Analysis Grand Challenge implementation with a Pythonic RDataFrame

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

ROOT

Data Analysis Framework

https://root.cern

Introduction

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")
```

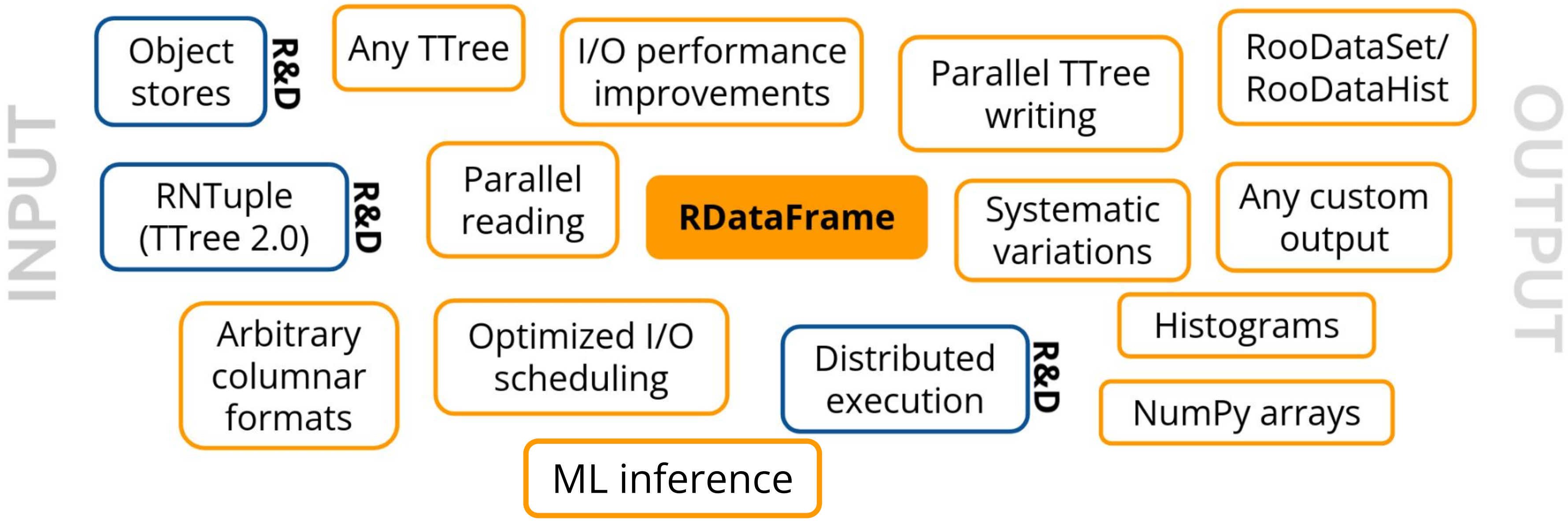
ROOT>=6.14

Wide adoption
(see RDF@ICHEP2022)

Improving with the community

Introduction

RDataFrame: entry point to modern ROOT



INPUT

Object stores R&D

Any TTree

I/O performance improvements

Parallel TTree writing

RooDataSet/ RooDataHist

RNTuple (TTree 2.0) R&D

Parallel reading

