Distributed Analysis in production with RDataFrame

Marta Czurylo[1,*], Danilo Piparo[1], Vincenzo Eduardo Padulano[1], Andrea Ola Mejicanos[2]

(1) CERN, EP-SFT
(2) Berea College

(*) marta.maja.czurylo@cern.ch

24.10.2024, Kraków, Poland

Introduction

Since ROOT 6.14

RDataFrame

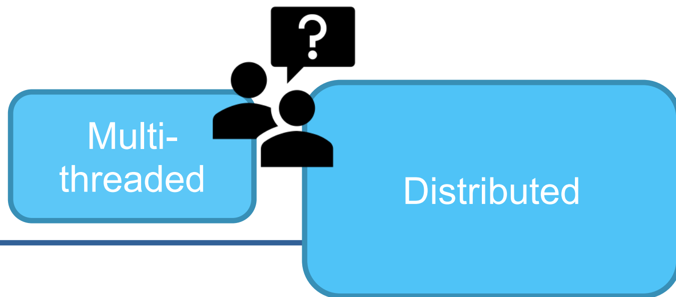
Multi-threaded

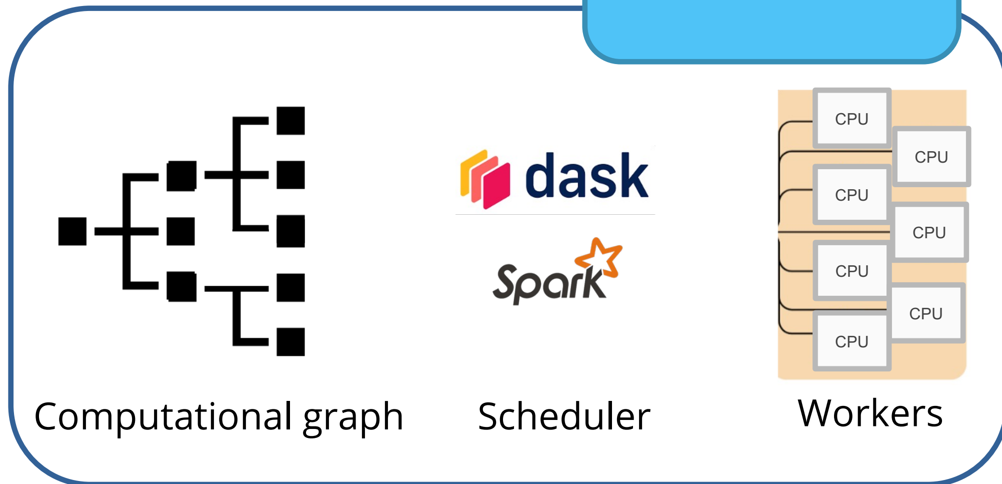
[ROOT's High Level Analysis Interface]

Introduction

Since ROOT 6.24

RDataFrame

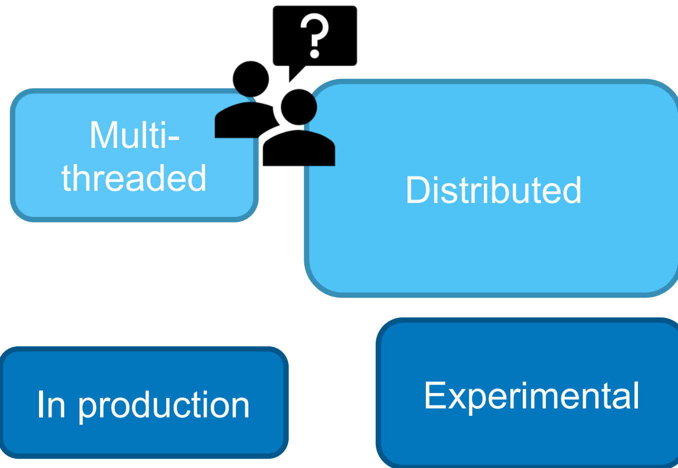




ROOT's
High Level Analysis
Interface

As of ROOT 6.32

RDataFrame



[ROOT's
High Level Analysis
Interface]

Beyond ROOT 6.32

RDataFrame

Multi-threaded

Distributed

Can we get out of Experimental

- What is the **user interface** like?
- Which **data input sources** are supported?
- What about the **Pythonisations**?
- Does DistRDF perform well with various **Analysis Facilities**?

