Scalable parallel analysis in Python

Marc Paterno, Saba Sehrish and Jim Kowalkowski

27 October 2020

Why are we presenting?

- In our SciDAC project HEP analytics on HPC we are developing an easy-to-use environment for fast and scalable analysis.
  - easy-to-use: Python, and Python data science tools (e.g. `numpy`, `pandas`);
  - fast: natively data-parallel and taking advantage of HPC features;
  - scalable: same code works on laptops, clusters, HPC systems.
- Our plan is for a tool that is *not* specific to any experiment, nor to just one HEP community (collider or neutrino).
- Our working name is *PandAna*.
- We are interested in working with any group who wants to give this a try.

Primary elements

- Programming language: Python
- Data model: “tidy data” (Wickham 2014), implemented using `pandas` dataframes
- Storage format: HDF5 files, storing “tables”
- Processing model: data parallel, using MPI and `pandas`

Tidy data

- A new name for a standard method for organizing multidimensional data: *data matrix*.
- Each *variable* is represented as a column; each *observation* is represented as a row.
- Related to *Boyce-Codd 3rd normal form*.
- We use a *data matrix* (table) for each set of related observations; analysis code sees `pandas.DataFrame` objects.

events table: one entry per event

evt	njets	nmuons
1	4	3
2	7	4
3	5	0

muons table: one entry per muon

evt	idx	px	py	eta
1	1	10.58	183.61	-3.34
1	2	19.39	53.97	1.19
1	3	257.26	60.12	-0.80
2	1	69.14	16.60	-1.82
2	2	-189.76	-83.38	-0.37
2	3	-103.03	268.04	-1.74
2	4	-66.85	74.68	-1.24

Storage in HDF5

- HDF5 is a file format designed to store large amounts of data.
  - It is supported by the HDF Group [https://www.hdfgroup.org]
  - It is widely available at HPC centers, and easily installable on laptops.
  - It supports (MPI) parallel IO, and has special drivers tuned for parallel filesystems.
    - bypasses the POSIX filesystems bottlenecks
- It has two very important abstractions:
  - *datasets*, which are multidimensional arrays (like `numpy`) of homogeneous types, and
  - *groups*, which are containers of datasets and other groups.
- We use it to store various *tables* (for columnar analysis):
  - a table corresponds to a *group*;
  - a column corresponds to a *dataset* in a *group*;
  - all datasets is a group have the same number of entries, but they can have different types.
- We have successfully used a 42 TB experiment data set in a single file in demos on Cori.

MPI

- MPI is the *Message Passing Interface*.
- It is the dominant parallel programming model in HPC; it also works on laptops and in clusters.
- The programming model uses multiple *operating system processes* communicating using *messages*:
  - Does not use threads, so no thread safety issues.
  - Every HPC center has an MPI optimized for the networking in that computer.
- `mpi4py` is a Python module that implements MPI.
- In our use, analysis code (almost) never makes calls to the MPI library (and mostly doesn't need to know that it exists).
- MPI provides a simple way into data-parallel multi-node processing.

Data parallel

- Analysis code is written (almost) exactly as in a serial program; parallelism is *implicit*.
- If you start your program using `mpirun`, each process sees disjoint (and so independent) data (*data parallelism*).
- PandAna supports parallel reading of HDF5 files which have our type of schema.
- Each MPI process processes a portion of each table that is used.
- Many analyses do not use *every* table in a large dataset.
  - PandAna will read from only the tables used in your analysis program.
- Many analyses do not use *every* column in each table that is used.
  - PandAna will read only those columns that are actually needed.
- Analysis code sees a `pandas.DataFrame` for each table, carrying only columns that will be used.
  - *All* needed data is in some dataframe.
  - *No* data are duplicated between processes.
  - Data are distributed to assure that no event is split across processes.

The two important PandAna analysis concepts: *Vars* and *Cuts*

- A *Var* represents (an array of) data (really a `pandas.Series`).
  - It can be *read from a column*, or
  - It can be calculated from one or more columns, or
  - It can be the result of aggregating a calculation of one or more columns of some grouping.
- A *Cut* represents selection criteria, yielding an (array of) `bool` value(s) for each *item* (e.g. event, muon, . . .)
  - It can be a simple comparison of a threshold to data.
  - It can be a comparison against a calculation from some data.
  - It can be a logical composition of other *Cuts*.
- Common in analysis is to define a *Var* to be calculated for each event that passes some *Cut*, and to create a summary (e.g. a histogram) of that values of that *Var*.

Example main program

```
import ... bunch of modules, omitted for space ...
def main(input_files, max_files):
    tables = Loader(input_files, limit=max_files)
    px = Var(lambda tables: tables['muons']['px'])
    my_spectrum = Spectrum(tables, kDiMuon, px)
    tables.Go()

    n, edges = my_spectrum.histogram(bins=50, range=(0, 1000))
    total_n = MPI.COMM_WORLD.reduce(n, op=MPI.SUM, root = 0)
    if MPI.COMM_WORLD.rank == 0:
        print('Selected ', total_n.sum(), ' muons')
```

Example Vars

```
kMuonPx = Var(lambda tables: tables['muons']['px'])

def kMuonPt(tables):
  df = tables['muons']
  return np.sqrt(df['px']**2 + df['py']**2)
kMuonPt = Var(kMuonPt)

def kMuonMaxPt(tables):
  return kMuonPt(tables).groupby('evt').agg(np.max)
kMuonMaxPt = Var(kMuonMaxPt)
```

Example Cuts

```
kDiMuon = Cut(lambda tables: tables['evt']['nmuons'] >= 2)
kMultiJet = Cut(lambda tables: tables['evt']['njets'] > 4)
kInteresting = kDiMuon & kMultiJet
```

Some performance results: Processing the entire LArIAT raw data sample

- ~42 TB of digitized waveforms; after compression, file size was 4.2 TB (single file)
- 15,684,689 events.
- 480 wires per event, 3072 samples per wire.
- Running on Cori Phase II
- Task:
  1. read and decompress waveform data
  2. apply FFT
  3. mask out noise signal in transform space
  4. apply inverse FFT

Process per node	Nodes	Time (s)	Normalized time
64	200	1068.0	1.000
64	1200	180.5	0.169

- Step 1 (read and decompress data) took ~23 seconds on 1200 nodes.

Usage experience

- We have multiple experiments writing HDF5 analysis ntuples directly from the framework in production
  - We have a C++ library `hep_hpc` on BitBucket to help write HDF5 files in our style.
  - This library is used by DUNE, NOvA, and Muon g-2
- NOvA collaborators have been using PandAna for several years
  - Users report faster development cycle compared to C++ (no compilation, fast exploration)
  - Users report 5-100 times *faster* than compiled C++/ROOT code, for various analyses
- We have an example program that uses `uproot` to read a CMS *nanoaod* "flat ntuple" and write our style of HDF5 file.
  - The generic equivalent is a fairly obvious modification.
- We are working with ATLAS on a PandAna analysis using *nanoaod* style data.

Summary: the most important features

- Analysis code written at high level
  - implicit data parallelism
  - operations on arrays, rather than loops
  - “tidy data” manipulations are quick to write and easy to read
- Scalability provided by data-parallel architecture
- By using popular packages, we get free improvement in packages
  - pandas is a dominant package in Python data science; uses numpy for performance
  - HDF5 is the dominant storage format in HPC; many ML packages prefer HDF5 storage
- We are interested in working with any group who wants to give this a try.

References

Wickham, Hadley. 2014. “Tidy Data.” *Journal of Statistical Software, Articles* 59 (10): 1–23. https://doi.org/10.18637/jss.v059.i10.