Adaptable framework LDRD update

https://indico.fnal.gov/event/52666/contributions/231769/attachments/153101/198580/SCDProjects_LDRD_Knoepfel.pdf

Kyle J. Knoepfel

DUNE/LDRD monthly meeting

14 September 2022

Since last time (July 13)

- **SIST summer internship is over**
  - Tyler Terwilliger explored a graph-based framework approach (using OneTBB's flow graph).
  - No earth-shattering contributions toward furthering the LDRD, but it helped us learn under what conditions a flow-graph approach could be successful.

- **Began analysis of current DUNE workflows**
  - I profiled memory usage and CPU efficiency for various DUNE *art* jobs to get a baseline.
  - Some memory issues could be addressed by improving LArSoft algorithms (used widely by DUNE).
  - I've implemented *some* improvements to LArSoft (pull requests are forthcoming). But fully addressing this problem is off-scope and will require dedicated effort.

Observations when looking at framework code

- Most framework processing can be categorized according to (combinations of) the following patterns:

  Transforming – creating data products from existing data products of the same processing level (e.g. produce functions)

  Reducing – creating data products based on accumulations of data at a more granular processing level (e.g. endSubRun)

  Filtering – processing a subset of data based on satisfying Boolean criteria.

- For each of these patterns, the user is not responsible for sending data to the corresponding algorithm--i.e. the user writes the code that ***reacts*** to the data.
- ***How*** that algorithm is called depends on the context.

art's design

Physics algorithms should be implemented in external libraries

Putting the physics algorithms (e.g., `make_interesting_product` in the example above) into a separate library provides two important benefits:

1. It allows the algorithm to be shared between several modules
2. It makes it possible to test the algorithm outside of the **art** framework

Testing of a stand-alone algorithm implementation outside of the **art** framework is simpler than testing a module that implements that algorithm because it does not require the production of modules to create the correct input for the module to be tested, nor the production of modules to test the output of the module to be tested.

https://cdcvs.fnal.gov/redmine/projects/art/wiki/Art_Module_Design_Guide#Physics-algorithms-should-be-implemented-in-external-libraries

- Almost nobody does this—and this is not because of lack of discipline.

  Even LArSoft's art "independent" code still relies on art's data model.

- With art (and other frameworks), there is enough boilerplate that it doesn't make sense for a physicist to write framework-agnostic code in the off-chance that they'll use the algorithm outside of a framework.

Can the design be improved?

- I believe so:
  - The framework's data model can be isolated (most of the time it's not used)
  - C++ advancements can help

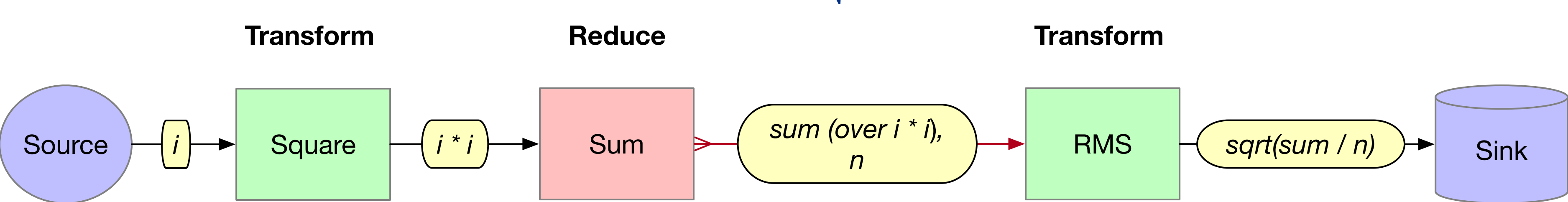
- But still should support legacy code where possible.

