Analysis Grand Challenge implementation with a Pythonic RDataFrame

V.E.Padulano, E.Guiraud, A.Falko

ROOT

Data Analysis Framework

https://root.cern

RDataFrame: declarative interface for data analysis

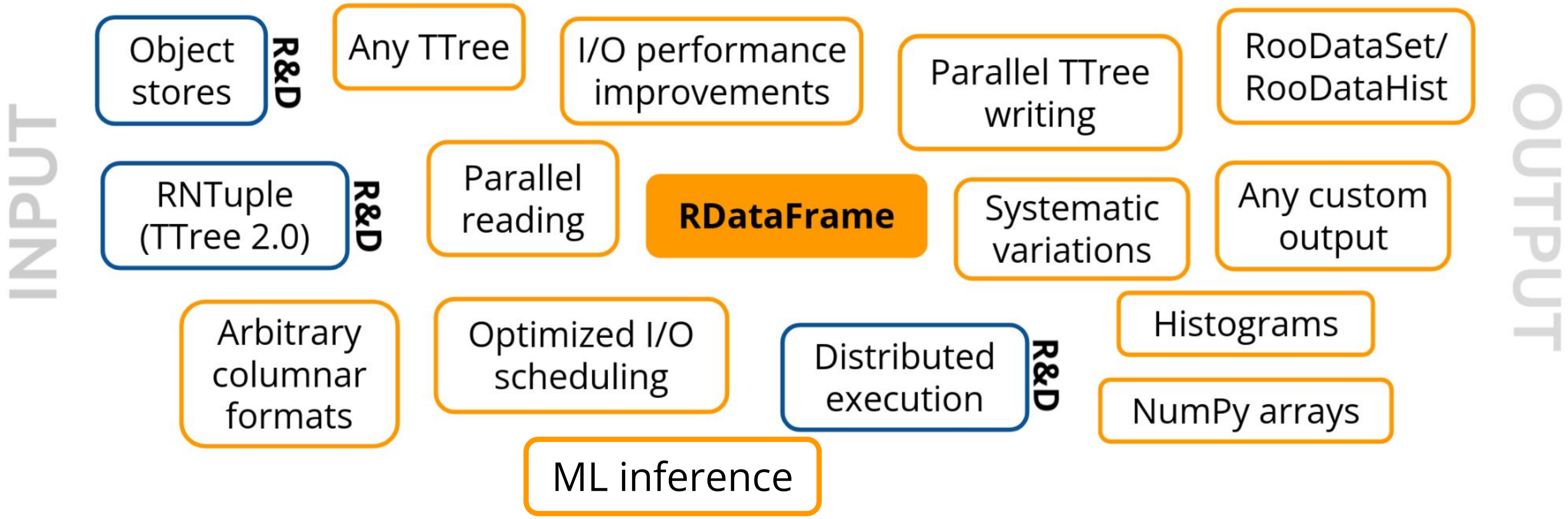
```
# Enable multithreading
ROOT.EnableImplicitMT()
df = ROOT.RDataFrame(dataset)

# Create observable
df = df.Define("my_px", "px[eta > 0]")

# Fill in a single pass
h1 = df.Histo1D("px")
h1 = df.Histo1D("my_px")
```

ROOT>=6.14

Wide adoption (see RDF@ICHEP2022)

