Data-loading (for ML applications) using TDFs

Stefan Wunsch

stefan.wunsch@cern.ch

2018-02-22

Motivation

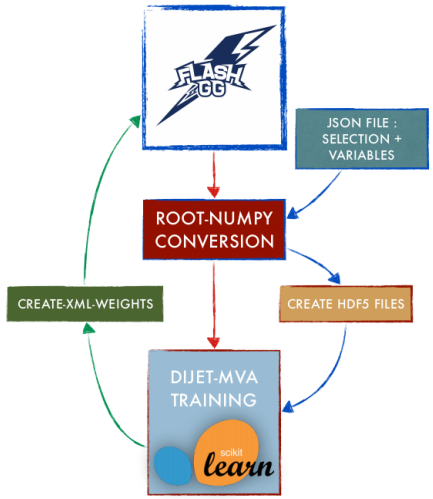
- Most of the data analysis of the high-level HEP analyses happens in the Python domain (frameworks of analysis groups on top of flat ntuples).
- Even more extrem for ML applications: Most frameworks are only usable from Python (Keras, xgboost, most of TensorFlow, PyTorch, ...)
- How data-loading often looks like (for ML applications) in HEP:

```
...
>>> x = root_pandas.read_root("file.root", "tree").as_matrix()
>>> print(x.shape)
(number_of_entries, number_of_branches)
>>> model.fit(x, ...)
...
```

- Most efficient solution today: `root_numpy` (used by `root_pandas`)
- But ROOT has the possibilities to do this more efficient.

Random slide from a MVA-based analysis

- Most of the tools that we will be using for the VBF analysis are based on python packages for data analysis:
  - For data handeling we will use numpy[0] and pandas[1]
  - to use scikit-learn[2], you will need to translate the root-tree into numpy arrays
  - A tool exist already to solve this issue root_numpy[3]



Feature request

- Support taking data from ROOT files and put it into memory (as fast as possible)
- Memory layout of the output: Contiguous, interpretable as n-dimensional arrays
- Make the data accessible from Python, interpretation of memory as numpy array

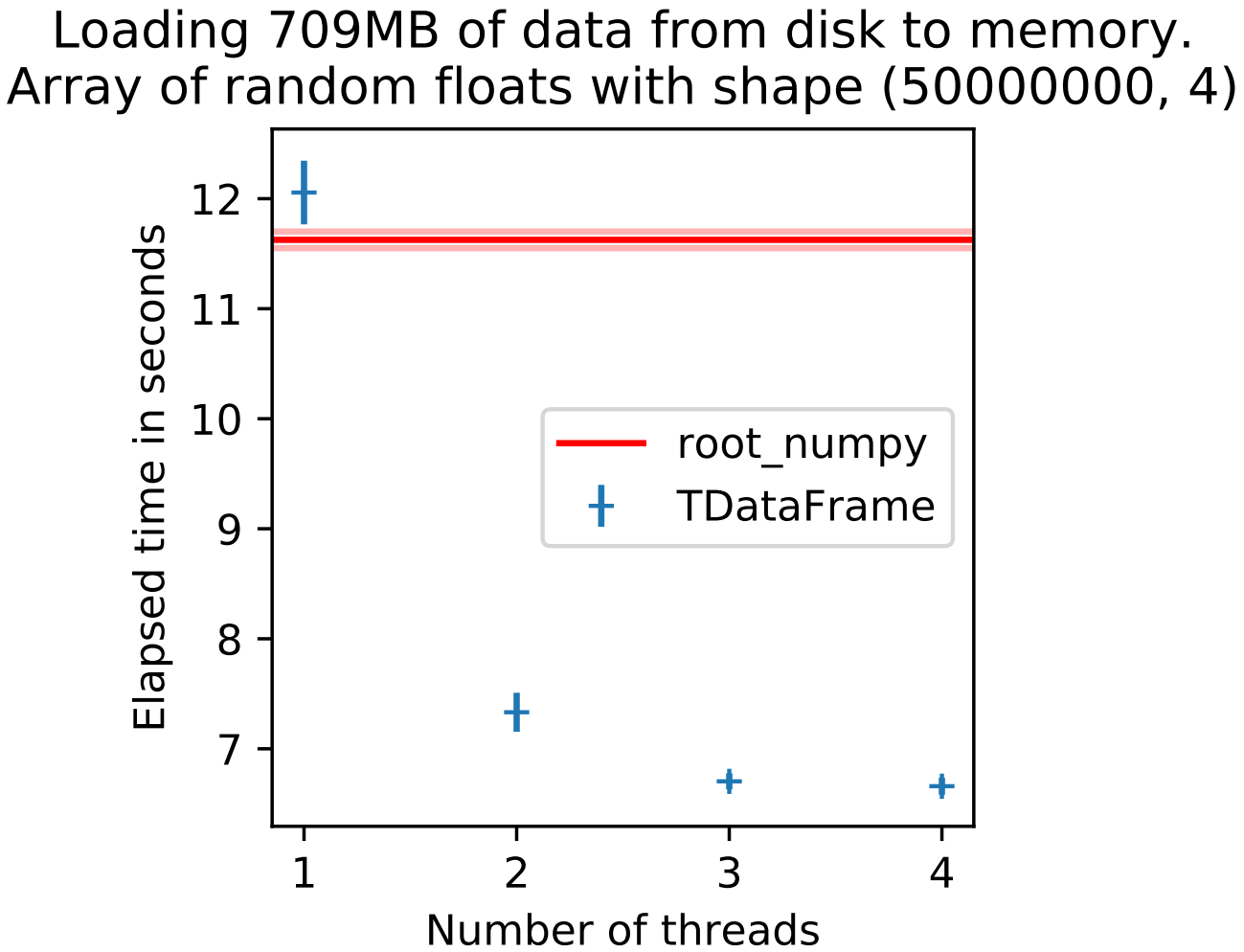
Interface proposal using `TDataFrame`:

```
>>> tdf = ROOT.Experimental.TDataFrame("tree", "file.root")
>>> tdf = tdf.Filter("var1>0").Define("new_var", "var1*var2")
>>> x = tdf.AsMatrix(["var1", "var2", "new_var"])
>>> print(x.shape)
(number_of_entries, 3)
```

