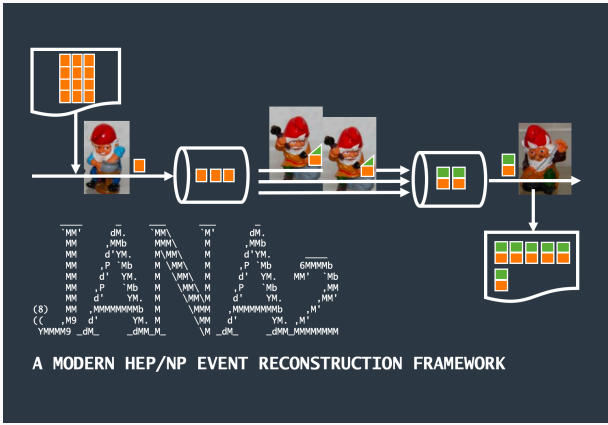
Introduction to JANA

Nathan Brei
nbrei@jlab.org

JLab Software and Computing Workshop
July 7, 2023

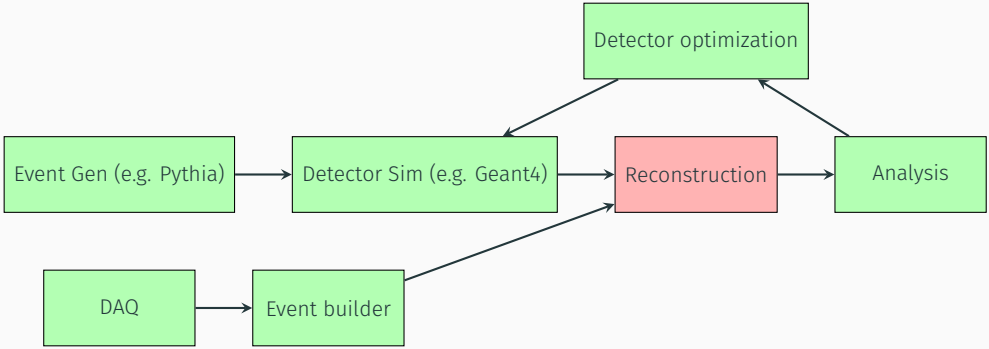


What is JANA?

- JANA is a multithreaded reconstruction framework project written in low-level C++
- Nearly 2 decades of experience behind it
- JANA2 is a rewrite that incorporates modern coding practices. Aims to be clean, lightweight, and user-friendly. “Paying off technical debt”
- JANA2 introduces an innovative dataflow parallelism paradigm under the hood, which has implications for streaming readout and heterogeneous hardware utilization
- Used by EIC ePIC, GlueX, INDRA-ASTRA, BDX, TriDAS+ERSAP+JANA2 streaming DAQ project

Where does JANA fit in the NP software pipeline?

reconstruct : hits -> tracks

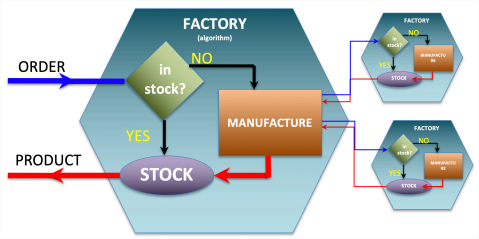


JANA API

- **Event sources**
  Sources accept a resource name (i.e. a file name or network socket), and emit a stream of *`JEvent`* objects. They run single-threaded by default.
- **Factories**
  Factories compute intermediate results, e.g. *`Hit->Track`*. This is essentially a *`map`* pattern. Each worker thread gets its own instance of each factory, so factories can calculate their results independently and without locking.
- **Event processors**
  Event processors consume the entire event stream sequentially after all of the necessary factories have run. They implement the *`reduce`* pattern. They are useful for writing DST files and histograms.
- **Services**
  Services provide access to shared data such as geometry, calibrations, random number generators, etc. They are essentially singleton objects. Services are the only JANA components that are required to be thread-safe.