Improving with the community

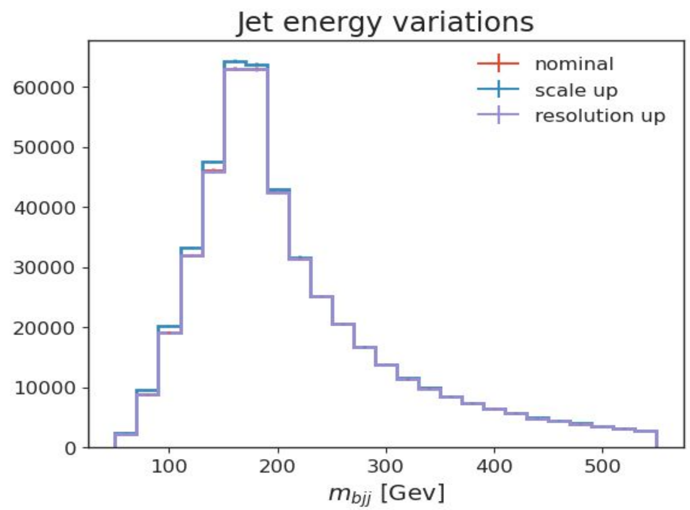
RDataFrame: entry point to modern ROOT

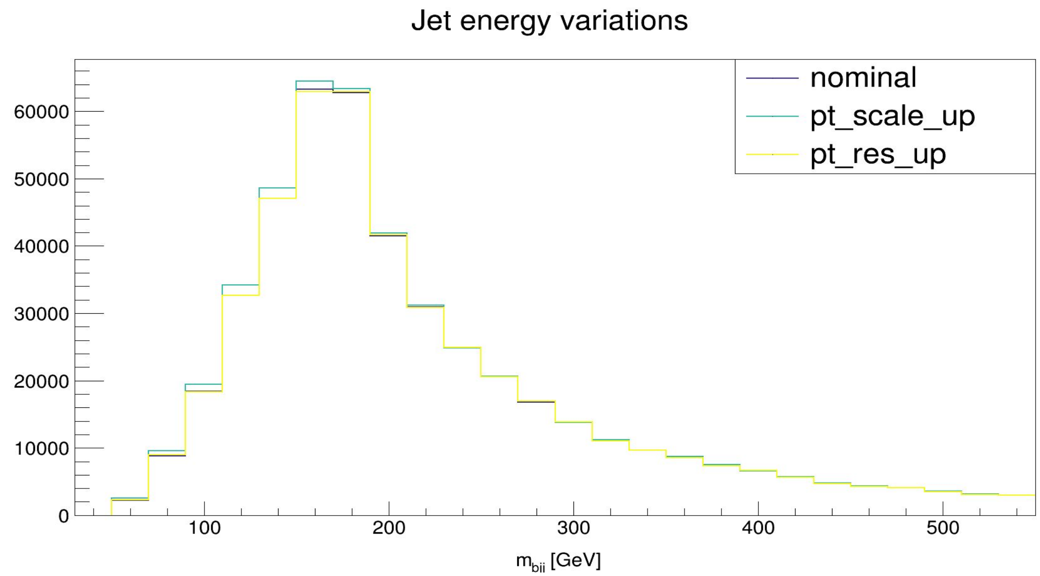


See [ACAT2022 talk]

RDataFrame for AGC

- Analysis Grand Challenge (**AGC**): realistic HEP analysis benchmarks with tools to execute them
- Using **RDataFrame** to implement **ttbar** example
- Reference benchmark snapshotted at commit c0b7e78
- Code available on github





Event selection

coffea

```
# pT > 25 GeV for leptons & jets
selected_electrons = events.electron[events.electron.pt > 25]
selected_muons = events.muon[events.muon.pt > 25]
jet_filter = events.jet.pt * events[pt_var] > 25  # pT > 25 GeV for jets (scaled by systematic variations)
selected_jets = events.jet[jet_filter]

# single lepton requirement
event_filters = ((ak.count(selected_electrons.pt, axis=1) + ak.count(selected_muons.pt, axis=1)) == 1)
# at least four jets
pt_var_modifier = events[pt_var] if "res" not in pt_var else events[pt_var][jet_filter]
event_filters = event_filters & (ak.count(selected_jets.pt * pt_var_modifier, axis=1) >= 4)
# at least one b-tagged jet ("tag" means score above threshold)
B_TAG_THRESHOLD = 0.5
event_filters = event_filters & (ak.sum(selected_jets.btag >= B_TAG_THRESHOLD, axis=1) >= 1)
```

RDataFrame

```
# event selection - the core part of the algorithm applied for both regions
# selecting events containing at least one lepton and four jets with pT > 25 GeV
# applying requirement at least one of them must be b-tagged jet (see details in the specification)
d = d.Define('electron_pt_mask', 'electron_pt>25').Define('muon_pt_mask', 'muon_pt>25').Define('jet_pt_mask', 'jet_pt>25')\
     .Filter('Sum(electron_pt_mask) + Sum(muon_pt_mask) == 1')\
     .Filter('Sum(jet_pt_mask) >= 4')\
     .Filter('Sum(jet_btag[jet_pt_mask]>=0.5)>=1')
```