[ROOT's High Level Analysis Interface]

Code Stability

User Interface

Multi-threaded, non-distributed RDF

```
ROOT.EnableImplicitMT()

RDataFrame = ROOT.RDataFrame

df = RDataFrame(treeName, fileName)
```

Distributed RDF

```
RDataFrame = ROOT.RDF.Experimental.Distributed.Dask.RDataFrame

df = RDataFrame(treeName, fileName, daskclient=client)
```

```
RDataFrame = ROOT.RDF.Experimental.Distributed.Spark.RDataFrame

df = RDataFrame(treeName, fileName, sparkcontext=sparkcontext)
```

Continue with Analysis – no code differences

```
myAnalysis = df.Define(...).Filter(...).Histo1D(...)
```

Feature parity between MT RDF and DistRDF conserved where applicable

- Recently added a few new RDF query functions, for example:

```
GetColumnNames(), GetColumnType("columnName")
```

User Interface – constructor unification

Multi-threaded, non-distributed RDF

```python
ROOT.EnableImplicitMT()

RDataFrame = ROOT.RDataFrame

df = RDataFrame(treeName, fileName)
```

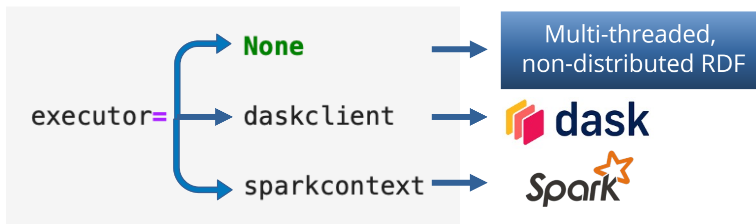
Distributed RDF

```python
RDataFrame = ROOT.RDF.Experimental.Distributed.Dask.RDataFrame

df = RDataFrame(treeName, fileName, daskclient=client)
```

```python
RDataFrame = ROOT.RDF.Experimental.Distributed.Spark.RDataFrame

df = RDataFrame(treeName, fileName, sparkcontext=sparkcontext)
```

Unify the three RDataFrame constructors based on the 3rd input argument specifying the executor

```python
RDataFrame = ROOT.RDataFrame(treeName, fileName, executor)
```



User Interface – functional unification

Some functional calls for Distributed and MT versions differed

```
if type(df).__module__ == "DistRDF.DataFrame":
    variationsfor_func = ROOT.RDF.Experimental.Distributed.VariationsFor
else:
    variationsfor_func = ROOT.RDF.Experimental.VariationsFor
```

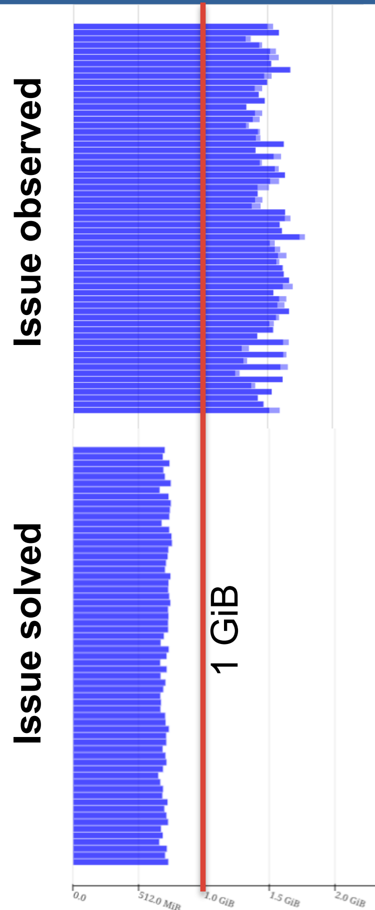
Now → unified version for both cases

```
ROOT.RDF.Experimental.VariationsFor
```

Also:

```
ROOT.RDF.RunGraphs
```

True **zero code change** for the user between MT and DistRDF



Issue observed: in some computationally heavy workflows, memory of the HTCondor Workers was increased to the level that the application was unusable

```
Worker tcp://127.0.0.1:36501 (pid=11453) exceeded 95% memory budget. Restarting...
Worker tcp://127.0.0.1:44505 (pid=11521) exceeded 95% memory budget. Restarting...
Worker tcp://127.0.0.1:38437 (pid=11474) exceeded 95% memory budget. Restarting...
Worker tcp://127.0.0.1:34547 (pid=11497) exceeded 95% memory budget. Restarting...
```

Issue solved: artifacts of the cached computation graphs on distributed workers are now better managed

New features

User Interface – C++ code inclusion

- Distributed RDF is fully Pythonic
- What if I have some C++ **functions in a header file?**

