COMPUTING METHODS IN HIGH ENERGY PHYSICS

S. Lehti

Helsinki Institute of Physics

Spring 2024

Columnar processing

Introduction

Traditionally high energy physics data are analysed with event loops. This means processing one event at a time, making decision whether to keep or drop the event, and filling histograms. When the amount of data increases, as is the case for Run3 at the LHC, we need more efficient ways to process the data. One way is to process vectors of data at the same time instead of just one event. This is called columnar processing.

Coffea

Coffea - Columnar Object Framework For Effective Analysis is a prototype framework containing basic tools and wrappers for a high energy physics analysis. Coffea is written in python, no need to compile C++. The python version is required to be 3.x. It makes use of uproot and awkward-array to provide an array-based syntax for manipulating HEP event data in an efficient and numpythonic way. Coffea analysis scales easily: from a testing on a laptop to a large multi-core server, computing clusters, and super-computers without the need to alter or otherwise adapt the analysis code itself. The links to the documentation and code are

https://coffeateam.github.io/coffea/index.html

https://github.com/Coffeateam/coffea

Columnar processing

Coffea can be installed on your machine, on a virtual environment, and docker images for running it in a singularity container are provided. One can also test it in a jupyter notepad.

Example: Installing coffea in a virtual environment

```
python -m venv MyVEnv
source MyVEnv/bin/activate
(MyVEnv) pip install coffea
```

Depending on the coffea version, one may need to install wheel, xrootd and numpy
As we are using a prototype, sometimes the version in the head doesnt work, and one has to revert to the last working version.

Example: Running coffea

```
class Analysis(processor.ProcessorABC):
  def __init__(self):
    self.histos = {}
    self.histos["h_mass"] = (
      hda.Hist.new
      .Reg(200, 0 ,200, name="x",
      label="x-axis")
      .Double()
    )

  def process(self, events):
    muons = ak.zip({"pt": events.Muon_pt,..},
      with_name="PtEtaPhiMCandidate",
      behavior=candidate.behavior,)
    cut = (ak.num(muons) == 2) &
    (ak.sum(muons.charge, axis=1) == 0)
    dimuon = muons[cut][:, 0] + muons[cut][:, 1]
    self.histos["h_mass"].fill(x=dimuon.mass)
    return self.histos

    events = NanoEventsFactory.from_root(
      {filename: "Events"},
      metadata={"dataset": "DoubleMuon"},
      schemaclass=BaseSchema,
    ).events()
    p = Analysis()
    out = p.process(events)

    (result,) = dask.compute(out)
```

Columnar processing

In the analysis class histograms are initialized. Events with two opposite sign muons are selected. The histogram is filled with the di-muon mass.
Results are returned with processor.Runner taking datasamples, tree and the analysis class as an argument. The runner is initialized with an executor, like IterativeExecutor, FuturesExecutor, DaskExecutor etc.
Histograms and counters can be saved into a root file with uproot

```
with uproot.recreate("output.root") as fOUT:
  fOUT[f'myHisto'] = result['myHisto']
```

Singularity

In the previous example we ran coffea in a virtual environment.

Another option is to use a singularity container. What is singularity? A singularity container is used to encapsulate all required software and dependencies for a workflow. This is extremely useful when trying to execute a particular workflow on different systems. Containers make compiling software on different compute platforms easy and effortless. Therefore singularity allows you to run coffea on other machines without having to worry about what is installed in the computing element. You can send your jobs to run in a batch system or it extends to even grid, your jobs can be sent to grid computing elements, and they run inside the encapsulated singularity container.

Columnar processing

Coffea has a pre-built docker image available at docker://coffeateam/coffea-dask. The amount of downloads per day is limited, so if you want to make your coffea job parallel, it is better to create the singularity image yourself, and copy the image file to the computing elements. How to create the singularity image file (sif):

```
singularity build coffea.sif
  docker://coffeateam/coffea-dask
```

You can also add missing stuff, if needed, into the sif file by

```
singularity shell coffea.sif
singularity> pip install ..
```

Escape from the singularity shell with ctrl-D.

How to run:

```
  singularity run -i -B ${PWD}:/work coffea.sif
bash -c /work/job.sh
```

Here the file job.sh contains lines moving to the singularity work/ directory, which is linked to ${PWD}, and running the coffea main.

File job.sh:

```
#!/usr/bin/sh
cd /work
python my_coffea_main.py
```

What is required for running the analysis in grid? First of all it's not practical to copy the analysed data into the grid CE. It's much better to access the data from a SE via xrootd. Accessing data requires a

Columnar processing

valid grid proxy, which means the x509 proxy file needs to transfered to the CE.

File grid.xrsl:

```
& (executable=grid.job)
(jobname = "myjob")
(stdout=analysis.out)
(stderr=analysis.err)
(inputfiles= ("grid.job" "") ("job.sh" "")
        ("analysis_coffea.py" "")
        ("x509up_u1024147" "")
        ("coffea.sif" ""))
(executables= grid.job job.sh coffea.sif)
(outputfiles=("output.root" ""))
```

RDataFrame

ROOT's RDataFrame offers a modern, high-level interface for analysis of data stored in TTree , CSV and other data formats, in C++ or Python. In addition, multi-threading and other low-level optimisations allow users to exploit all the resources available on their machines.
RDataFrame program follows a workflow:

1. Construct the dataframe object
2. Selection and custom column creation
3. Actions producing the results

Actions are used to aggregate data into results. Most actions are lazy, i.e. they are not executed on the spot, but registered with RDataFrame and executed only when a result is accessed for the first time. The most typical result produced by ROOT analyses is a histogram, but RDataFrame

Columnar processing

supports any kind of data aggregation operation, including writing out new ROOT files.

Example: Running RDataFrame

```
df = ROOT.RDataFrame("Events",
  "data.root")

df = df.Filter("nMuon == 2", "two muons")
df = df.Filter("Muon_charge[0] +
  Muon_charge[1] == 0", "opposite charge")

df_dimuon = df.Define("Dimuon_mass",
  "InvariantMass(Muon_pt, Muon_eta,
  Muon_phi, Muon_mass)")

histo = df_dimuon.Histo1D(("h_mass",
  ";x-axis;y-axis", 200, 0, 200),

  "Dimuon_mass")
```

RDataFrame reads collections as a special type RVec: for example, a branch containing an array of floating point numbers can be read as a RVecF. C-style arrays (with variable or static size), STL vectors and most other collection types can be read this way.
RVec is a container similar to std::vector (and can be used just like a std::vector) but it also offers an interface to operate on the array elements in a vectorised fashion, similarly to Python's NumPy arrays.
RDataFrame::Define can be used to create a new column, like Dimuon_mass in the example.