Trijets

coffea

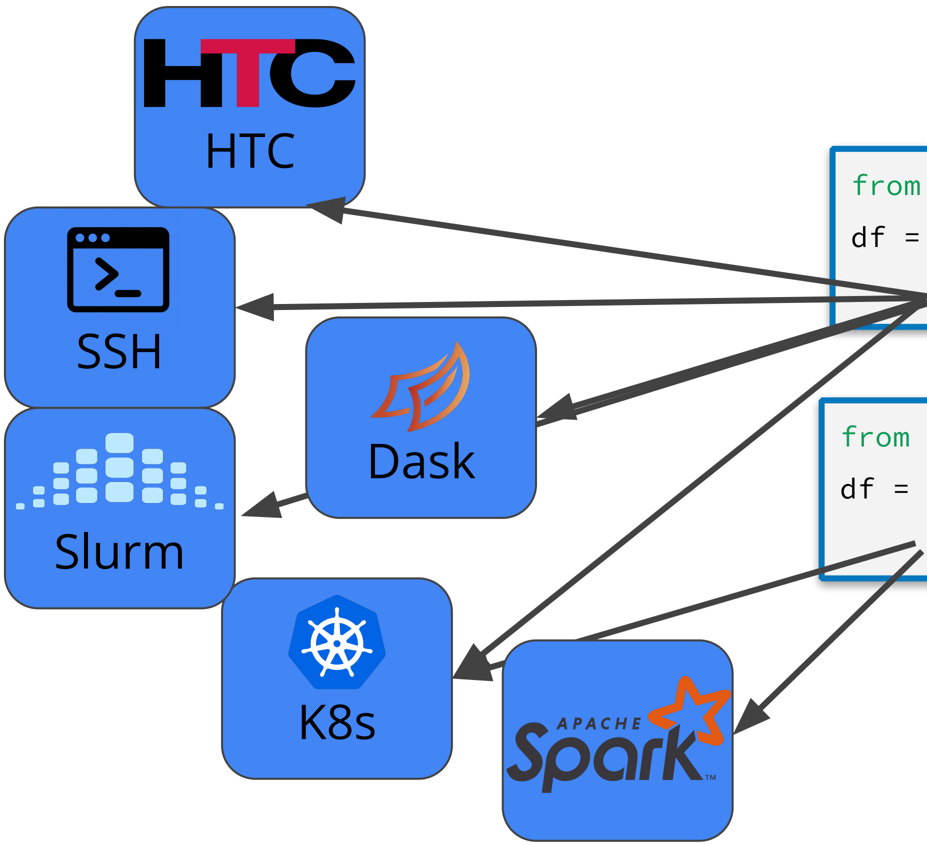
```
# reconstruct hadronic top as bjj system with largest pT
# the jet energy scale / resolution effect is not propagated to this observable at the moment
trijet = ak.combinations(selected_jets_region, 3, fields=["j1", "j2", "j3"])  # trijet candidates
trijet["p4"] = trijet.j1 + trijet.j2 + trijet.j3  # calculate four-momentum of tri-jet system
```

RDataFrame

```
# building trijet combinations
fork = fork.Define('trijet',
                   'ROOT::VecOps::Combinations(jet_pt[jet_pt_mask],3)'
                   ).Define('ntrijet', 'trijet[0].size()')

# assigning four-momentums to each trijet combination
fork = fork.Define('trijet_p4',
                   'ROOT::VecOps::RVec<ROOT::Math::PxPyPzMVector> trijet_p4(ntrijet);' +
                   'for (int i = 0; i < ntrijet; ++i) {' +
                   'int j1 = trijet[0][i]; int j2 = trijet[1][i]; int j3 = trijet[2][i];' +
                   'trijet_p4[i] = jet_p4[j1] + jet_p4[j2] + jet_p4[j3];' +
                   '}' +
                   'return trijet_p4;'
                   )
```