Example: RMS calculation

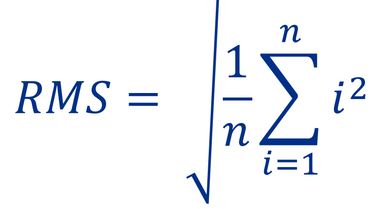
$$RMS = \sqrt{\frac{1}{n}\sum_{i=1}^{n} i^2}$$

Example: RMS calculation

$$RMS = \sqrt{\frac{1}{n}\sum_{i=1}^{n} i^2}$$

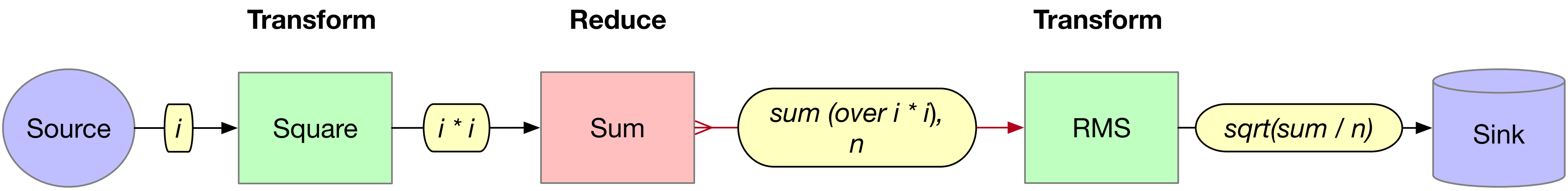


Example: RMS calculation



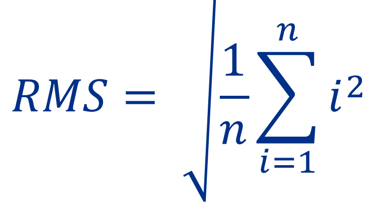
$$RMS = \sqrt{\frac{1}{n}\sum_{i=1}^{n} i^2}$$

The *art* way




```
class Squarer {
  void produce(Event& e) {
    auto i = e.get("num");
    e.put("squared_num", i *
i);
  }
};
```

Example: RMS calculation



$$RMS = \sqrt{\frac{1}{n}\sum_{i=1}^{n} i^2}$$

The *art* way

Transform — Reduce — Transform

Source → *i* → Square → *i * i* → Sum → *sum (over i * i), n* → RMS → *sqrt(sum / n)* → Sink

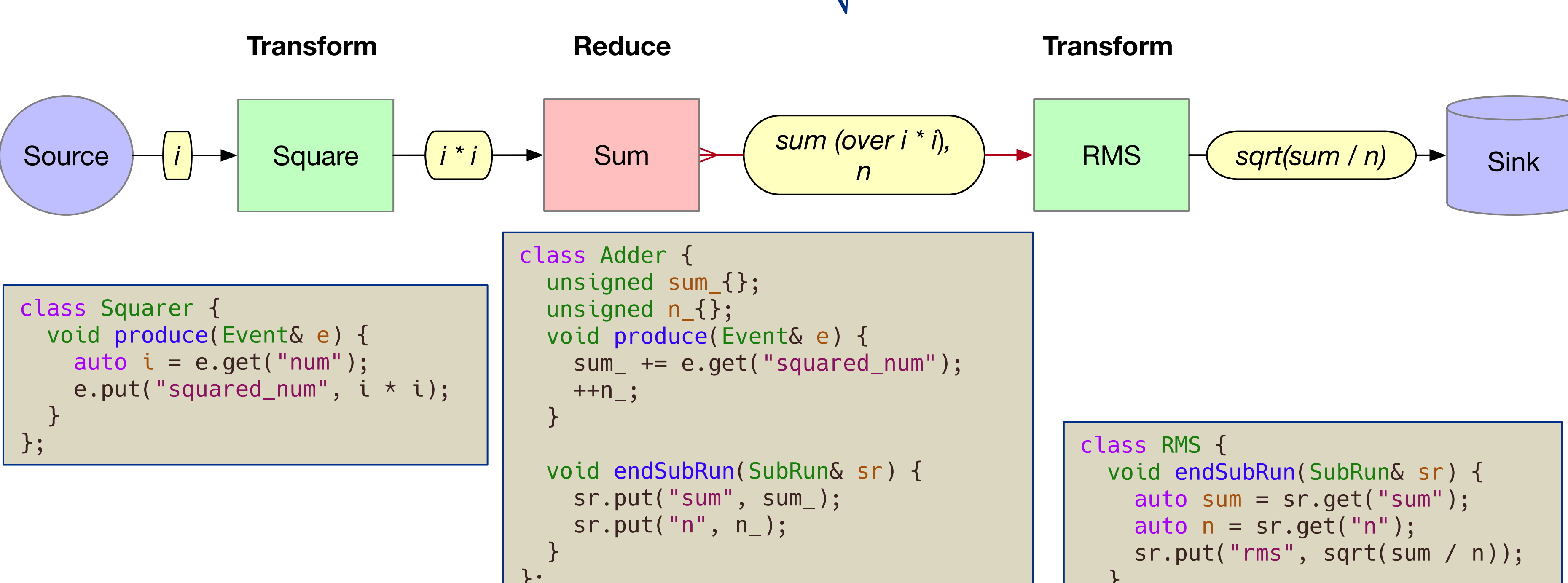
```
class Squarer {
  void produce(Event& e) {
    auto i = e.get("num");
    e.put("squared_num", i * i);
  }
};
```

```
class Adder {
  unsigned sum_{};
  unsigned n_{};
  void produce(Event& e) {
    sum_ += e.get("squared_num");
    ++n_;
  }

  void endSubRun(SubRun& sr) {
    sr.put("sum", sum_);
    sr.put("n", n_);
  }
};
```

