Evaluating awkward arrays, uproot, and coffea as a query platform for High Energy Physics data

Nick Smith, Lindsey Gray
ACAT2021
1 December 2021
Physics - The Motivation

• The present challenge:
  - Analyze all LHC Run 2 data: \(O(10\) billion\) events
  - Investigate data quality issues with fast time-to-insight
  - Optimize complex (e.g. deep learning) algorithms
• Multiply by \(O(1000)\) data analysts
• These challenges magnified 20x in HL-LHC

• Solutions must be:
  - Easy to use
  - Scalable
  - Fast
Coffea - The Solution

• Coffea: physics analysis in the scientific python ecosystem
  - `$ pip install coffea`

• NanoEvents: column object interface abstraction
  - Supports NanoAOD, DAOD_PhysLite, Delphes, TreeMaker
  - Physics vector semantics (to replace with vector)

• Processor: abstraction for data delivery and scaling
  - Python multiprocessing (today’s focus)
  - Distributed systems: Dask, WorkQueue, Parsl, PySpark
  - ServiceX integration

• Lookup tools
  - Columnar implementations of experiment-specific corrections
  - Abstraction of the above: correctionlib

• Coffea depends on:
  - Scikit-HEP libraries
  - Scientific python libraries
Baby Ecosystem

Coffea serves as incubator for rapid prototyping of missing pieces in our ecosystem. Good abstractions are factored out.
Scaling Coffea

- User is provided a chunk of events
- User fills set of accumulators
  - Histograms, dictionaries of counts, appendable arrays, …
  - Protocol: anything that implements `__add__` or is a dictionary/set
- Coffea executors take care of the rest
  - Task splitting, result aggregation

```python
import awkward as ak
import hist
from coffea import processor

class MyProcessor(processor.ProcessorABC):
    def __init__(self, flag: bool):
        self._flag = flag

    def process(self, events: ak.Array) -> processor.Accumulatable:
        # do physics...
        return {
            "entries": len(events),
            "met": hist.Hist.new.Reg(50, 0, 100).Double().fill(events.MET.pt),
        }

p = MyProcessor(True)
```

root files
parquet files
...
ADL Benchmarks

• A set of 8 simplified HEP analysis tasks
  - Also useful to compare performance and scaling characteristics
  - Both done for RDataFrame vs. industry query solutions in arxiv.org:2104.12615

• Coffea implementation
  - Using Coffea functionality that predates tasks
  - Formatted with black

• Performance and scaling evaluation
  - Google cloud n2-standard-48
  - 48 core (incl. HT) 192GB
  - Use /dev/shm to isolate CPU performance
    • GCP SSD surprisingly slow? 50 MB/s
  - Coffea FuturesExecutor for multi-core
    • IterativeExecutor for single core

```
<table>
<thead>
<tr>
<th>Query</th>
<th>Lines</th>
<th>Characters</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boilerplate</td>
<td>24</td>
<td>556</td>
</tr>
<tr>
<td>Q1</td>
<td>36</td>
<td>936</td>
</tr>
<tr>
<td>Q2</td>
<td>36</td>
<td>928</td>
</tr>
<tr>
<td>Q3</td>
<td>36</td>
<td>970</td>
</tr>
<tr>
<td>Q4</td>
<td>39</td>
<td>1073</td>
</tr>
<tr>
<td>Q5</td>
<td>47</td>
<td>1366</td>
</tr>
<tr>
<td>Q6</td>
<td>59</td>
<td>1905</td>
</tr>
<tr>
<td>Q7</td>
<td>52</td>
<td>1507</td>
</tr>
<tr>
<td>Q8</td>
<td>71</td>
<td>2462</td>
</tr>
</tbody>
</table>
```
import awkward as ak
import hist
import matplotlib.pyplot as plt
from coffea import nanoevents, processor

processor.NanoAODSchema.warn_missing_crossrefs = False

class Q3Processor(processor.ProcessorABC):
    """Plot the $p_T$ of jets with $|\eta| < 1$."""

def process(self, events):
    return (
        hist.Hist.new.Reg(100, 0, 200, name="ptj", label="Jet $p_{T}$ [GeV]"
                          .Double()
                          .fill(ak.flatten(events.Jet[abs(events.Jet.eta) < 1].pt))
    )

def postprocess(self, accumulator):
    return accumulator

if __name__ == "__main__":
    runner = processor.Runner(  
        executor=processor.FuturesExecutor(workers=4),  
        schema=nanoevents.NanoAODSchema,  
        chunksize=2 ** 19,
    )