Distributing the AGC

RDataFrame **distributed**: seamlessly leverage clusters




```python
from dask.distributed import Client
df = RDataFrame('treename', 'filename.root',
        daskclient = Client('tcp://hostname:port'))
```

```python
from pyspark import SparkContext
df = RDataFrame('treename', 'filename.root',
        sparkcontext = SparkContext('spark://IP:PORT'))
```

Distributing the AGC

```python
def create_connection(nodes, ncores) -> Client:
    parsed_nodes = nodes.split(',')
    scheduler = parsed_nodes[:1]
    workers = parsed_nodes[1:]

    print("List of nodes: scheduler ({}) and workers ({})".format(scheduler, workers))

    cluster = SSHCluster(scheduler + workers,
              connect_options={ "known_hosts": None },
              worker_options={ "nprocs" : ncores, "nthreads": 1, "memory_limit" : "32GB"
    client = Client(cluster)

    return client
```

```python
def create_connection(_, ncores):
    cluster = LocalCluster(n_workers=ncores, threads_per_worker=1, processes=True)
    client = Client(cluster)
    return client
```

```python
def main():

    with create_connection(ARGS.nodes, ARGS.ncores) as conn:
        for _ in range(ARGS.ntests):
            results, runtime = analyse(conn)
```

Distributing the AGC

Hardware setup:

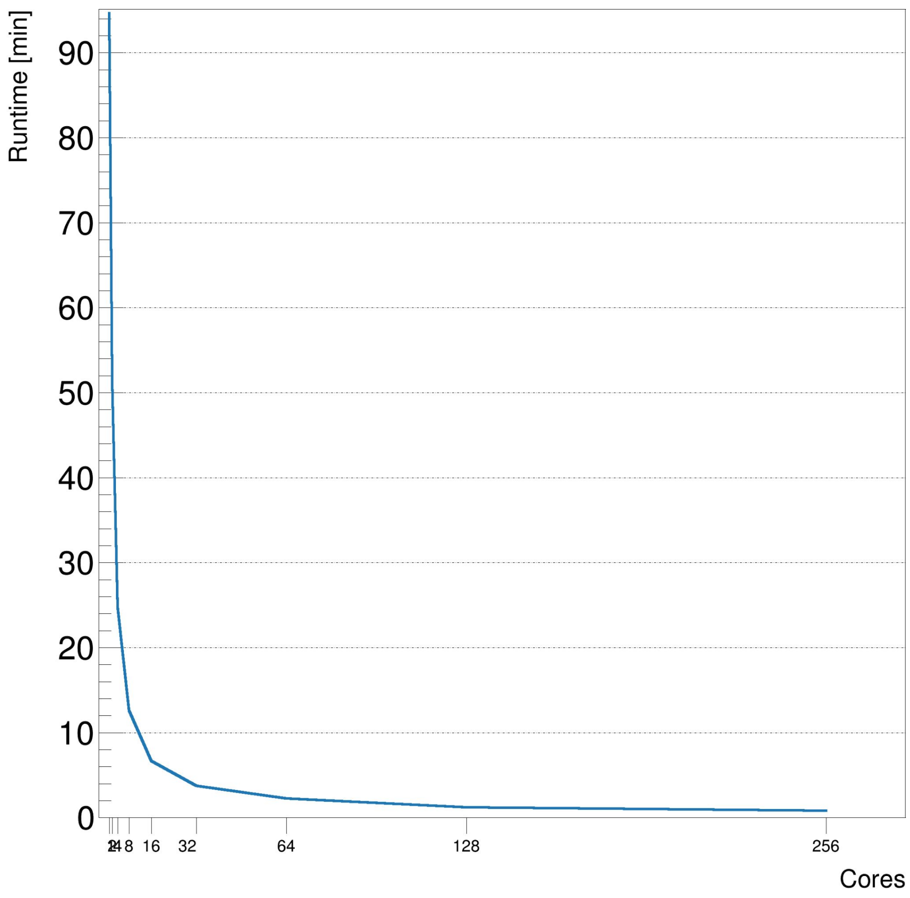
- 32 physical cores per node (no hyperthreading)
- 512 GB RAM
- 100 Gbps network
- Managed through Slurm