Example: RMS calculation

$$RMS = \sqrt{\frac{1}{n}\sum_{i=1}^{n} i^2}$$

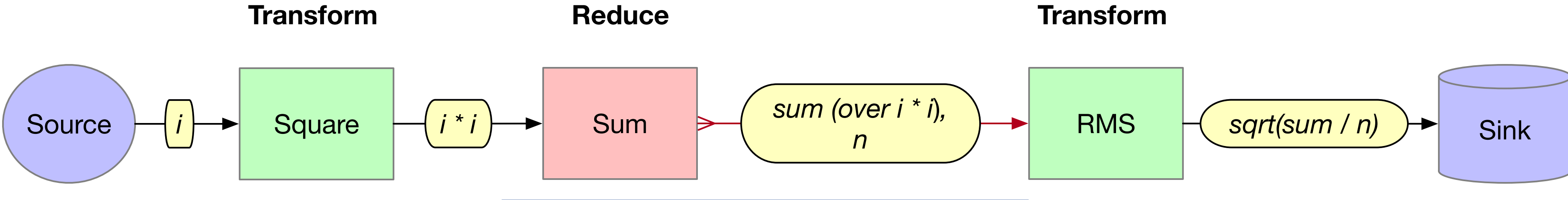
The *art* way



Example: RMS calculation

$$RMS = \sqrt{\frac{1}{n}\sum_{i=1}^{n} i^2}$$

The *art* way




```
class Squarer {
  void produce(Event& e) {
    auto i = e.get("num");
    e.put("squared_num", i * i);
  }
};
```

```
class Adder {
  unsigned sum_{};
  unsigned n_{};
  void produce(Event& e) {
    sum_ += e.get("squared_num");
    ++n_;
  }

  void endSubRun(SubRun& sr) {
    sr.put("sum", sum_);
    sr.put("n", n_);
  }
};
```

Processing levels are explicit

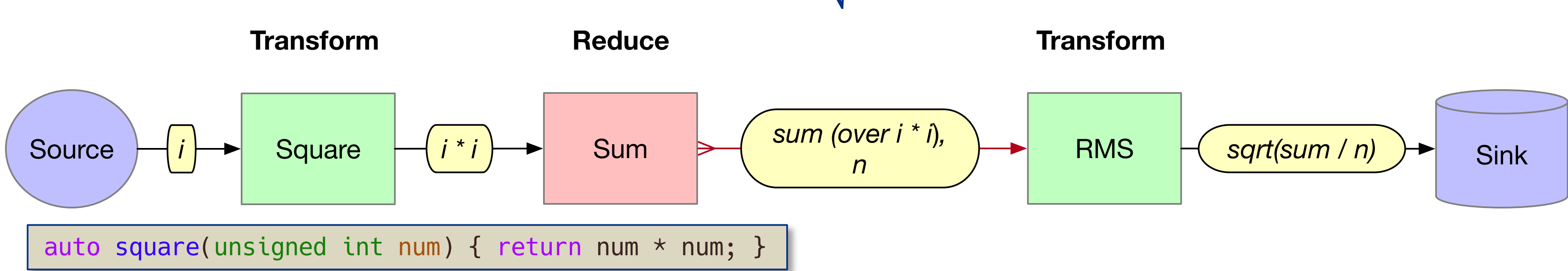
```
class RMS {
  void endSubRun(SubRun& sr) {
    auto sum = sr.get("sum");
    auto n = sr.get("n");
    sr.put("rms", sqrt(sum / n));
  }
};
```