RDataFrame

Systematic variations

Any custom output

Arbitrary columnar formats

Optimized I/O scheduling

Distributed execution R&D

Histograms

NumPy arrays

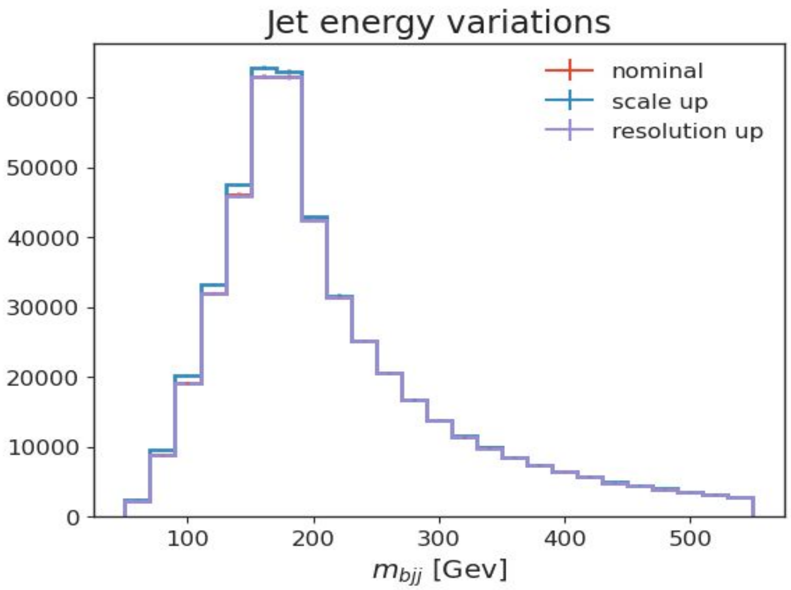
ML inference

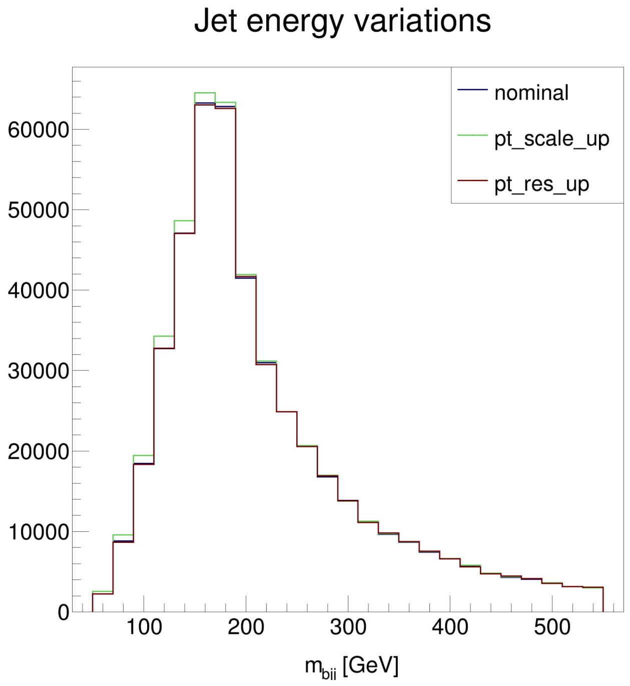
OUTPUT

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 tag v.0.1
- Code available on github

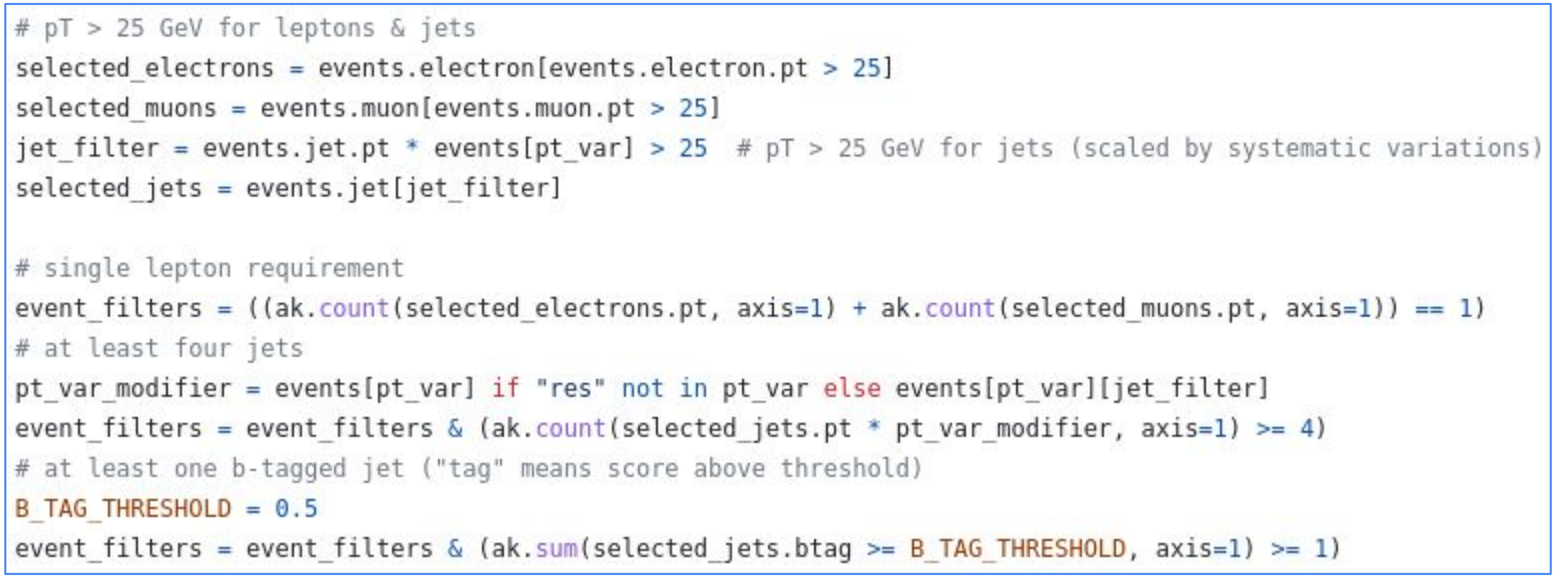




The translation

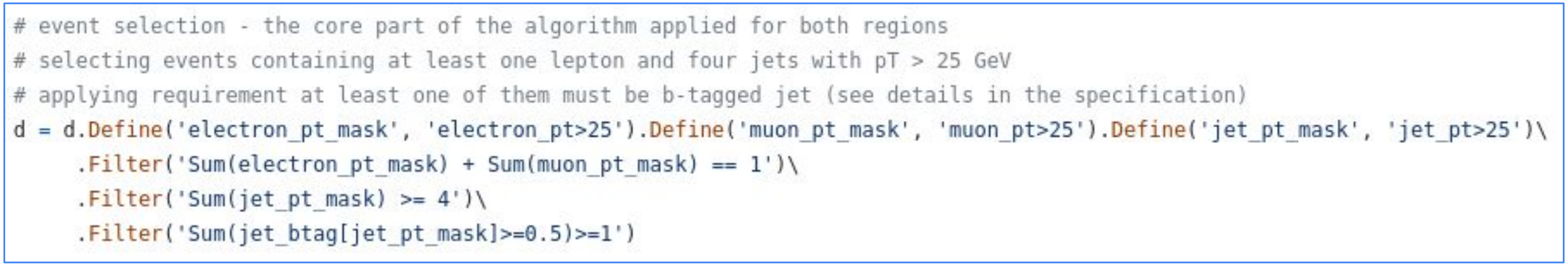
Event selection

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

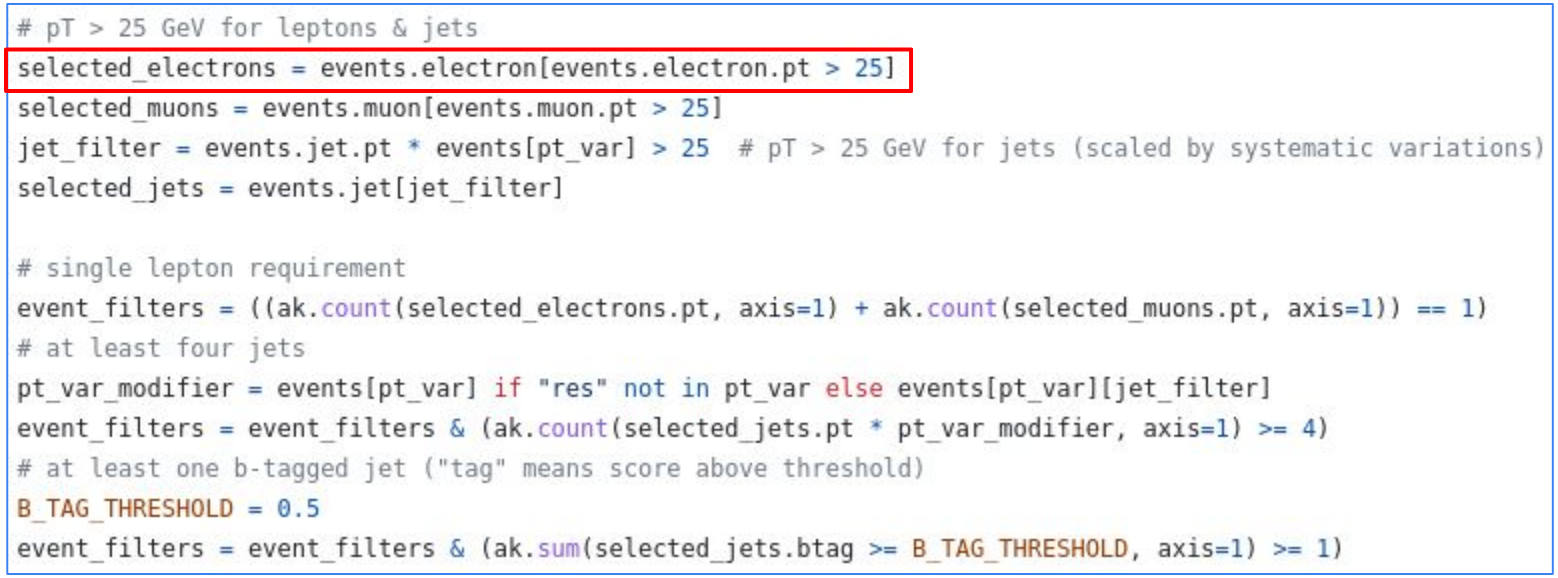