Config:

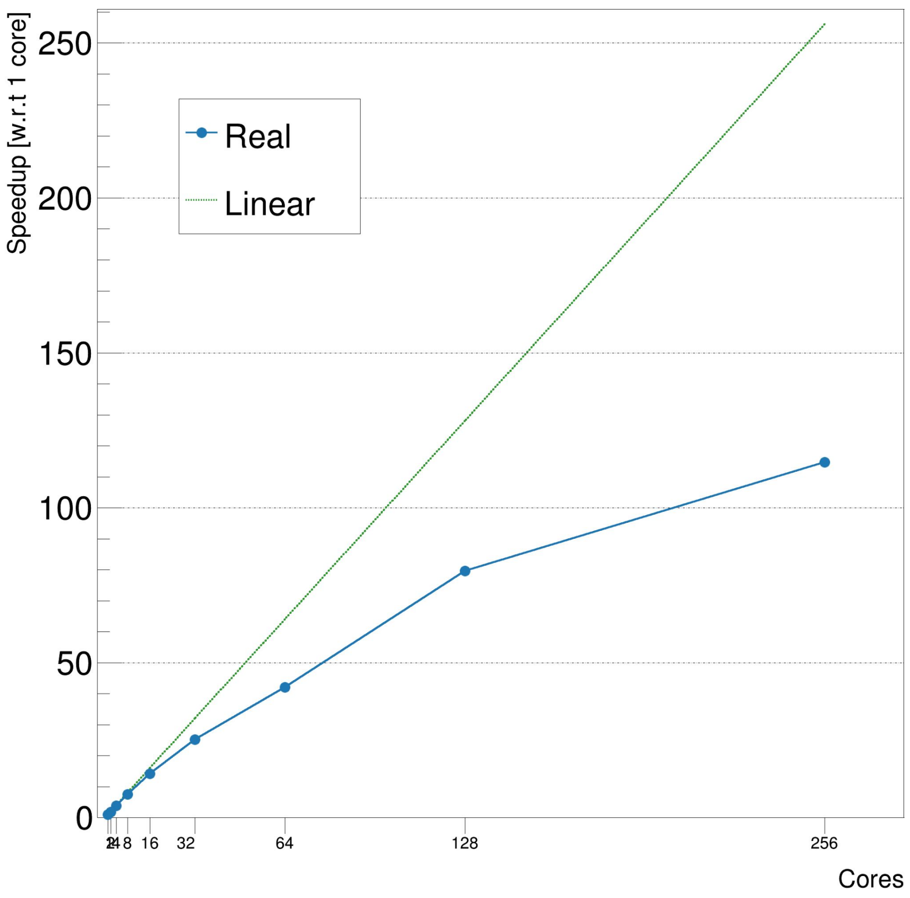
- Using from 1 to 8 computing nodes, exclusive access
- Requesting 1 extra node for the scheduler

Distributing the AGC

end-to-end runtime



Speedup



More performance studies in [Andrea Sciabà's talk]

Pythonizing the interface

- RDataFrame offers the **flexibility** to express virtually **any** HEP **analysis**
- This includes allowing any **C++** code to be executed through the API
- Leads to language **overlaps** when using Python
- **WIP:** enable **pure Python** interface through **numba** JIT

Pythonizing the interface

Simple cases: directly pass Python lambdas

```
# event selection - the core part of the algorithm applied for both regions
# selecting events containing at least one lepton and four jets with pT > 25 GeV
# applying requirement at least one of them must be b-tagged jet (see details in the specification)
d = d.Define('electron_pt_mask', lambda electron_pt: electron_pt > 25)\
     .Define('muon_pt_mask', lambda muon_pt: muon_pt > 25)\
     .Define('jet_pt_mask', lambda jet_pt: jet_pt > 25)\
     .Filter(lambda electron_pt_mask, muon_pt_mask: numpy.sum(electron_pt_mask) + numpy.sum(muon_pt_mask) == 1)\
     .Filter(lambda jet_pt_mask: numpy.sum(jet_pt_mask) >= 4)\
     .Filter(lambda jet_btag, jet_pt_mask: numpy.sum(jet_btag[jet_pt_mask] >= 0.5) >= 1)
```

