's aim was to deploy and run two different data science pipelines in the cloud. The experimental results lead to conclusion **that it was possible to run successfully on cloud resources all the tested workflows**. The outcomes of the scalability tests provided additional insights:

- A bottleneck was encountered for the CMS Big Data reduction workload running on OCI and reading data from the OCI Object Store. The bottleneck appears when running at scale greater than 20 concurrently utilized CPUs. This appears to be related to storage access and needs to be further investigated with OCI engineers.
- It was possible to compare the performance of distributed training with Spark and BigDL, Tensorflow 2.0 with tf.distribute and Tensorflow 2.0 on GPU. The results are useful for future work in this area and also provide material to further discuss with the development team of BigDL.

Another key outcome of the project, besides the workload measurements, is the development of a set of deliverables, including bug fixes, pull requests, scripts and markdown files, that **demonstrate how to run Spark and data analysis workflows on OCI with Kubernetes.** All the software developed has been uploaded to the relevant repositories and linked as references in this report. This opens the way for future users of the platform to deploy their Spark applications on OCI.

The results of the project lead to possible continuation in several ways. After receiving the feedback from Intel and Oracle on the identified performance issues, **both tests of Spark DL Trigger and CMS Big Data on OCI Object Storage can be repeated** with improved versions of the software. The GPU training is still a wide area for further investigation, especially **scalability of TensorFlow MultiWorkerMirroredStrategy on the cluster of multiple GPU-enabled instances in the cloud**. The lessons learned on OCI can be directly used for deployments on CERN private cloud.

6. BIBLIOGRAPHY

[1] R. Brun and F. Rademakers, “ROOT - An Object Oriented Data Analysis Framework,” in *Proceedings AIHENP'96 Workshop*, Lausanne, 1996.

[2] I. Bird, “Computing for the Large Hadron Collider,” *Annual Review of Nuclear and Particle Science, Vol. 61*, pp. 99-118, November 2011.

[3] CERN, “Cloud Infrastructure,” [Online]. Available: https://cern.service-now.com/service-portal/function.do?name=cloud-infrastructure. [Accessed 27 August 2019].

[4] V. Dimakopoulos, “Apache Spark on Hadoop YARN & Kubernetes for Scalable Physics Analysis,” CERN openlab, 2018.

[5] O. Gutsche, L. Canali, I. Cremer, M. Cremonesi, P. Elmer, I. Fisk, M. Girone, B. Jayatilaka, J. Kowalkowski, V. Khristenko, E. Motesnitsalis, J. Pivarski, S. Sehrish, K. Surdy and A. Svyatkovskiy, “CMS Analysis and Data Reduction with Apache Spark,” *CoRR,* vol. abs/1711.00375, 2017.

[6] Google, “spark-on-kubernetes-operator,” Github, [Online]. Available: https://github.com/GoogleCloudPlatform/spark-on-k8s-operator. [Accessed 29 August 2019].

[7] M. Bien, “public work done in 2019 with Openlab,” 20 August 2019. [Online]. Available: https://gitlab.cern.ch/db/spark-service/openlab2019/. [Accessed 3 October 2019].

[8] Oracle, “HDFS Connector for Object Storage,” Oracle Cloud, [Online]. Available: https://docs.cloud.oracle.com/iaas/Content/API/SDKDocs/hdfsconnector.htm. [Accessed 27 August 2019].

[9] M. Cremonesi, C. Bellini, B. Bian, L. Canali, V. Dimakopoulos, P. Elmer, I. Fisk, M. Girone, O. Gutsche, S.-Y. How, B. Jayatilaka, V. Khristenko, A. Luiselli, A. Melo, E. Evangelos, D. Olivito, J. Pazzini, J. Pivarski, A. Svyatkovskiy and M. Zanetti, “Using Big Data Technologies for HEP Analysis,” *CoRR,* 22 January 2019.

[10] T. Q. Nguyen, D. Weitekamp, D. Anderson, R. Castello, O. Cerri, M. Pierini, M. Spiropulu and J.-R. Vlimant, “Topology classification with deep learning to improve real-time event selection at the LHC,” 2018.

[11] M. Migliorini, R. Castellotti, L. Canali and M. Zanetti, “Machine Learning Pipelines with Modern Big Data Tools for High Energy Physics,” *arXiv,* 23 September 2019.