```
# event selection - the core part of the algorithm applied for both regions
# selecting events containing at least one lepton and four jets with pT > 25 GeV
# applying requirement at least one of them must be b-tagged jet (see details in the specification)
d = d.Define('electron_pt_mask', 'electron_pt>25').Define('muon_pt_mask',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

The translation

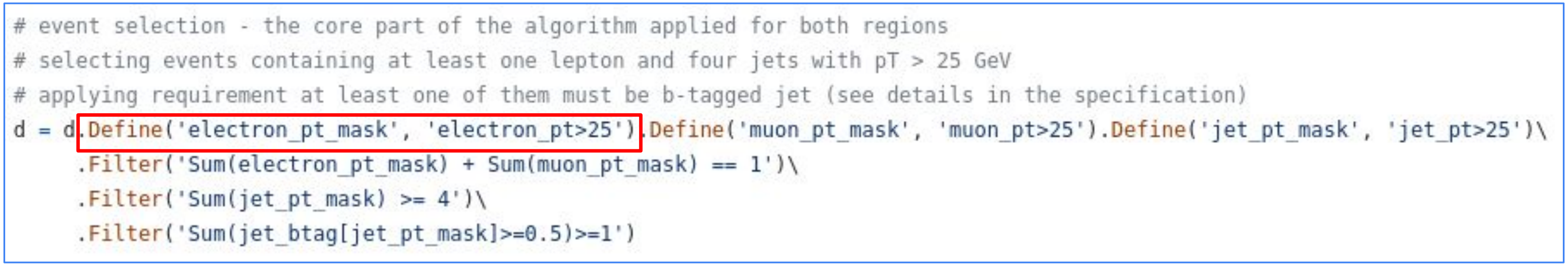
Event selection

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

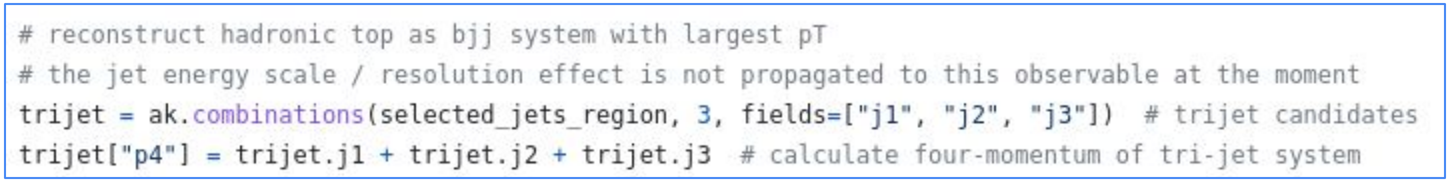


```
# event selection - the core part of the algorithm applied for both regions
# selecting events containing at least one lepton and four jets with pT > 25 GeV
# applying requirement at least one of them must be b-tagged jet (see details in the specification)
d = d.Define('electron_pt_mask', 'electron_pt>25').Define('muon_pt_mask',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

The translation

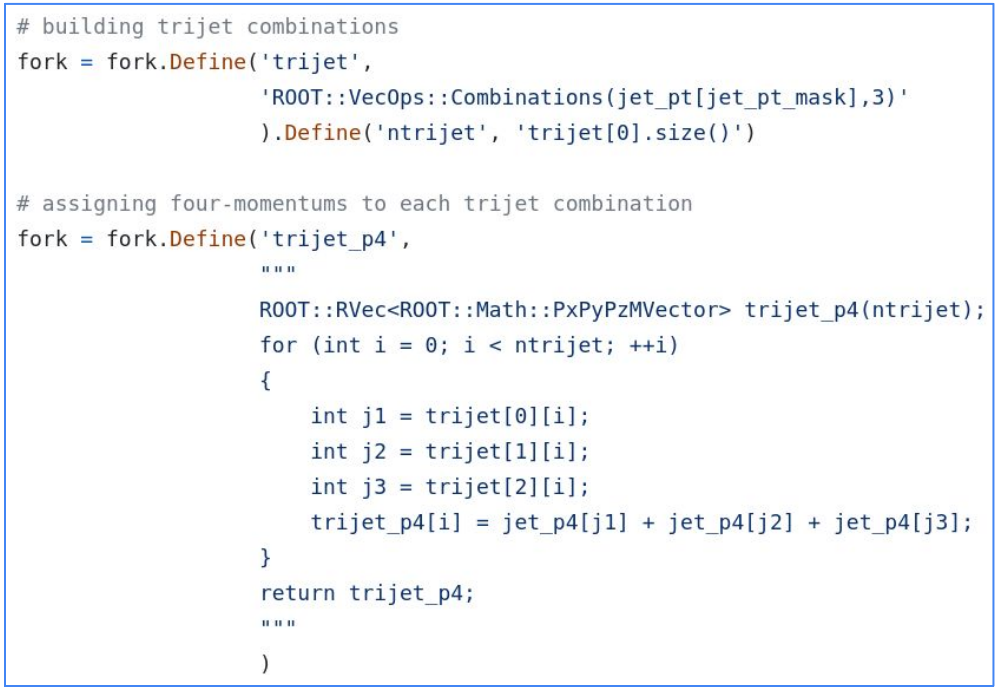
Trijets

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
```

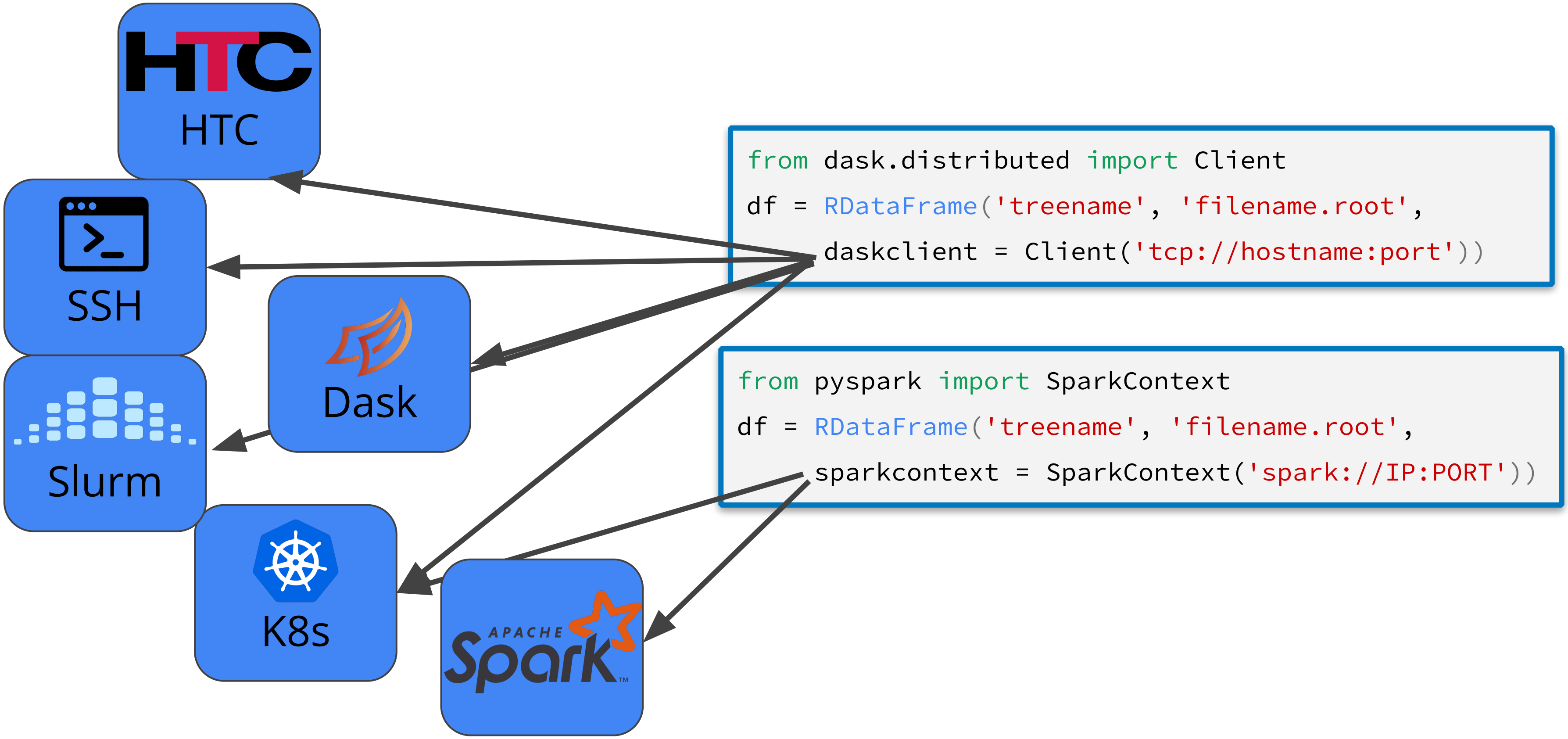
RDataFrame



```
# 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::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

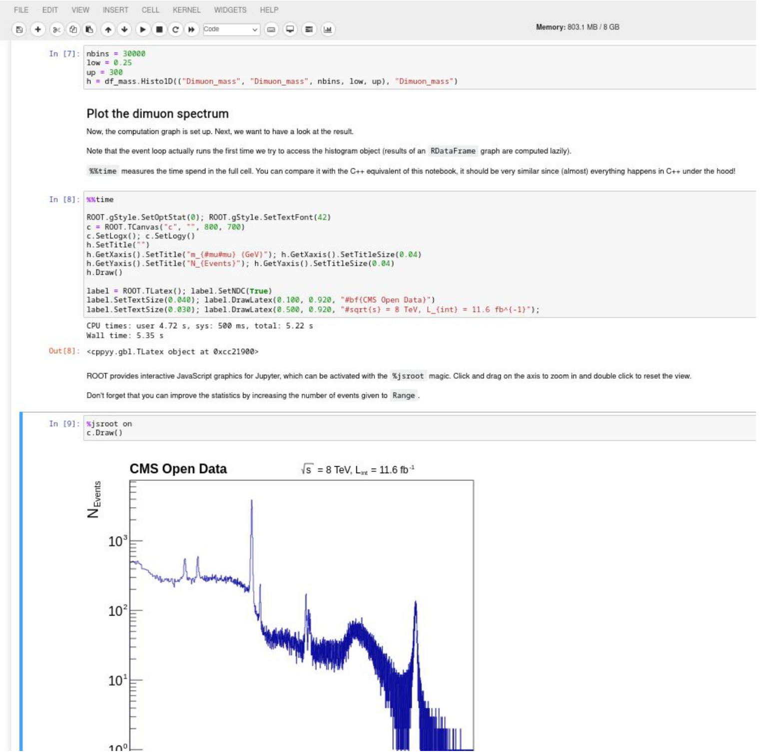