Difficult cases: leverage cppyy wrappers

```
# building trijet combinations
fork = fork.Define('trijet', combinations, ["jet_pt", "jet_pt_mask"])\
           .Define('ntrijet', lambda trijet: len(trijet[0]), ["trijet"])

# assigning four-momentums to each trijet combination
fork = fork.Define('trijet_p4',
                   build_trijetp4,
                   ["jet_p4", "trijet", "ntrijet"]
                   )
```

Pythonizing the interface

Simple cases: directly pass Py

```
# event selection - the core part of the algorithm ap
# selecting events containing at least one lepton and
# applying requirement at least one of them must be b
d = d.Define('electron_pt_mask', lambda electron_pt:
     .Define('muon_pt_mask', lambda muon_pt: muon_pt > 25)\
     .Define('jet_pt_mask', lambda jet_pt: jet_pt > 25)\
     .Filter(lambda electron_pt_mask, muon_pt_mask: numpy.sum(electron_pt_mask) + numpy.sum(muon_pt_mask) == 1)\
     .Filter(lambda jet_pt_mask: numpy.sum(jet_pt_mask) >= 4)\
     .Filter(lambda jet_btag, jet_pt_mask: numpy.sum(jet_btag[jet_pt_mask] >= 0.5) >= 1)
```

Limit 1:
We can be as good as numba

Difficult cases: leverage cppyy wrappers

```
# building trijet combinations
fork = fork.Define('trijet', combinations, ["jet_pt", "jet_pt_mask"])\
           .Define('ntrijet', lambda trijet: len(trijet[0]), ["trijet"])

# assigning four-momentums to each trijet combination
fork = fork.Define('trijet_p4',
                   build_trijetp4,
                   ["jet_p4", "trijet", "ntrijet"]
                   )
```

Pythonizing the interface

Simple cases: directly pass Py

```
# event selection - the core part of the algorithm ap
# selecting events containing at least one lepton and
# applying requirement at least one of them must be b
d = d.Define('electron_pt_mask', lambda electron_pt:
     .Define('muon_pt_mask', lambda muon_pt: muon_pt > 25)\
```

```
_pt_mask) + numpy.sum(muon_pt_mask) == 1)\
```

```
sk] >= 0.5) >= 1)
```

Limit 1:
We can be as good as numba

Limit 2:
Support for fundamental types
and arrays thereof (through RVec<T>)
no RVec<RVec<...>>

D ers

```
fork = fork.Define('trijet', combinations, ["jet_pt", "jet_pt_mask"])\
           .Define('ntrijet', lambda trijet: len(trijet[0]), ["trijet"])

# assigning four-momentums to each trijet combination
fork = fork.Define('trijet_p4',
                   build_trijetp4,
                   ["jet_p4", "trijet", "ntrijet"]
                   )
```

Pythonizing the interface

Simple cases: directly pass Python lambdas

```
# event selection - the core part of the algorithm ap
# selecting events containing at least one lepton and
# applying requirement at least one of them must be b
d = d.Define('electron_pt_mask', lambda electron_pt:
     .Define('muon_pt_mask', lambda muon_pt: muon_pt > 25)\
pt_mask) + numpy.sum(muon_pt_mask) == 1)\
sk] >= 0.5) >= 1)
```

Limit 1:
We can be as good as numba

Limit 2:
Support for fundamental types
and arrays thereof (through RVec<T>)
no RVec<RVec<...>>

D ... ers

```
fork = fork.Define('trijet', combinations, ["jet_pt", "jet_pt_mask"])\
           .Define('ntrijet', la

# assigning four-momentums to ea
fork = fork.Define('trijet_p4',
                   build_trijetp
                   ["jet_p4", "t
                   )
```

Improvements happen transparently
e.g. cppyy<->numba (see ACAT2022), awkward<->numba (see CHEP2023)

Conclusions

- Implemented the **ttbar** example from **AGC** with **RDataFrame**
- Multithreading or distributed execution **just work**
- New Pythonizations shorten the interface gap