    output = runner(  
        fileset={"SingleMu": ["Run2012B_SingleMu.root"]},  
        treename="Events",  
        processor_instance=Q3Processor(),
    )
    output.plot()
    plt.gcf().savefig("pt.pdf")
Complete example: query 3

```python
import awkward as ak
import hist
import matplotlib.pyplot as plt
from coffea import nanoevents, processor

processor.NanoAODSchema.warn_missing_crossrefs = False

class Q3Processor(processor.ProcessorABC):
    """Plot the $p_T$ of jets with $|\eta| < 1$.""

    def process(self, events):
        return (hist.Hist.new.Reg(100, 0, 200, name="ptj", label="Jet $p_T$ [GeV]")
                .Double()
                .fill(ak.flatten(events.Jet[abs(events.Jet.eta) < 1].pt))
        )

    def postprocess(self, accumulator):
        return accumulator

if __name__ == "__main__":
    runner = processor.Runner(
        executor=processor.FuturesExecutor(workers=4),
        schema=nanoevents.NanoAODSchema,
        chunksize=2 ** 19,
    )

    output = runner(
        fileset={"SingleMu": ["Run2012B_SingleMu.root"]},
        treename="Events",
        processor_instance=Q3Processor(),
    )
    output.plot()
    plt.gcf().savefig("pt.pdf")
```

class Q8Processor(processor.ProcessorABC):
    """For events with at least three light leptons and a same-flavor
opposite-charge light lepton pair, find such a pair that has the
invariant mass closest to 91.2 GeV in each event and plot the transverse
mass of the system consisting of the missing transverse momentum and
the highest-p_T light lepton not in this pair.
"""

def process(self, events):
    events["Electron", "pdgId"] = -11 * events.Electron.charge
    events["Muon", "pdgId"] = -13 * events.Muon.charge
    events["leptons"] = ak.concatenate([events.Electron, events.Muon], axis=1)
    events = events[ak.num(events.leptons) >= 3]

    pair = ak.argcombinations(events.leptons, 2, fields=["11", "12"])
    pair = pair[(events.leptons[pair.l1].pdgId == -events.leptons[pair.l2].pdgId)]
    with np.errstate(invalid="ignore"):
        pair = pair[ak.singletons(
            ak.argmin(
                ak.abs(
                    (events.leptons[pair.l1] + events.leptons[pair.l2]).mass
                    - 91.2
                ),
                axis=1,
            )
        )]

    events = events[ak.num(pair) > 0]
    pair = pair[ak.num(pair) > 0][:, 0]

    l3 = ak.local_index(events.leptons)
    l3 = l3[(l3 != pair.l1) & (l3 != pair.l2)]
    l3 = l3[ak.argmax(events.leptons[l3].pt, axis=1, keepdims=True)]
    l3 = events.leptons[l3][:, 0]

    mt = np.sqrt(2 * l3.pt * events.MET.pt * (1 - np.cos(events.MET.delta_phi(l3))))
    return hist.Hist.new.Reg(
        100, 0, 200, name="mt", label=r"$\ell$-MET transverse mass [GeV]"
    ).Double().fill(mt)

    def postprocess(self, accumulator):
        return accumulator
Profiling

• After initial performance evaluation, investigated hotspots
  - Improved a few awkward **kernels**
  - Improved NanoEvents **vector math**
    - pow -> mul
    - numba kernels
• Significant improvement in Q6 by user-facing change
  - Eager coordinate transformation
  - Excellent case study for future auto-optimization of delayed compute graph
• Subsequent results use coffea 0.7.9

```python
trijet = ak.combinations(events.Jet, 3, fields=["j1", "j2", "j3"],)
trijet["p4"] = trijet.j1 + trijet.j2 + trijet.j3
```

```python
jets = ak.zip(
    {k: getattr(events.Jet, k) for k in ["x", "y", "z", "t", "btag"],}
    with_name="LorentzVector",
    behavior=events.Jet.behavior,
)
trijet = ak.combinations(jets, 3, fields=["j1", "j2", "j3"],)
trijet["p4"] = trijet.j1 + trijet.j2 + trijet.j3
```
Task splitting

- Coffea parallelizes in “chunks” of event records
  - Lower bound due to overhead (more chunks = more time in python, file parsing)
  - Upper bound due to memory limits (contiguous arrays)
  - In this test, ~500k events is best
- User-facing parameter for now
  - WorkQueue executor already implements dynamic chunk sizing (memory/time)
**Result aggregation**

- Coffea executors merge output as it is produced
  - Sequential for all executors except Dask (which uses tree reduce)
- Run extensions of Q2 with successively more output data
  - 1-3D 100-bin jet kinematics histogram
- For FuturesExecutor, monitor client process CPU load
  - Workers are separate python subprocesses
  - Client deserializes and merges worker output
  - For various settings of chunk size, worker count
  - Client load not limiting throughput in this test
Query performance, repeatability

- Repeat each query three times
- Repeat queries after a few days in new GCP instance
  - Check sensitivity to multi-tenant, etc.
- Results appear consistent
- Mild degradation of scaling after 12 workers
Query performance comparison with RDataFrame

- Performance is similar to that of RDataFrame
  - Compiled C++ RDF, reviewed (+improved Q6) by ROOT experts
  - N.B. Coffea uses child processes, RDF uses threads
Conclusion

• Coffea processors are easy to use
• Coffea FuturesExecutor shows good scaling characteristics in single-machine environments
  - Distributed executors rely on the scaling characteristics of the underlying library
• Columnar analysis with the scientific python stack has competitive performance to traditional solutions