Advantages compared to `root_numpy` approach

- Useful set of TDF features directly usable
  - Efficient selection of data (`Filter`)
  - Define new variables (`Define`)
  - Other fancy operations (`ForEach`)
  - ...
- Size of input files not limited by memory
- Make use of implicit multi-threading
  $\rightarrow$ Gain of a factor of N in speedup (ideally)

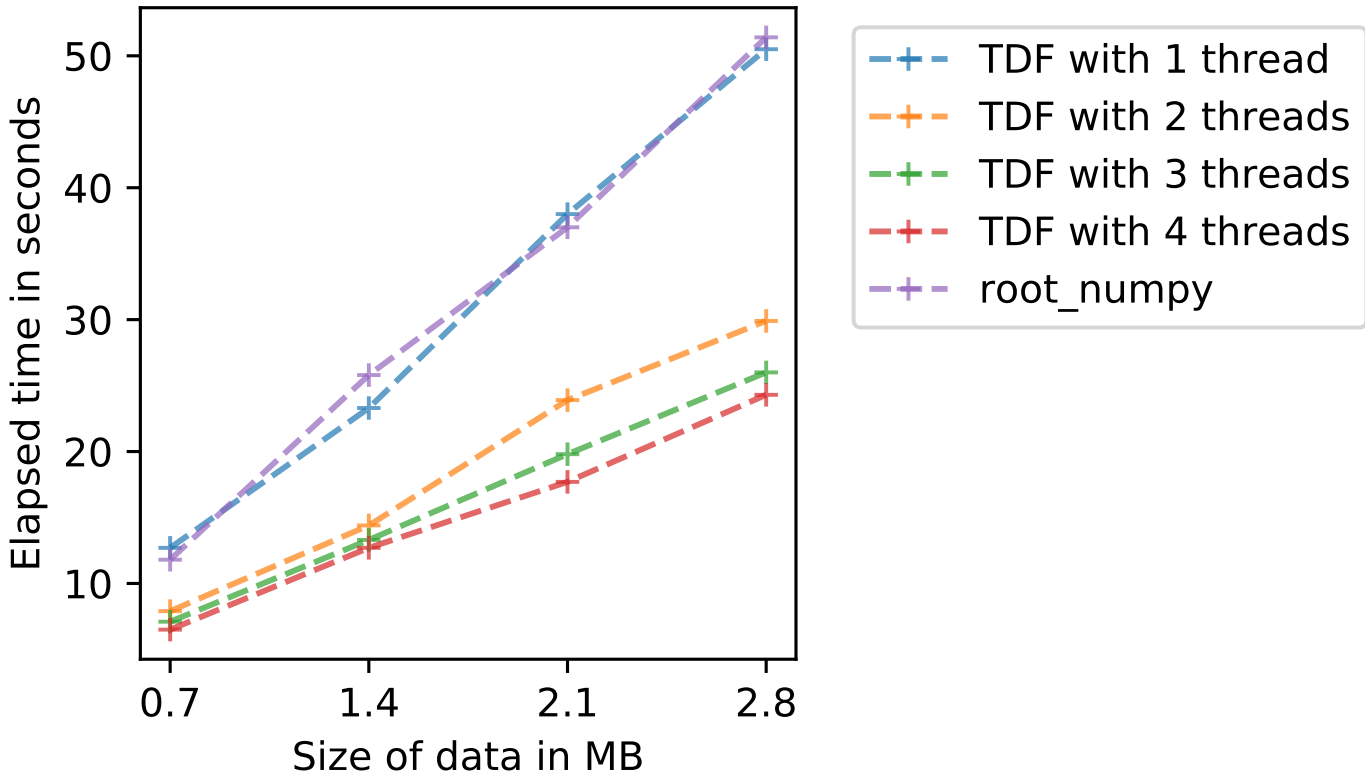
First benchmarks (1)