[12] CernDB, “tf-spawner,” 2 September 2019. [Online]. Available: https://github.com/cerndb/tf-spawner. [Accessed 20 September 2019].

[13] J. Dai, Y. Wang, X. Qiu, Y. Zhang, Y. Wang, X. Jia, C. Zhang, Y. Wan, Z. Li, J. Wang, S. Huang, Z. Wu, Y. Wang, Y. Yang, B. She, D. Shi, Q. Lu, K. Huang and G. Song, “BigDL: A Distributed Deep Learning Framework for Big Data,” *CoRR*, vol. abs/1804.05839, 2018.

[14] Oracle, “Terraform for Oracle Container Engine,” [Online]. Available: https://github.com/oracle-terraform-modules/terraform-oci-oke. [Accessed 27 August 2019].

[15] Oracle, “Cloud Computing VM Instances,” [Online]. Available: https://cloud.oracle.com/compute/virtual-machine/features. [Accessed 27 August 2019].

[16] L. Canali, “A Performance Dashboard for Apache Spark,” 12 February 2019. [Online]. Available: http://db-blog.web.cern.ch/blog/luca-canali/2019-02-performance-dashboard-apache-spark. [Accessed 20 September 2019].

[17] L. Canali, “[SPARK-28091[CORE] Extend Spark metrics system with user-defined metrics using executor plugins,” 18 June 2019. [Online]. Available: https://github.com/apache/spark/pull/24901. [Accessed 20 September 2019].

[18] CernDB, “Spark Executor Plugins,” 9 September 2019. [Online]. Available: https://github.com/cerndb/SparkExecutorPlugins2.4. [Accessed 20 September 2019].

7. APPENDIX: MEASUREMENTS, METRICS AND INSTRUMENTATION

a. SPARK DASHBOARD INTEGRATION

In order to provide metrics for monitoring Spark jobs running in the cluster, it was decided to use Grafana with InfluxDB backend (solution developed by Spark and Hadoop service at CERN [16]) to ingest data from spark and express them on the dashboard. To facilitate in-cluster deployment, a **helm chart was developed** and deployed, containing both Grafana and its functional dependencies. Then, Spark configuration was adjusted to follow the dashboard configuration.

Parameter	Value
spark.metrics.conf.*.sink.graphite.class	org.apache.spark.metrics.sink.GraphiteSink
spark.metrics.conf.*.sink.graphite.host	(spark-dashboard hostname)
spark.metrics.conf.*.sink.graphite.port	(spark-dashboard graphite port)
spark.metrics.conf.*.sink.graphite.period	10
spark.metrics.conf.*.sink.graphite.unit	seconds
spark.metrics.conf.*.sink.graphite.prefix	(prefix)
spark.metrics.conf.*.source.jvm.class	org.apache.spark.metrics.source.JvmSource
spark.app.status.metrics.enabled	true

Table 1: Spark instrumentation parameters

b. OCI INSTRUMENTATION

To allow seamless and uniform integration with OCI metrics, **a patch was backported from Spark 3.0 PR proposal** [17] **to the Spark 2.4.3**, allowing plugins inclusion in spark metrics system. Furthermore, **oci-hdfs-connector was modified to provide metrics** and **the plugin was developed** [18], leading to connection of all the parts of the Spark OCI instrumentation. The modified version of Spark was always run with additional configuration option to configure new plugin and initialize custom Spark metrics: *spark.executor.metrics.plugins=ch.cern.ExecutorMetricsPluginScala.OCICustomMetrics*

c. MEASUREMENT CRITERIA AND SPEEDUP

The execution time was used as a metric to measure performance of most experiments. All times were calculated in minutes. The software-provided metrics were used wherever possible, otherwise Linux *time* application was used to register application execution *wall time*.

Considering $t(x_n)$ is defined as an execution time of n-th experiment, counting from zero, sorted by ascending experimental cluster size, the speedups on *"Parallelization efficiency"* graphs were calculated using the formula:

$$S(x_i) = \frac{t(x_i)}{t(x_0)}$$

Meanwhile, *"Normalized parallelization efficiency"* graph has its speedup defined as follows:

$$NS(x_i) = \frac{(S(x_i) - 1) * t(s_0)}{t(x_0)} + 1$$

Where s_0 is the maximum value of x_0 of all experiment sets presented on the normalized graph.