```python
from dask.distributed import Client
df = RDataFrame('treename', 'filename.root',
                daskclient = Client('tcp://hostname:port'))
```

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

Distributing the AGC

SWAN



```
In [7]: nbins = 30000
        low = 0.25
        up = 300
        h = df_mass.Histo1D(("Dimuon_mass", "Dimuon_mass", nbins, low, up), "Dimuon_mass")
```

Plot the dimuon spectrum

Now, the computation graph is set up. Next, we want to have a look at the result.

Note that the event loop actually runs the first time we try to access the histogram object (results of an `RDataFrame` graph are computed lazily).

`%%time` measures the time spend in the full cell. You can compare it with the C++ equivalent of this notebook, it should be very similar since (almost) everything happens in C++ under the hood!

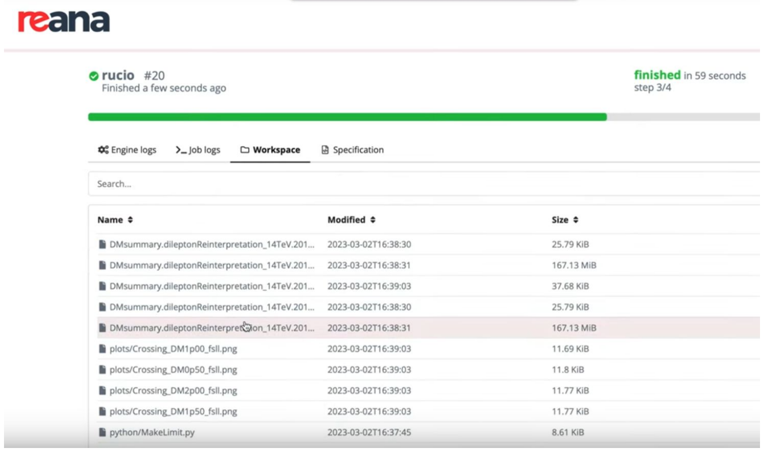
```
In [8]: %%time

        ROOT.gStyle.SetOptStat(0); ROOT.gStyle.SetTextFont(42)
        c = ROOT.TCanvas("c", "", 800, 700)
        c.SetLogx(); c.SetLogy()
        h.SetTitle("")
        h.GetXaxis().SetTitle("m_{#mu#mu} (GeV)"); h.GetXaxis().SetTitleSize(0.04)
        h.GetYaxis().SetTitle("N_{Events}"); h.GetYaxis().SetTitleSize(0.04)
        h.Draw()

        label = ROOT.TLatex(); label.SetNDC(True)
        label.SetTextSize(0.040); label.DrawLatex(0.100, 0.920, "#bf{CMS Open Data}")
        label.SetTextSize(0.030); label.DrawLatex(0.500, 0.920, "#sqrt{s} = 8 TeV, L_{int} = 11.6 fb^{-1}");

        CPU times: user 4.72 s, sys: 500 ms, total: 5.22 s
        Wall time: 5.35 s
Out[8]: <cppyy.gbl.TLatex object at 0xcc21900>
```

ROOT provides interactive JavaScript graphics for Jupyter, which can be activated with the `%jsroot` magic. Click and drag on the axis to zoom in and double click to reset the view.

Don't forget that you can improve the statistics by increasing the number of events given to `Range`.

```
In [9]: %jsroot on
        c.Draw()
```

VRE

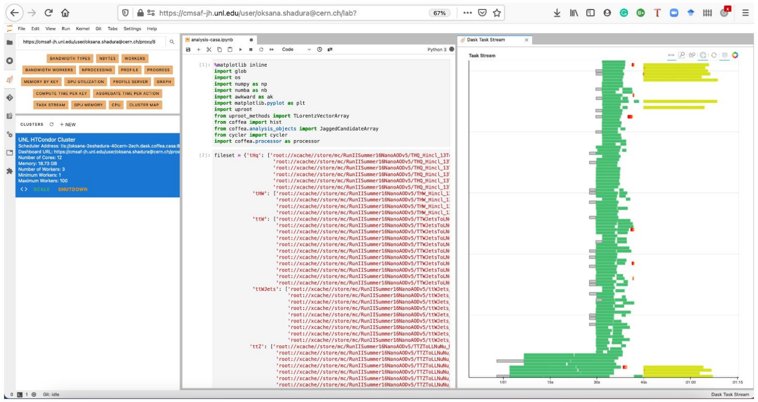


rucio #20
Finished a few seconds ago

finished in 59 seconds
step 3/4

Name	Modified	Size
DMsummary.dileptonReinterpretation_14TeV.201...	2023-03-02T16:38:30	25.79 KiB
DMsummary.dileptonReinterpretation_14TeV.201...	2023-03-02T16:38:31	167.13 MiB
DMsummary.dileptonReinterpretation_14TeV.201...	2023-03-02T16:39:03	37.68 KiB
DMsummary.dileptonReinterpretation_14TeV.201...	2023-03-02T16:38:30	25.79 KiB
DMsummary.dileptonReinterpretation_14TeV.201...	2023-03-02T16:38:31	167.13 MiB
plots/Crossing_DM1p00_fsll.png	2023-03-02T16:39:03	11.69 KiB
plots/Crossing_DM0p50_fsll.png	2023-03-02T16:39:03	11.8 KiB
plots/Crossing_DM2p00_fsll.png	2023-03-02T16:39:03	11.77 KiB
plots/Crossing_DM1p50_fsll.png	2023-03-02T16:39:03	11.77 KiB
python/MakeLimit.py	2023-03-02T16:37:45	8.61 KiB

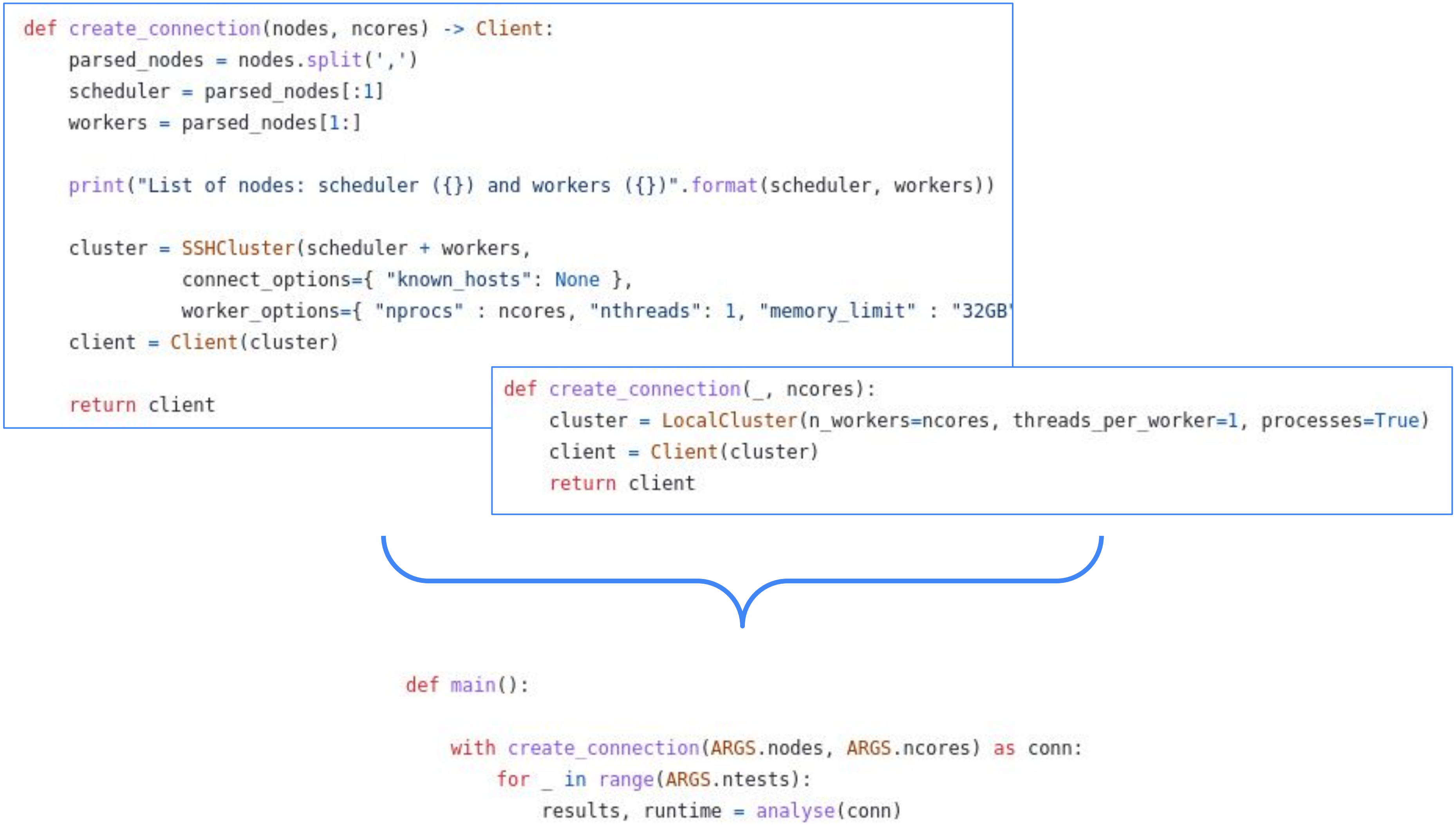
coffea-casa