Example: RMS calculation

$$RMS = \sqrt{\frac{1}{n}\sum_{i=1}^{n} i^2}$$

The *"better"* way?

No explicit processing levels required




```
auto square(unsigned int num) { return num * num; }
```

Example: RMS calculation

$$RMS = \sqrt{\frac{1}{n}\sum_{i=1}^{n} i^2}$$

The *"better"* way?

No explicit processing levels required

Transform — Reduce — Transform

Source → *i* → Square → *i * i* → Sum → *sum (over i * i), n* → RMS → *sqrt(sum / n)* → Sink

```
auto square(unsigned int num) { return num * num; }
```

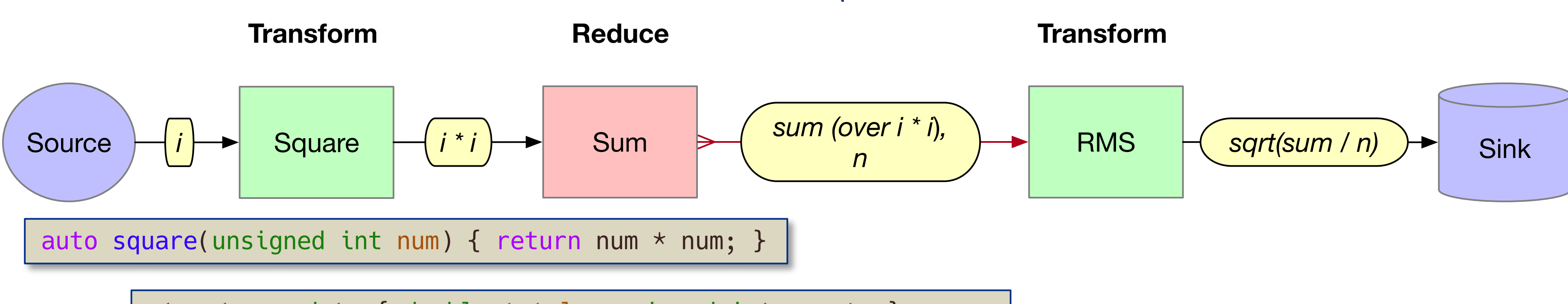
```
struct sum_data { double total; unsigned int count; };

auto sum_reduction_op(sum_data left, unsigned int squared)
{
  return sum_data{.total=left.total + squared,
                  .count=left.count + 1};
}
```

Example: RMS calculation

$$RMS = \sqrt{\frac{1}{n}\sum_{i=1}^{n} i^2}$$

The *"better"* way?

No explicit processing levels required




```
auto square(unsigned int num) { return num * num; }
```

```
struct sum_data { double total; unsigned int count; };

auto sum_reduction_op(sum_data left, unsigned int squared)
{
  return sum_data{.total=left.total + squared,
                  .count=left.count + 1};
}
```

