Modern distributed analysis with ROOT

Integrating Spark in a multi-managed cluster system

Vincenzo Padulano, Stefan Wunsch

ROOT
Data Analysis Framework
https://root.cern

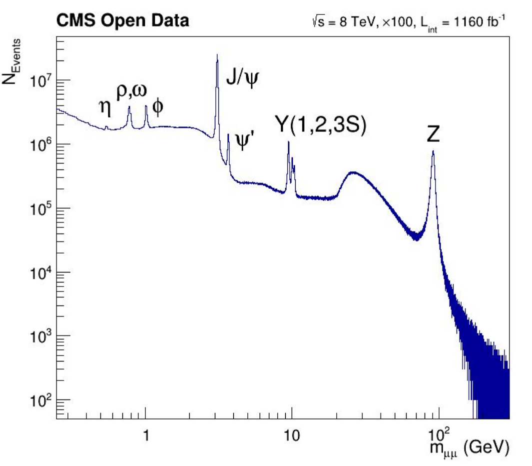
Distributed ROOT with Spark

- Targets final analysis with a high turnaround cycle (~ 30 min)
  - Making final selections, histograms, repeating the same analysis while tweaking the parameters.

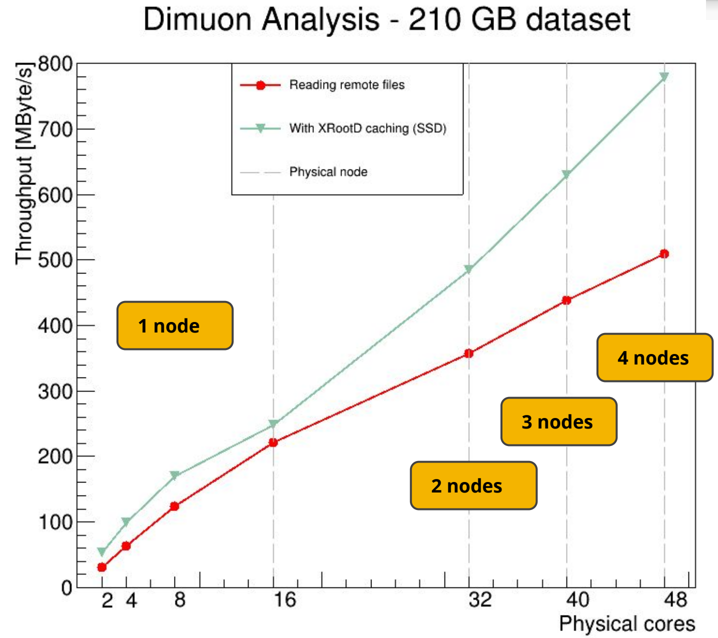
- Aggregation of the results directly in the application

- **Focus on scale out with minimal latency**
  - Scale out the application instantly on a cluster
  - Get the aggregated results right back in the application with minimal latency
  - Supports interactive analysis and minimizes the turnaround cycle

First results on KIT cluster



- [Dimuon analysis]
- Processing 210 GB (100% of total dataset)
- Data stored in:
  - Public EOS
  - Cached locally on the nodes



Processing speed with 48 cores:

- Reading from EOS: 510 MByte/s
- With cache: 780 MByte/s

Coordination requirements

In order to create a coordinated system between the traditional batch scheduler and Spark, the following requirements should be satisfied:

- Retrieve cores/memory unused by the batch system and direct them towards the Spark backend
- Allow multiple users to have access to the same resources before scaling out to other cores of the cluster
- Be able to spawn a Spark “node” with variable cpu/memory quota
  - Spawn multiple “node” objects on the same machine (Spark standalone way)
  - Have a daemon always running that can change dynamically cpu/memory (YARN way)
- Guarantee application isolation for authentication purposes:
  - Run as the submitting user, with their credentials and don’t interfere with other users
- Ensure FAIR scheduling between applications on the Spark cluster

Spark cluster setup: Standalone

```
# Create the software stack: install Spark with only requirements Java and Python

# Define the required environment variables
export SPARK_WORKER_DIR=/path/to/work/dir
export SPARK_LOG_DIR=/path/to/log/dir

export PATH=$SPARK_HOME/bin:$SPARK_HOME/sbin:$PATH
export PYTHONPATH=$SPARK_HOME/python:$PYTHONPATH

# Spawn the master
start-master.sh

# Spawn the workers (on any machine in the network)
start-slave.sh spark://<hostname or ip of the master>:7077
```

- Minimal dependencies
- Simple setup
- Weak support for multi-user scenario
  - No fair scheduling
  - No dynamic scaling of available resources
  - No native integration with authentication tools