```
[1]: %matplotlib inline
     import glob
     import os
     import numpy as np
     import numba as nb
     import awkward as ak
     import matplotlib.pyplot as plt
     import uproot
     from uproot_methods import TLorentzVectorArray
     from coffea import hist
     from coffea.analysis_objects import JaggedCandidateArray
     from cycler import cycler
     import coffea.processor as processor
```

```
[2]: fileset = {'tHq': ['root://xcache//store/mc/RunIISummer16NanoAODv5/THQ_Hincl_13T
                        'root://xcache//store/mc/RunIISummer16NanoAODv5/THQ_Hincl_13
                        'root://xcache//store/mc/RunIISummer16NanoAODv5/THQ_Hincl_13
                        'root://xcache//store/mc/RunIISummer16NanoAODv5/THQ_Hincl_13
                        'root://xcache//store/mc/RunIISummer16NanoAODv5/THQ_Hincl_13
                        'root://xcache//store/mc/RunIISummer16NanoAODv5/THQ_Hincl_13
                 'tHW': ['root://xcache//store/mc/RunIISummer16NanoAODv5/THW_Hincl_1
                         'root://xcache//store/mc/RunIISummer16NanoAODv5/THW_Hincl_1
                         'root://xcache//store/mc/RunIISummer16NanoAODv5/THW_Hincl_1
                         'root://xcache//store/mc/RunIISummer16NanoAODv5/THW_Hincl_1
                 'ttW': ['root://xcache//store/mc/RunIISummer16NanoAODv5/TTWJetsToLN
                         'root://xcache//store/mc/RunIISummer16NanoAODv5/TTWJetsToLN
                         'root://xcache//store/mc/RunIISummer16NanoAODv5/TTWJetsToLN
                         'root://xcache//store/mc/RunIISummer16NanoAODv5/TTWJetsToLN
                         'root://xcache//store/mc/RunIISummer16NanoAODv5/TTWJetsToLN
                         'root://xcache//store/mc/RunIISummer16NanoAODv5/TTWJetsToLN
                         'root://xcache//store/mc/RunIISummer16NanoAODv5/TTWJetsToLN
```

Distributing the AGC

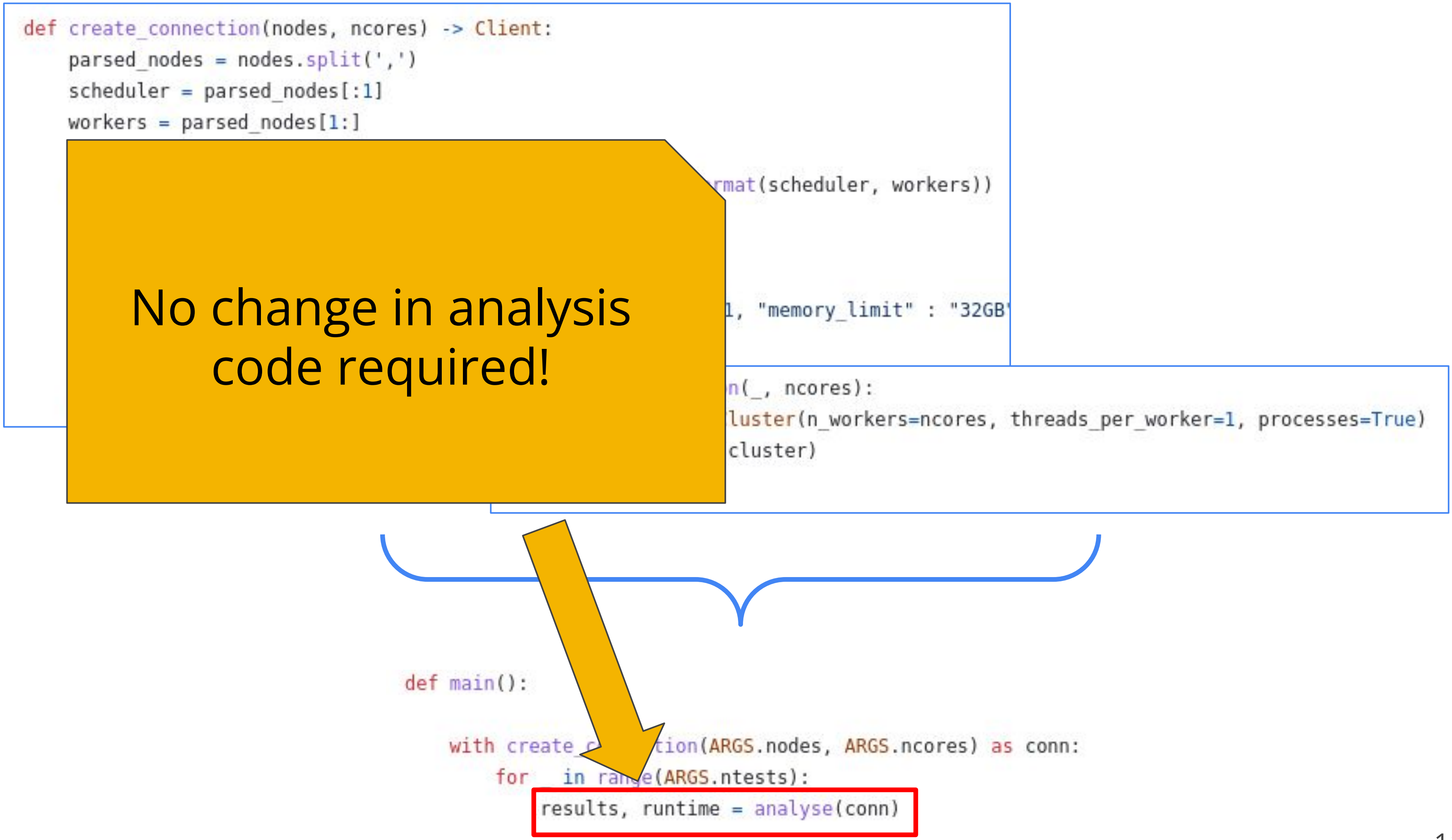


```python
def create_connection(nodes, ncores) -> Client:
    parsed_nodes = nodes.split(',')
    scheduler = parsed_nodes[:1]
    workers = parsed_nodes[1:]

    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



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

rmat(scheduler, workers))

1, "memory_limit" : "32GB"
```

```
n(_, ncores):
luster(n_workers=ncores, threads_per_worker=1, processes=True)
cluster)
```

No change in analysis code required!