Factories

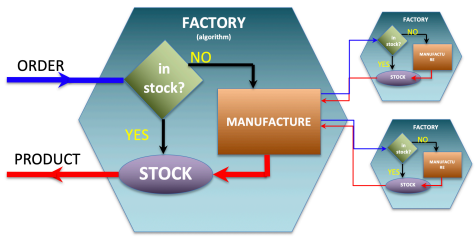
```
#include <JANA/JFactoryT.h>

class SimpleClusterFactory : public JFactoryT<Cluster> {
    // ...


public:
    SimpleClusterFactory() {
        SetTag("ECalClusters");
    }
    void Init() {
        // ...


    }
    void ChangeRun(const std::shared_ptr<const JEvent> &event) {
        // ...


    }
    void Process(const std::shared_ptr<const JEvent> &event) {
        auto hits = event->Get<Hit>("RawECalHits");
        std::vector<Cluster*> clusters = clusterize(hits);
        Set(clusters);
    }
};
```



- Lazy: Factories are only run if their data is requested
- Recursive: Factories can call out to other factories
- Memoized: The result is cached so it's only computed once
- Allowed to be stateful, with limits

Factories

```
#include <JANA/JFactoryT.h>

class SimpleClusterFactory : public JFactoryT<Cluster> {
    double m_threshold = 1.0;
    std::shared_ptr<BFieldMapSvc> m_bfieldmap_svc;
    std::shared_ptr<BFieldMap> m_bfieldmap;

public:
    SimpleClusterFactory() {
        SetTag("ECalClusters");
    }
    void Init() {
        // Obtain parameters and services here
        auto app = GetApplication();
        app->SetDefaultParameter("threshold", m_threshold);
        m_bfieldmap_svc = app->GetService<MagFieldMap>();
    }
    void ChangeRun(const std::shared_ptr<const JEvent> &event) {
        /// Obtain calibrations/conditions here
        auto run_nr = event->GetRunNumber();
        m_bfieldmap = m_bfieldmap_svc->GetBFieldForRun(run_nr);
    }
    void Process(const std::shared_ptr<const JEvent> &event) {
        auto hits = event->Get<Hit>("RawECalHits");
        std::vector<Cluster*> clusters = clusterize(hits);
        Set(clusters);
    }
};
```



- Lazy: Factories are only run if their data is requested
- Recursive: Factories can call out to other factories
- Memoized: The result is cached so it's only computed once
- Allowed to be stateful, with limits

Multifactories

```
#include <JANA/JMultifactory.h>

class ClusterFactory : public JMultifactory {

public:
    ClusterFactory() {
        DeclareOutput<Cluster>("ECalClusters");
        DeclareOutput<ClusterAssoc>("ECalClusterAssocs");
    }
    void Init() {
        // ...
    }
    void ChangeRun(const std::shared_ptr<const JEvent> &event) {
        // ...
    }
    void Process(const std::shared_ptr<const JEvent> &event) {
        auto hits = event->Get<Hit>("RawECalHits");
        auto [clusters, assocs] = clusterize(hits);
        SetData("ECalClusters", clusters);
        SetData("ECalClusterAssocs", assocs);
    }
};
```

- New as of JANA2 v2.1.0!
- Use this when you have multiple outputs originating from one algorithm
- Tiny performance penalty compared to JFactoryT

Sources

```cpp
#include <JANA/JEventSource.h>

class MyEventSource : public JEventSource {
    MyReader m_reader;
    int m_total_events;
public:
    MyEventSource(std::string resource_name, JApplication* app)
        : JEventSource(resource_name, app) {}

    void Open() override {
        m_reader.openFile(GetResourceName());
        m_total_events = m_reader.getEntries();
    }
    void GetEvent(std::shared_ptr<JEvent>) {
        if (GetEventCount() == m_total_events) {
            throw RETURN_STATUS::kNO_MORE_EVENTS;
        }

        auto event_data = m_reader.read(GetEventCount());
        event->SetEventNumber(event_data.getEventNumber());
        event->SetRunNumber(event_data.getRunNumber());

        auto ecalRawHits = event_data.get<RawHit>("EcalRawHits");
        event->Insert<RawHit>(ecalRawHits, "EcalRawHits");
        // etc...
        // Note: Use the Visitor Pattern to do this more generally
    }
};
```

- JEventSources are auto-mapped to resource names via *`JEventSourceGenerator`*.
- Supports polling a network socket
- Supports notifying when an event finishes
- Does NOT support random access yet