Before

```python
def load_header():
    """Load C++ helper functions. Works for both local and distributed execution."""
    try:
        # when using distributed RDataFrame 'my_header.h' is copied to the
        # local_directory of every worker (via `distribute_unique_paths`)
        localdir = get_worker().local_directory
        cpp_header = Path(localdir) / "my_header.h"
    except ValueError:
        # include the local execution as well
        cpp_header = "my_header.h"

    ROOT.gInterpreter.Declare(f'#include "{str(cpp_header)}"')


ROOT.RDF.Experimental.Distributed.initialize(load_header)
```

Now

```python
ROOT.Distributed.DistributeHeaders("my_header.h")
```

User Interface – C++ code inclusion

- What if I **I want to declare some C++ code?**

```
ROOT.Distributed.DeclareCppCode("""
        #ifndef MY_CODE
        #define My_CODE
        bool check_number_less_than_five(int num){
            return num < 5;
        }
        #endif
        """)

df = ROOT.RDataFrame(treeName, fileName, client)
df_filtered = df.Filter("check_number_less_than_five(rdfentry_)")
```

- What if I **I want to use shared libraries?**

```
ROOT.Distributed.DistributeHeaders("my_header.h")
ROOT.Distributed.DistributeSharedLibs("lib_my_header.so")
```

Input Data Sources

- Before 2024: TTree or empty data source
- In 2024: Introduction of RNTuple – see ACAT 2024 talk
- New addition: **RDatasetSpec**

```
meta = ROOT.RDF.Experimental.RMetaData()
meta.Add("meta_key", "meta_value")
```

```
mySample = ROOT.RDF.Experimental.RSample("mySampleName", treeName, fileName, meta)

spec = ROOT.RDF.Experimental.RDatasetSpec()
spec.AddSample(mySample)

df = ROOT.RDataFrame(spec, executor=daskclient)
```

- **In progress:** implement FromSpec functionality for DistRDF
  - Create an RDataFrame from a JSON specification file

Use of Pythonisations

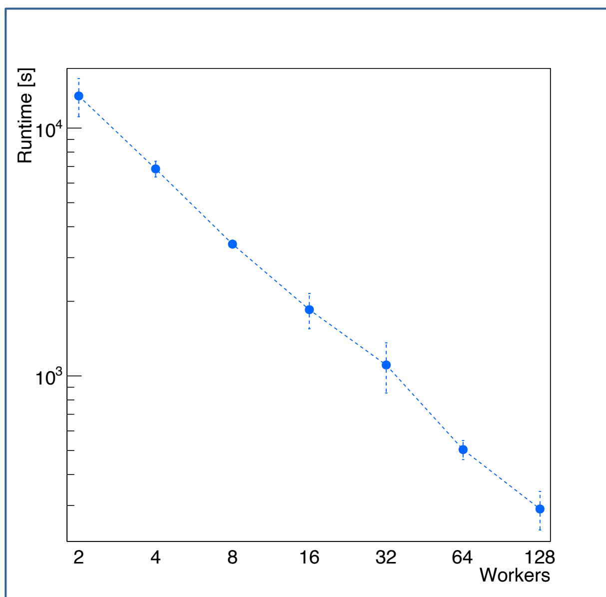
- More Pythonic ROOT → DistRDF analysis much easier
- For example, **background estimation using BDT** in Analysis Grand Challenge
  → pre-trained XGBoost model files
  - How to easily use those in RDF?
  - **Before**: external C++ class needed (FastForest)
  - **Now**: easily save XGBoost models into ROOT files and use those further with TMVA's RBDT, see ROOT AGC repository for more details

```
from xgboost import XGBClassifier

myBdt = XGBClassifier()
myBdt.load_model(f"myModel.json")
ROOT.TMVA.Experimental.SaveXGBoost(myBdt, "myBdt", "myModel.root", num_inputs=num_inputs)
```

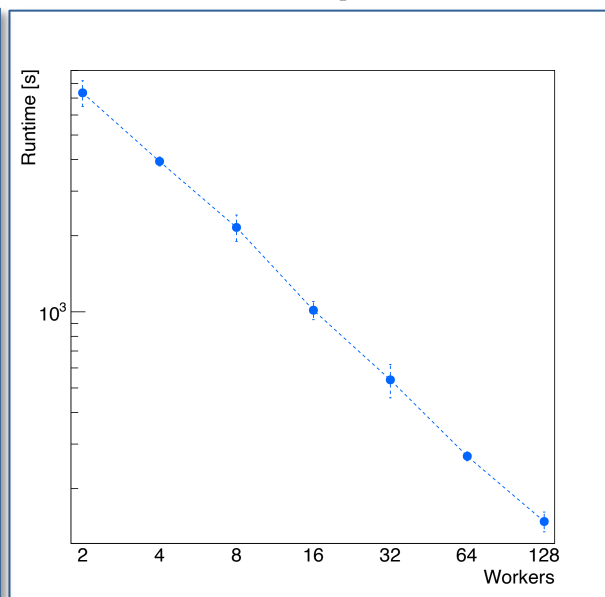
Analysis Facilities

- AGC with the BDT inference
- **Leverage all mentioned improvements of DistRDF**