```
def main():

    with create_c     tion(ARGS.nodes, ARGS.ncores) as conn:
        for _ in ran e(ARGS.ntests):
            results, runtime = analyse(conn)
```

Distributing the AGC

Hardware setup:

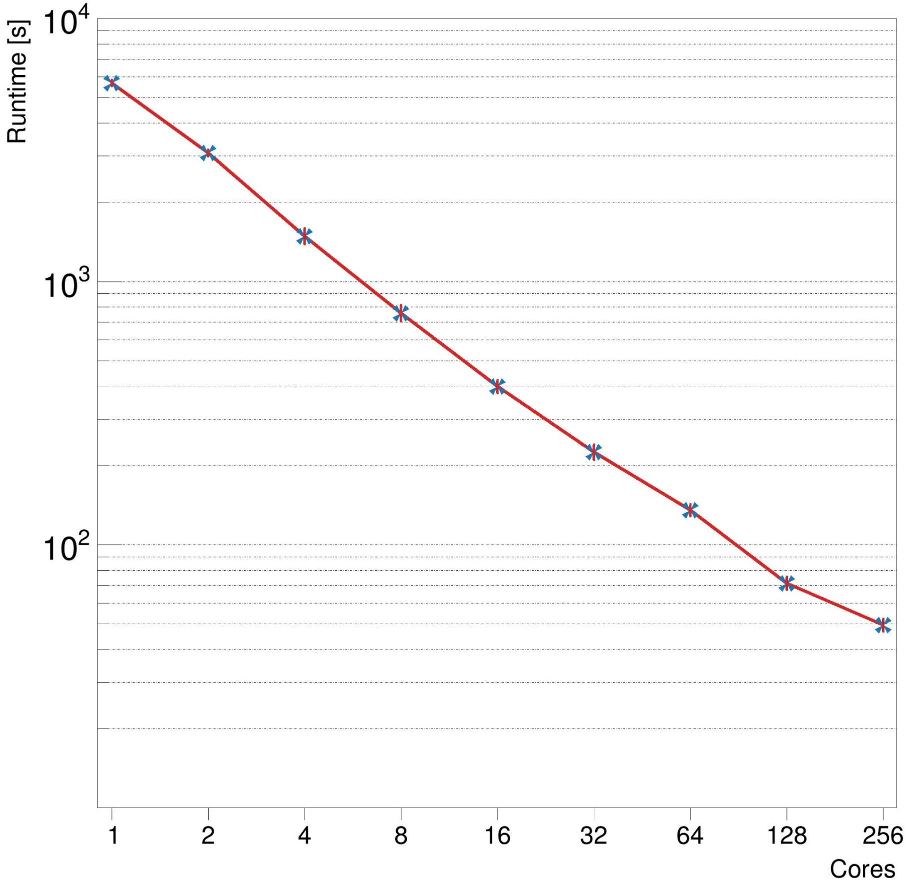
- 32 physical cores per node (no hyperthreading)
- 512 GB RAM
- 100 Gbps network
- Managed through Slurm

Config:

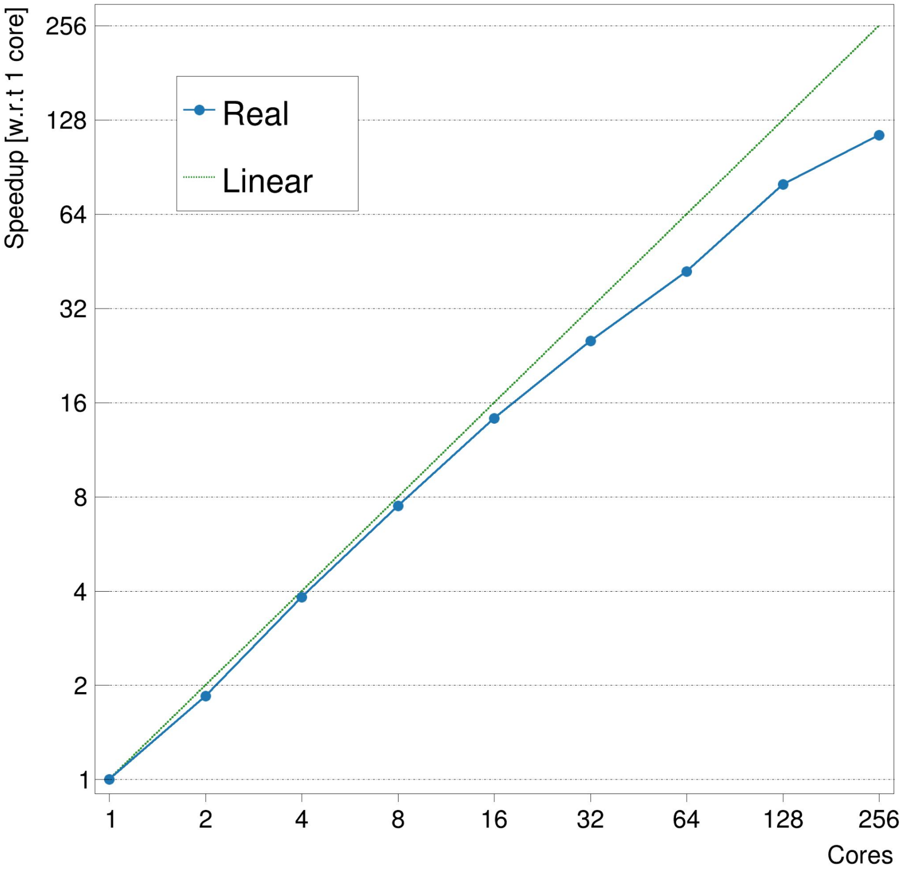
- Using from 1 to 8 computing nodes, exclusive access
- Requesting 1 extra node for the scheduler

Distributing the AGC

End-to-end runtime



Speedup



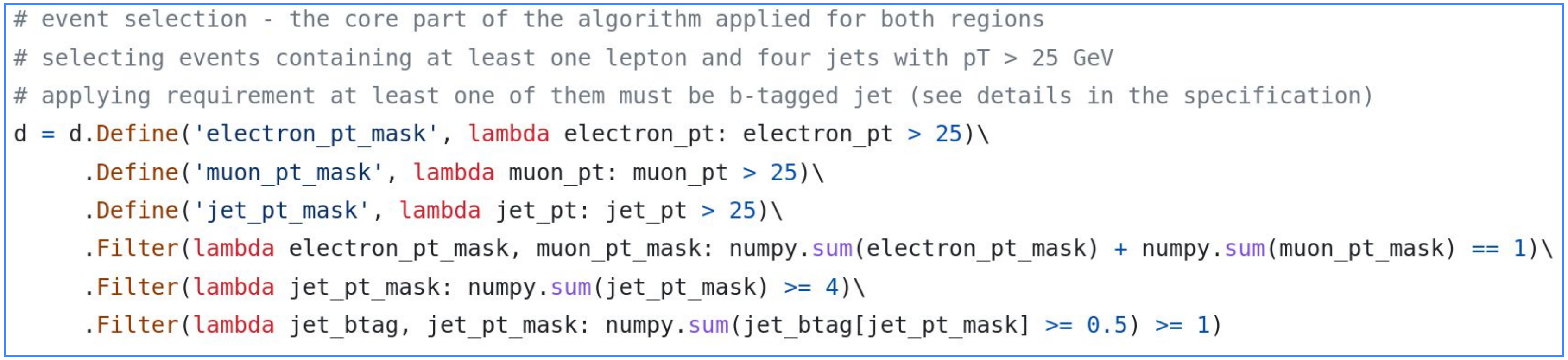
More performance studies in [Andrea Sciabà's talk]

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

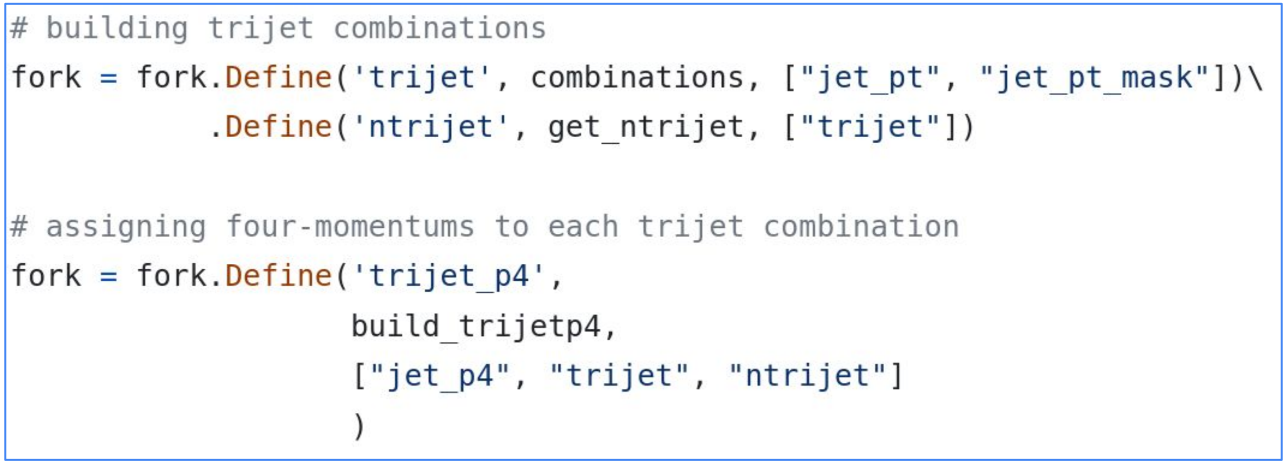
Pythonizing the interface

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)
```

