TTreeProcessor: A toy framework for parallel ntuple processing

Brian Bockelman
DIANA forum, 14 November 2016
In the beginning

```
TH1F *myHist = new TH1F("h1", "ntuple", 100, -4, 4);
TFile *tf = TFile::Open("myfile.root");
TTreeReader myReader("T", tf);
TTreeReaderValue<Float_t> myPx(myReader, "px");
TTreeReaderValue<Float_t> myPy(myReader, "py");
while (myReader.Next()) {
    myHist->Fill(*myPx + *myPy);
}
```

TH1F *myHist = new TH1F("h1", "ntuple", 100, -4, 4);
TFile *tf = TFile::Open("myfile.root");
TTreeReader myReader("T", tf);
TTreeReaderValue<Float_t> myPx(myReader, "px");
TTreeReaderValue<Float_t> myPy(myReader, "py");
while (myReader.Next()) {
    myHist->Fill(*myPx + *myPy);
}

“Given a new histogram, fill it with the contents of (px+py) from the tree T in the file myfile.root.

The rest is mostly boilerplate!
Boilerplate Hurts!

- Boilerplate hurts!
  - Cognitively, it distracts from what the user is trying to accomplish.
  - Provides opportunity for bugs.
  - Forces use of a particular API (4 years ago, the example would have used “SetBranchAddress” and friends).
  - Forces the user to hardcode semantics that may not be necessary.

- Hardcoded semantics in this example:
  - Single thread.
  - Loop iterations are dependent.
  - TTreeReader-based reading.

- Other than a for-loop, what other paradigms could be used to process ntuples?
Stream Processing

• Stream processing is a programming paradigm where, given a sequence of data (a stream), a series of operations (kernel functions) is applied to each element in the stream.

• Idea:
  • User should specify a series of a few simple kernels.
  • The processing framework should take care of creating streams and executing the kernels. The framework finds parallelism (fork/join streams) as necessary.
  • Framework provides a few common helper kernels to ease use.
  • Encourages functional-like programming, but is not functional (kernels may have side-effects).

Background reading: https://en.wikipedia.org/wiki/Stream_processing
http://www.oracle.com/technetwork/articles/java/ma14-java-se-8-streams-2177646.html
Stream Processing for ROOT

• I would like to introduce the stream processing paradigm to the ROOT ecosystem.

• I believe it could be made non-invasive: users could quickly pick up the concepts but not have to learn Haskell-with-C++-syntax.

  • Stylistically, aligns with the “Big Data” ecosystem but still keeps with the familiar (ROOT).

• Currently project: the TTreeProcessor: https://github.com/bbockelm/ttreeprocessor

  • The TTreeProcessor library is a header-only package, dependent on ROOT, TBB, and Vc.
Welcome to C++ Meta-Programming Hell

- TTreeProcessor heavily utilizes C++ meta-programming in order to generate the majority of the code at compile-time.
  
  - TTreeProcessor itself is a template whose arguments are the branch types and a list of kernels.
  
  - Code should be read with a beer in one hand and coffee in the other.

- **Goal**: All kernels are inlined and compiler merges them effectively into a single common block.
  
  - **No polymorphism**. No type erasure.
  
  - Intermediate `std::tuple` objects are eliminated.
  
  - Even with the C++ template scaffolding, try to have equivalent performance as a plain-old C loop.

- **Mostly achieved!** Will never be equivalent to a dedicated stream processing language, but .
Mappers and Filters

template<typename Tuple, typename... InputArgs>
class TTreeProcessorMapper : public TTreeMapper {
    public:
        TTreeProcessorMapper() {}
        TTreeProcessorMapper(const TTreeProcessorMapper&) = delete;
        TTreeProcessorMapper(TTreeProcessorMapper&&) = default;

        Tuple map(InputArgs...) const noexcept {};

        bool finalize() {return true;}  
        typedef T output_type;
};

template<typename... InputArgs>
class TTreeProcessorFilter : public TTreeFilter {
    public:
        TTreeProcessorFilter() {}
        TTreeProcessorFilter(const TTreeProcessorFilter&) = delete;
        TTreeProcessorFilter(TTreeProcessorFilter&&) = default;

        bool filter(InputArgs...) const noexcept {};

        bool finalize() {return true;}
};

• Kernels must inherit from either a **Mapper** or a **Filter** class.
  • Must be declared **final** to avoid virtual functions.
  • A map takes the input from the previous step (**InputArgs**... parameter pack) and return the input for the subsequent kernel as a **std::tuple<>**.
  • **filter** and **map** are **const**: they must be thread-safe.
  • A filter will return a boolean; if **false**, the streams discards the event.
  • **finalize** is invoked after all streams are finished. Guaranteed to be invoked in a single-threaded context.
Pre-packaged kernels

• Users are not expected to write their own kernels in the most case.