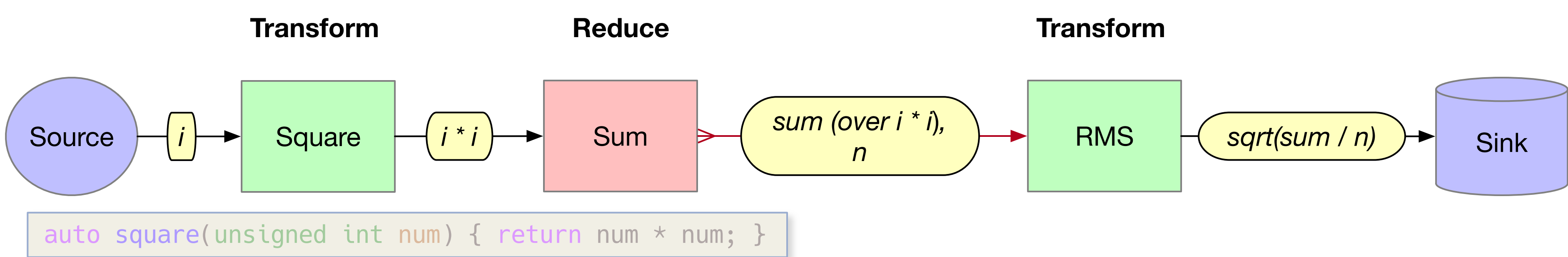
```
double rms(sum_data data) { return std::sqrt(data.total / data.count); }
```

Example: RMS calculation

$$RMS = \sqrt{\frac{1}{n}\sum_{i=1}^{n} i^2}$$

The *"better"* way?

No explicit processing levels required




```
auto square(unsigned int num) { return num * num; }
```

```
struct sum_data { double total; unsigned int count; };

auto sum_reduction_op(sum_data left, unsigned int squared)
{
  retur
}
```

```
// Framework "glue" code
declare_transform(square).input("num").output("squared_num");
declare_reduction(sum_reduction_op, sum_data{}).input("squared_num").output("sum_data");
declare_transform(rms).input("sum_data").output("rms");
```

```
double rms(sum_data data) { return std::sqrt(data.total / data.count); }
```

Framework glue code

- **Naturally separates user code from framework assumptions**
  - The input arguments for each registered function are data products produced by upstream functions or provided by the input source.
  - The return value(s) of each registered function will be registered as a data product.
  - Possible to get product information for a registered function—just wrap the function argument with the `handle<>` template
- **Full glue code specification is richer than what you see on the previous page**
  - Allows the registration of member functions for concrete objects (a *component*) that can be initialized by e.g. ParameterSet
  - Can specify concurrency level for each callback
  - Non-product dependencies will need to be specifiable in some cases.
  - Product-lookup policy will be specifiable via the glue code.

Framework glue code (cont.)

- More granular approach to framework processing, yet a pattern exists for supporting legacy code

```
// Can be automatically generated for legacy module
auto c = make_component<MyModule>(pset);
c.declare_transform(&MyModule::beginRun).listen_for<Run>();
c.declare_transform(&MyModule::beginSubRun).listen_for<SubRun>();
c.declare_reduction(&MyModule::produce).listen_for<Event>();
c.declare_reduction(&MyModule::endSubRun).depends_on(&MyModule::produce);
c.declare_reduction(&MyModule::endRun).depends_on(&MyModule::endSubRun);
```

- Problems to be addressed with new paradigm:

  What do services look like?

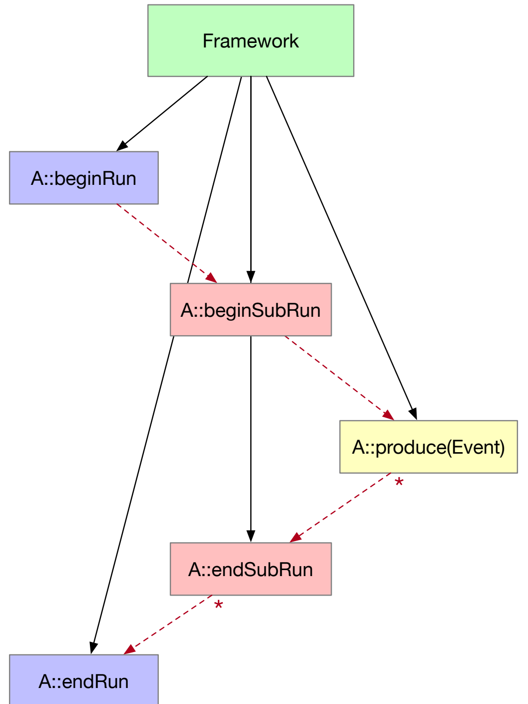
  What about persistable pointers?

  etc.

Backup

Thinking of modules as graphs

The *art* way



The "better" way