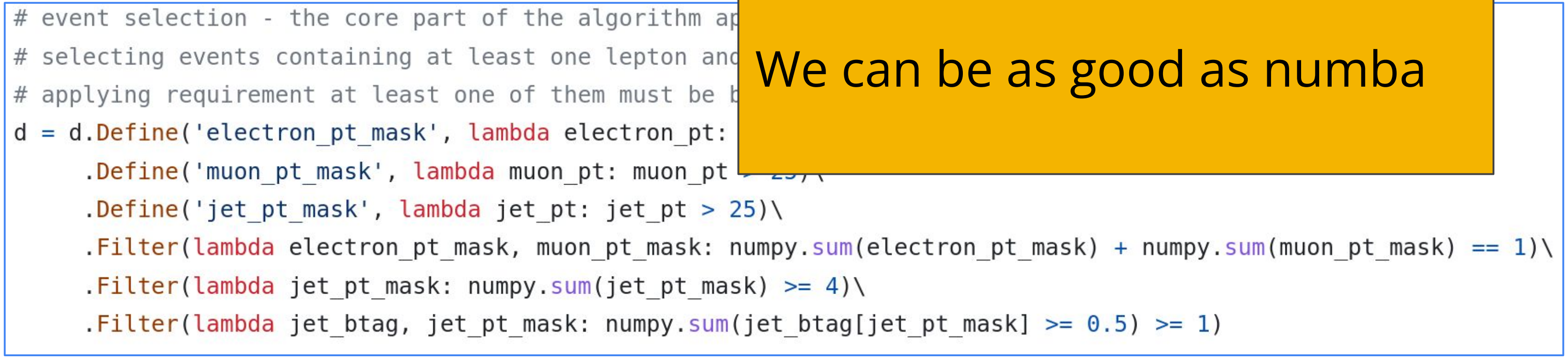
Difficult cases: leverage cppyy wrappers



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

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

Pythonizing the interface

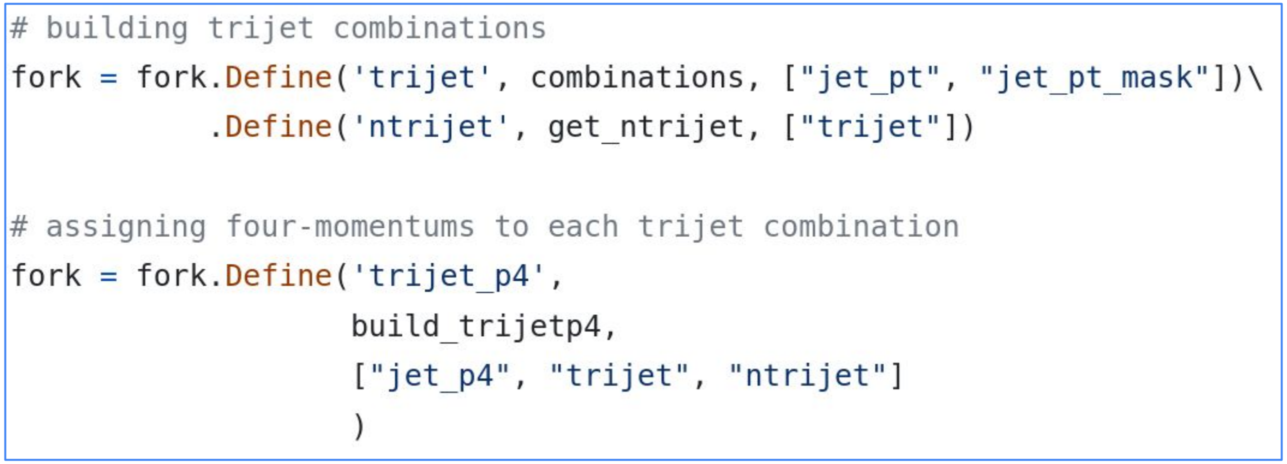
Simple cases: directly pass Py



```
# 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
     .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)
```

We can be as good as numba

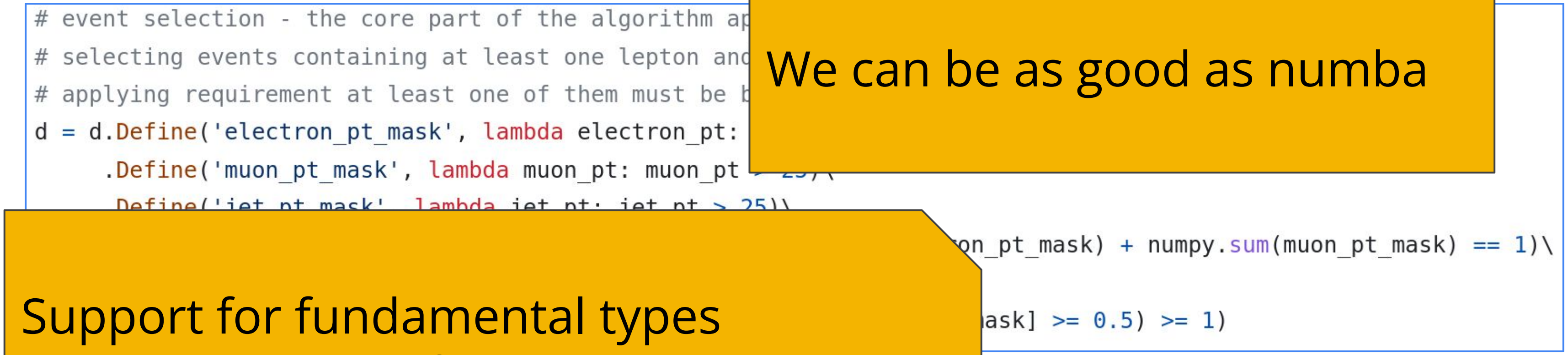
Difficult cases: leverage cppyy wrappers