Spark cluster setup: YARN

```
# Create the software stack (Java and Python required):
# Download Hadoop, Spark (must be compatible versions)

# Define the required environment variables
export HADOOP_HOME=/path/to/hadoop/dir
export SPARK_HOME=/path/to/spark/dir
export JAVA_HOME=/path/to/java/dir

export PATH=$SPARK_HOME/bin:$SPARK_HOME/sbin:$PATH
export PATH=$HADOOP_HOME/bin:$HADOOP_HOME/sbin:$PATH
export PYTHONPATH=$SPARK_HOME/python:$PYTHONPATH

# Spawn the YARN resourcemanager (RM)
yarn resourcemanager

# Spawn the YARN nodemanagers (on any machine in the network)
yarn nodemanager # the RM IP is written in the config file
```

- Slightly more dependencies
- Setup becomes more complicated and multiple configuration files need to be tweaked
- Strong support for multiple users
  - Fair scheduling natively
  - NM quota can be changed dynamically through REST API
  - Kerberos integration

COBalD/TARDIS + YARN/SPARK

- Through YARN we can scale Spark cluster resources up/down through a full REST api
- How do we coordinate between YARN and COBalD/TARDIS?
  - Full resources utilisation: fire up Spark workers when needed and give idle resources to HTCondor
  - Interactive analysis as first class citizen on some nodes

Backup

YARN REST API

```python
import requests
import json

# ResourceManager URL
rm_url = “http://hostname:port”
# Nodes of the cluster
nodes_path = rm_url + “/ws/v1/cluster/nodes”

# Retrieve resources of a certain node
node_url = nodes_path + NODE_ID
resources = requests.get(node_url).json()[“node”][“totalResource”]

# Set new quotas
payload = {"resource": {"memory": MEMORY, "vCores": CORES}, "overCommitTimeout": -1}
requests.post(node_url + “/resource”, json=payload)
```

Distributed analysis with RDataFrame

Example of a distributed analysis with RDataFrame

```python
import ROOT
import PyRDF # Development module for distributed RDataFrame

PyRDF.use('spark', conf={
    'spark.master': 'spark://sg01:7077',
    'spark.app.name': 'Dimuon spectrum',
})

df = PyRDF.RDataFrame('Events',
                      'root://eospublic.cern.ch//eos/opendata/cms/derived-data/' +
                      'AOD2NanoAODOutreachTool/Run2012BC_DoubleMuParked_Muons.root')

h = df.Filter('nMuon == 2')
      .Filter('Muon_charge[0] != Muon_charge[1]')
      .Define('Dimuon_mass',
              'ROOT::VecOps::InvariantMass(Muon_pt, Muon_eta, Muon_phi, Muon_mass)')
      .Histo1D(ROOT.RDF.TH1DModel('', '', 30000, 0.25, 300), 'Dimuon_mass')

h.Draw() # Trigger event loop executed distributedly on the Spark cluster
```

- **Does this work only with Spark?**
  - The layer is independent of a specific backend/scheduler
  - Not reinventing the wheel:
    Using third-party scheduler to execute tasks distributedly
  - So far supported is ...
    - ... local multi-threading (supported natively by RDataFrame)
    - ... Spark (local and distributed)
  - Another popular scheduler to be added soon is Dask

- **Why do we need a layer on top of RDataFrame to distribute the computation?**
  - Computing appropriate ranges for single partitions of the full dataset taking into account the details of the ROOT file format, e.g., the range of compressed clusters
  - Minimal changes to the programming model of conventional RDataFrame code

Importance of caching

- Final steps of the analysis with high turnaround cycle is typically heavily IO bound

- Typical bandwidth to the file server: 10 Gbit/s
  - 10 Gbit/s = 1280 MByte/s
  - Running in the cluster on 100 cores ➔ 13 MByte/s per core
  - Typical single core performance when reading from disk: ~ 50 MByte/s
  - Scaling out (and scaling up) to hundreds of cores complicated while reading from remote

- Caching as the solution to improve the throughput and provide fast turnaround cycles

- Typical read speed with random access of a ...
  - ... HDD: < 10 MByte/s
  - ... SSD: ~ 100 MByte/s ➔ SSD cache is important with high concurrency per node

- Possible solutions for the cache design:
  - XRootD proxy with various setups, e.g.,
    single proxy per node with filesystem cache
  - TFilePrefetch (similar to XRootD filesystem cache)

XRootD

Future fields of study

- How to integrate a Spark cluster (or any other cluster) in the typical HEP ecosystem?

- Is the future in HEP a mix of Spark-like distributed task scheduling and traditional batch systems?
  - HTCondor-like system ensures efficient usage of resources for computation intensive tasks
    (simulation, skimming, ntuplization, ...)
  - Spark-like system with a minimal latency reduce the turnaround cycle of the final analysis steps which are often repeated
    (counting, histogramming, fast control plots, ...)

- Is it an interesting project to offer a simple yet efficient integration of Spark-like and HTCondor-like schedulers in a coherent infrastructure?

- Similar projects, mainly focused on the scikit-hep software stack
  - USCMS analysis facilities (HSF workshop 2020)
  - Aachen T4 cluster (FSP 2020)