Measured on a machine with (2) 4 (physical) logical cores.

First benchmarks (2)

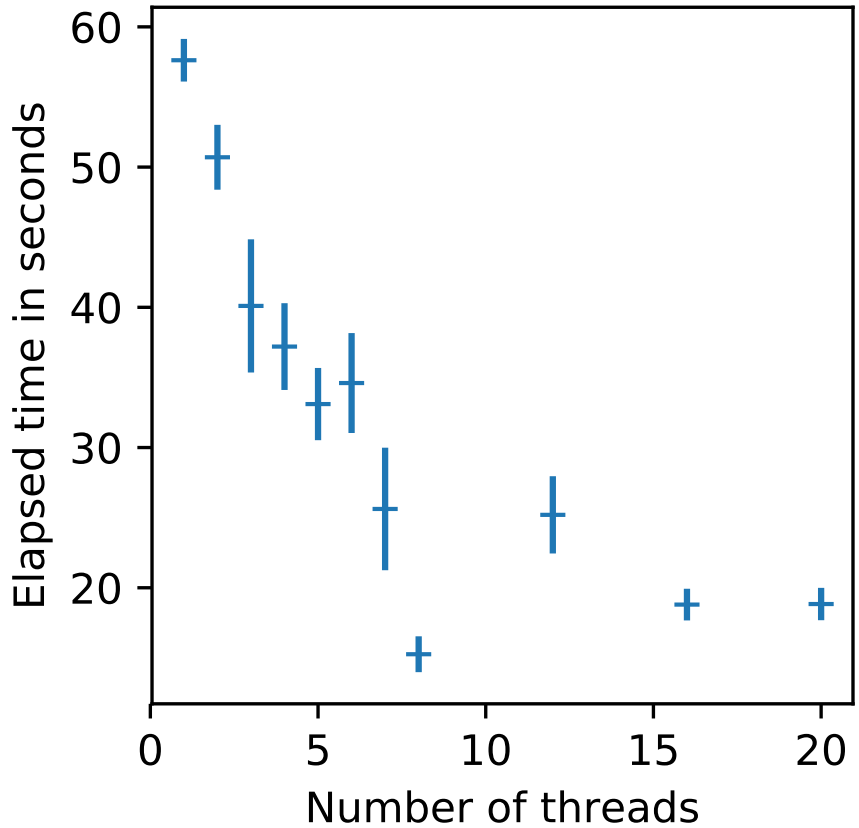
Performance subject to input data size and number of threads



Measured on a machine with (2) 4 (physical) logical cores.

First benchmarks (3)

Loading 2.8GB of data from disk to memory.



Measured on a machine with (24) 48 (physical) logical cores.

What is missing to do this properly?

- Proposal for a matching interface in C++ (Container for returned data?)

- Proper PyROOT handling of numpy arrays
  - Input argument handling: Interpreted as `float*`, shape information is lost
  - Return value handling: Not supported (?)