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



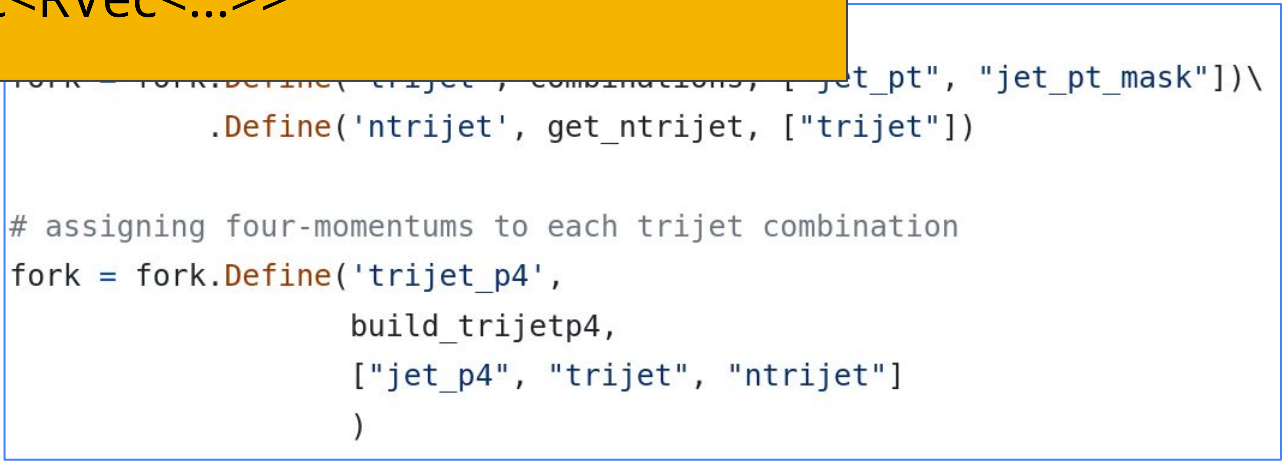
```
# event selection - the core part of the algorithm ap
# selecting events containing at least one lepton and
# applying requirement at least one of them must be b
d = d.Define('electron_pt_mask', lambda electron_pt:
     .Define('muon_pt_mask', lambda muon_pt: muon_pt > 25)\
     .Define('jet_pt_mask',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

                                                                     ask] >= 0.5) >= 1)
```

We can be as good as numba

D ers

Support for fundamental types
and arrays thereof (through RVec<T>)

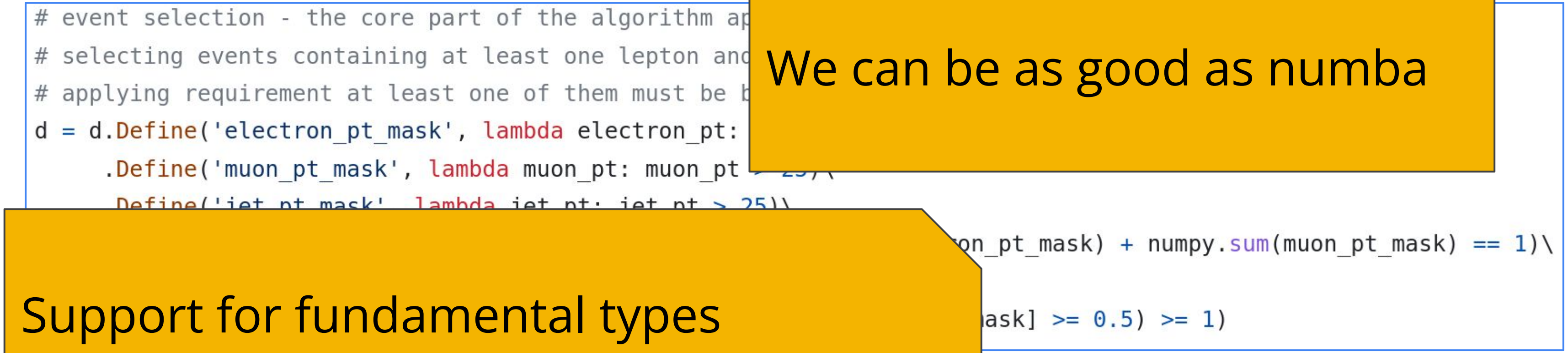
No RVec<RVec<...>>



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



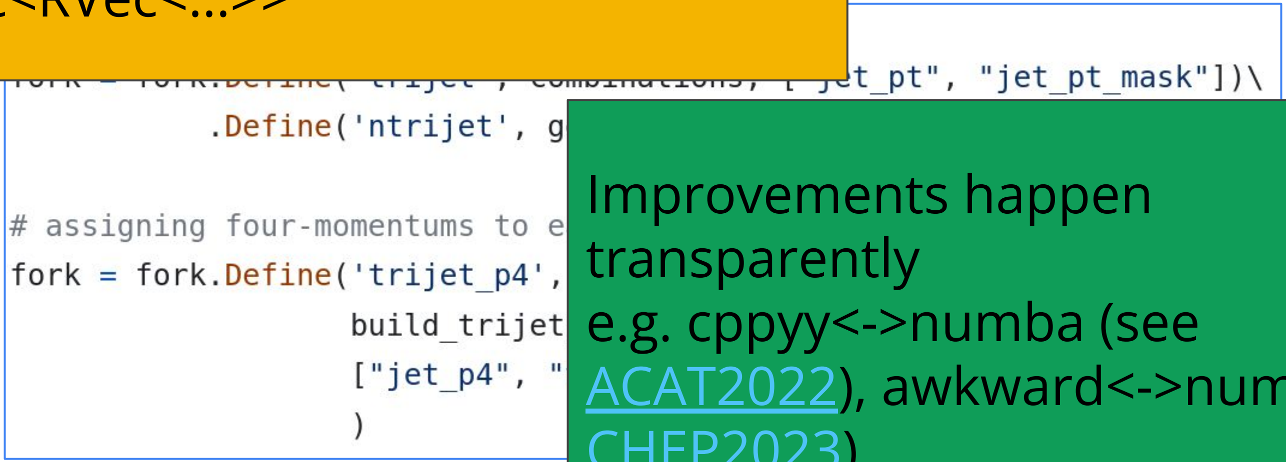
```
# event selection - the core part of the algorithm ap
# selecting events containing at least one lepton and
# applying requirement at least one of them must be b
d = d.Define('electron_pt_mask', lambda electron_pt:
     .Define('muon_pt_mask', lambda muon_pt: muon_pt
     .Define('jet_pt_mask', lambda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

                                        ask] >= 0.5) >= 1)
```

We can be as good as numba

Support for fundamental types
and arrays thereof (through RVec<T>)

No RVec<RVec<...>>

D ers



```
                                         t_pt", "jet_pt_mask"])\
          .Define('ntrijet', g

# assigning four-momentums to e
fork = fork.Define('trijet_p4',
                   build_trijet
                   ["jet_p4", "
                   )
```

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

Questions?