• TTreeProcessor uses metaprogramming to generate built-in kernels for common use cases:
  
  • `.map(fn)` method generates a new Mapper kernel given a lambda function, returning a new `TTreeProcessor` object with the additional kernel added to the template.
  
  • `.filter(fn)` method does same but with a new Filter kernel.
Silly Example

```cpp
#include "TTreeProcessor.h"

int main(int argc, char *argv[]) {
    TFile *tf = TFile::Open("myfile.root");
    ROOT::TTreeProcessor<float, int, double> processor({"a", "b", "c"});
    processor
        .filter([](float a, int b, double c) {return a <= 5;})
        .map([](float a, int b, double c) -> std::tuple<float, int> {return {a*a+1, a+b};})
        .process("T", {tf});
    return 0;
}
```

- In plain English:
  - **Process** branches a, b, and c of type float, int, and double, respectively, as found in Tree T and file myfile.root.
  - **Filter** on events where the value of a is <= 5.
  - **Map** a and b to a*a + 1 and a+b, respectively.
  - Not particularly useful without side-effects!
Parallel Streams

- **Idea:** `map` and `filter` are thread safe and each event is data-independent. Let’s process in parallel!

- Utilize TBB (already present in ROOT for IMT) to break the streams into independent tasks.

  - What’s the right “granularity” of event processing? Task per event = too fine-grained. Task per file = too coarse-grained.

  - Settled on a *task per event cluster*: typically results in one task per every 20MB of data.

- Currently, must be enabled explicitly by using `parallelProcess` method.

  [https://www.threadingbuildingblocks.org](https://www.threadingbuildingblocks.org)
Vectorization

• If the kernels accept Vc-based vector types, then the processor will read out multiple events at once and invoke the kernel chain with the vector equivalent of the arguments.

  • Kernels are invoked with a mask argument; this is a bitmask indicating which events are currently valid.

  • Filters no longer return a bool but rather an updated mask.

  • Note: actual implementation still in-progress.

• Everything “looks” the same except for different types.

• Example use of the interface:

```cpp
ROOT::TTreeProcessor<float, int, double> processor({"a", "b", "c"});
processor
  .map([](maskv m, floatv a, intv b, doublev c)
      -> std::tuple<int, float>
      {return {y, x};})
  .process("T", {tf1, tf2, tf3});
```
A Toy Framework

• The TTreeProcessor is in its infancy:
  • Can generate new maps and filters on the chain via lambdas.
  • Can write your own classes.
  • Mostly been tested on unrealistically-trivial data formats.
  • At least one example of how the finalize method works.
  • Parallel and serial processing works; vectorization should be done by Friday.
• Many places to contribute!
Thoughts on the future

• Many miles left to go to explore this idea:

  • Tutorials, blogs, documentation to write. This presentation is the first time the processor has “seen light of day”. Will move to the DIANA/HEP project group soon.

  • Would like the library to be integrated in ROOT itself.

  • Probably needs 5-10 kernels for common operations like histogramming and file I/O. Would like to implement the majority of the DIANA-developed histogrammar language.

  • Did we eliminate the boilerplate? Or trade it off for esoteric C++ features? What can be done?

  • Python is honestly the better language for prototyping. Numba has demonstrated that a subset python can JIT’d using LLVM, even when integrating inside a larger framework.

    • **Could we write TTreeProcessor kernels in Python** and still JIT the entire infrastructure?

    • How powerful is cling’s JIT? Could it do type deduction from the ROOT branches and instantiate the correct TTreeProcessor template?

• Fundamentally, interested in faster/better ntuple processing because I want an improved IO stack. With the TTreeProcessor, I hope we can increase processing rates in order to advantage of bulk IO APIs.