DUNE framework LDRD

https://indico.fnal.gov/category/1415/attachments/154813/201467/FY2022_LDRD_Description_Knoepfel.pdf

Kyle J. Knoepfel

LDRD report @ SCD Projects Meeting

15 September 2022

Since last time

- **Began analysis of current DUNE workflows**
  - DUNE's hope is that their eventual framework will make it easier to manage memory well.
  - To that end, 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.
- **Technical progress**
  - I've met with Chris Jones and Marc Paterno to discuss the data model and the programming model.
  - I'm exploring an API where users register functions with the framework similar to how C++ functions are bound to Python in pybind11 (https://pybind11.readthedocs.io/en/stable/basics.html#creating-bindings-for-a-simple-function)
  - This approach provides a clean separation between framework and user code.

Registering a function with the framework

```
// add.hpp
int add(int i, int j) {
    return i + j;
}
```

Registering a function with the framework

```
// add.hpp
int add(int i, int j) {
    return i + j;
}
```

- **Framework must know:**

- What data products correspond to the inputs “i” and “j”
- The name of the output data product
- The processing levels of the data products
- The desired concurrency level

Fermilab

Registering a function with the framework – the art way

```
// add.hpp
int add(int i, int j) {
    return i + j;
}
```

- **Framework must know:**

- What data products correspond to the inputs “i” and “j”
- The name of the output data product
- The processing levels of the data products
- The desired concurrency level

```
#include "art/Framework/Core/SharedProducer.h"
#include "app.hpp"


class Adder : public art::SharedProducer {
public:
  Adder(ParameterSet const&, art::ProcessingFrame const&) {
    produces<int>("sum");
    async<art::Event>();
  }

  void produce(art::Event& e, art::ProcessingFrame const&) override {
    auto const sum = add(e.getProduct<int>("i"),
                         e.getProduct<int>("j"));
    e.put(std::make_unique<int>(sum), "sum");
  }
};

DEFINE_ART_MODULE(Adder);
```


Fermilab

Registering a function with the framework – the art way

```
// add.hpp
int add(int i, int j) {
    return i + j;
}
```

- **Framework must know:**

- What data products correspond to the inputs “i” and “j”
- The name of the output data product
- The processing levels of the data products
- The desired concurrency level

```
#include "art/Framework/Core/SharedProducer.h"
#include "app.hpp"


class Adder : public art::SharedProducer {
public:
  Adder(ParameterSet const&, art::ProcessingFrame const&) {
    produces<int>("sum");
    async<art::Event>();
  }

  void produce(art::Event& e, art::ProcessingFrame const&) override {
    auto const sum = add(e.getProduct<int>("i"),
                         e.getProduct<int>("j"));
    e.put(std::make_unique<int>(sum), "sum");
  }
};

DEFINE_ART_MODULE(Adder);
```


Fermilab

Registering a function with the framework – the art way

```
// add.hpp
int add(int i, int j) {
    return i + j;
}
```

- **Framework must know:**

- What data products correspond to the inputs “i” and “j”
- The name of the output data product
- The processing levels of the data products
- The desired concurrency level

```
#include "art/Framework/Core/SharedProducer.h"
#include "app.hpp"


class Adder : public art::SharedProducer {
public:
  Adder(ParameterSet const&, art::ProcessingFrame const&) {
    produces<int>("sum");
    async<art::Event>();
  }

  void produce(art::Event& e, art::ProcessingFrame const&) override {
    auto const sum = add(e.getProduct<int>("i"),
                         e.getProduct<int>("j"));
    e.put(std::make_unique<int>(sum), "sum");
  }
};

DEFINE_ART_MODULE(Adder);
```

*A lot of **rigid** boilerplate!*

***Consequence:** ~Nobody separates the meaningful code from the framework wrapper.*

Fermilab

Registering a function with the framework – a better way

```
// add.hpp
int add(int i, int j) {
    return i + j;
}
```

Glue code approach

In computer programming, **glue code** is executable code (often source code) that serves solely to "adapt" different parts of code that would otherwise be incompatible.

https://en.wikipedia.org/wiki/Glue_code

Registering a function with the framework – a better way

```
// add.hpp
int add(int i, int j) {
    return i + j;
}
```

Glue code approach

In computer programming, **glue code** is executable code (often source code) that serves solely to "adapt" different parts of code that would otherwise be incompatible.

https://en.wikipedia.org/wiki/Glue_code

```
#include <pybind11/pybind11.h>
#include "add.hpp"

PYBIND11_MODULE(example, m) {
  m.def("add", add, "...", py::arg("i"), py::arg("j"));
}
```

pybind11 syntax as motivation

Registering a function with the framework – a better way

```cpp
// add.hpp
int add(int i, int j) {
    return i + j;
}
```

Glue code approach

In computer programming, **glue code** is executable code (often source code) that serves solely to "adapt" different parts of code that would otherwise be incompatible.

https://en.wikipedia.org/wiki/Glue_code

```cpp
#include <pybind11/pybind11.h>
#include "add.hpp"

PYBIND11_MODULE(example, m) {
  m.def("add", add, "...", py::arg("i"), py::arg("j"));
}
```

pybind11 syntax as motivation

```cpp
#include <meld/module.hpp>
#include "add.hpp"

DEFINE_MODULE(m, pset) {
  m.declare_transform("add", add).concurrency(unlimited).input("i", "j").output("sum");
}
```

Registering a function with the framework – a better way

```cpp
// add.hpp
int add(int i, int j) {
    return i + j;
}
```

Glue code approach

In computer programming, **glue code** is executable code (often source code) that serves solely to "adapt" different parts of code that would otherwise be incompatible.

https://en.wikipedia.org/wiki/Glue_code

```cpp
#include <pybind11/pybind11.h>
#include "add.hpp"

PYBIND11_MODULE(example, m) {
  m.def("add", add, "...", py::arg("i"), py::arg("j"));
}
```

pybind11 syntax as motivation

```cpp
#include <meld/module.hpp>
#include "add.hpp"

DEFINE_MODULE(m, pset) {
  m.declare_transform("add", add).concurrency(unlimited).input("i", "j").output("sum");
}
```

Registering a function with the framework – a better way

```
// add.hpp
int add(int i, int j) {
    return i + j;
}
```

Glue code approach

In computer programming, **glue code** is executable code (often source code) that serves solely to "adapt" different parts of code that would otherwise be incompatible.

https://en.wikipedia.org/wiki/Glue_code

```
#include <pybind11/pybind11.h>
#include "add.hpp"

PYBIND11_MODULE(example, m) {
  m.def("add", add, "...", py::arg("i"), py::arg("j"));
}
```

pybind11 syntax as motivation

```
#include <meld/module.hpp>
#include "add.hpp"

DEFINE_MODULE(m, pset) {
  m.declare_transform("add", add).concurrency(unlimited).input("i", "j").output("sum");
}
```

No hard-coded processing level required.

Fermilab

Miscellany

- Much cleaner (and finer-grained) binding between user code and the framework
- New approach allows for backwards compatibility with existing framework modules
- Met with DUNE yesterday—definite shift in mental model
- Continue to meet regularly with Brett Viren re. framework/WireCell interface

Next steps

- Flesh out functionality for workflows with filters and “slicing up events”
- Recording framework metadata
- Put as much time as possible toward LDRD for rest of FY22