TTree



RNTuple



- Ideal scaling for both
- RNTuple: **1.5 – 2x** faster than TTree → with zero code change for the user

Other Analysis Facilities

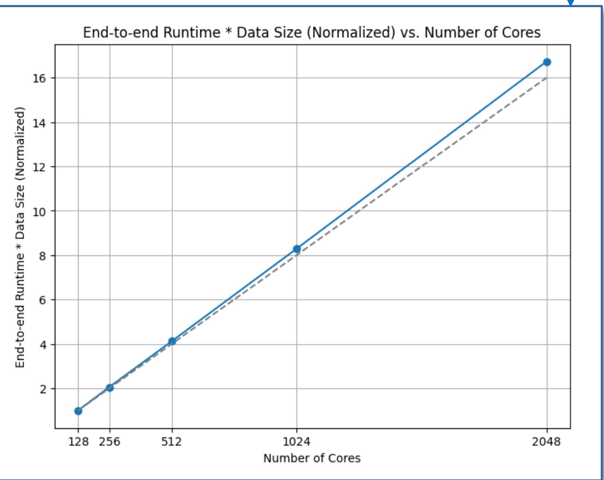
Over the years

Performed many tests on various AFs

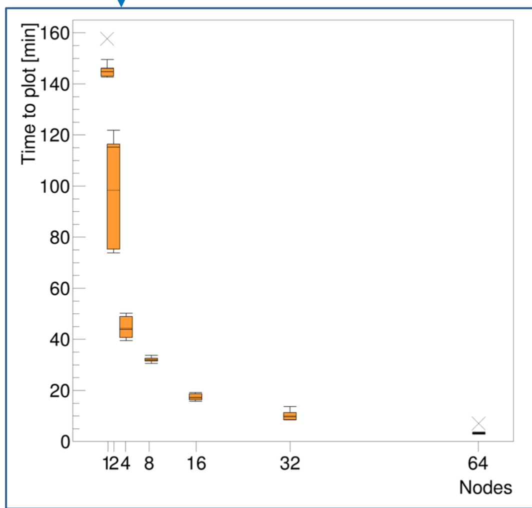
Jülich HPC, CERN HPC, also check: AWS, INFN

Recently

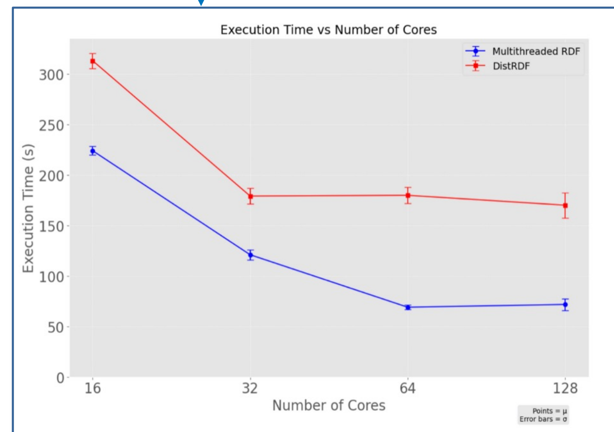
LUMI HPC, also check: ALPS HPC



Boulis J.



Padulano V.E. et al.



Mehrabi A.

More results at:
plenary talk on RNTuple

Next steps and conclusion

Next steps

- **Optimize RNTuple processing** post first RNTuple production release
- Generalize **RDatasetSpec** to accommodate complex workflows

Inputs and collaboration suggestions from users
(e.g. testing DistRDF in your AF)
are always very welcome!

Conclusion

Can we get out of **Experimental**

- User interface: **stabilised** and **unified**, with **easy inclusion of C++ code**
- Data input sources include **TTree, RNTuple** and **RDatasetSpec**
- The more **Pythonisations** in ROOT, the better the DistRDF
- DistRDF is performant in many different **Analysis Facilities**

Conclusion

Can we get out of Experimental **?**

YES!

Watch out for the next ROOT releases