Processors

```
#include <JANA/JEventProcessorSequentialRoot.h>

class MyProcessor: public JEventProcessorSequentialRoot {
private:
    // Data objects we will need from JANA e.g.
    PrefetchT<edm4hep::SimCalorimeterHit> rawhits = {this, "EcalRawHits"};

    // Declare histogram and tree pointers here. e.g.
    TH1D* hEraw = nullptr;

public:
    MyProcessor() { SetTypeName(NAME_OF_THIS); }

    void InitWithGlobalRootLock() {
        auto rootfile_svc = GetApplication()->GetService<RootFileSvc>();
        auto rootfile = rootfile_svc->GetHistFile();
        rootfile->mkdir("myFirstPlugin")->cd();
        hEraw = new TH1D("Eraw", "BEMC hit energy (raw)", 100, 0, 0.075);
    }
    void ProcessSequential(const std::shared_ptr<const JEvent>& event) {
        for(auto hit : rawhits())
            hEraw->Fill( hit->getEnergy());
    }
    void FinishWithGlobalRootLock() {

    }
};
```

- Unlike factories, processors are singletons and see the entire event stream
- Processors run sequentially, so do as much work as possible in factories instead (“critical section”)
- Each processor should produce one kind of output (single responsibility principle)
- You can choose which processors are included at runtime by using plugins

Event data model

- JANA factories can produce any datatype you need!
- JANA provides a simple, optional EDM base class called *JObject*
  - Reports its own fields, their contents, and their documentation
  - Maintains a lightweight graph of associations with other *JObjects*
  - Works well with external tools for inspecting/visualizing/debugging the event stream
- JANA supports using a different base class, e.g. *TObject*.
  - Your EDM classes can optionally multiply-inherit
  - Call *JFactory::EnableGetAs<BaseT>()* to register each base class with the factory
  - Call *JFactory::GetAs<BaseT>* to retrieve its data as *std::vector<BaseT*>*
  - Call *JEvent::GetAllChildren<BaseT>()* to retrieve all event data as *std::map<(typename,tagname), BaseT*>*.

PODIO event data model

```
options :
  getSyntax: False
  exposePODMembers: True
  includeSubfolder: True

datatypes :
  ExampleHit :
    Description : "Example Hit"
    Author : "B. Hegner"
    Members:
      - unsigned long long cellID      // cellID
      - double x       // x-coordinate
      - double y       // y-coordinate
      - double z       // z-coordinate
      - double energy // measured energy deposit

  ExampleCluster :
    Description : "Cluster"
    Author : "B. Hegner"
    Members:
      - double energy // cluster energy
    OneToManyRelations:
      - ExampleHit Hits // Hits used to make this
      - ExampleCluster Clusters // Subclusters
```

- A heavy-duty library for generating an EDM from a YAML file
- I/O to/from C++ object trees, Python object trees, and RDataframe
- Enforces correct ownership, constness, inter-object references
- JANA now has first-class support for PODIO including the ability to access collections directly via `event->GetCollection(name)` and `JFactoryPodioT::SetCollection`.

Plugins

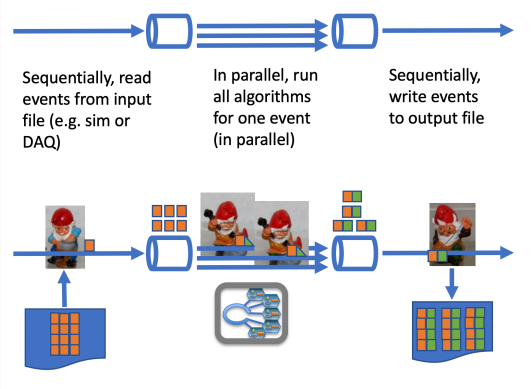
```
#include <JANA/JApplication.h>

extern "C" {
void InitPlugin(JApplication *app) {
    InitJANAPlugin(app);
    app->Add(new JFactoryGeneratorT<MyClusterFactory>);
}

jana -Pplugins=podio,reco,tracking $PATH_TO_INPUTFILE
```

- Plugins are one mechanism for enforcing modular design
- Each plugin is its own (dynamically loaded) shared library.
- Each plugin can register any number of components (i.e. sources, factories, processors, services) with JANA's *`JApplication`*.
- Plugins can depend on other plugins, e.g. *`tracking`* could depend on *`acts`*.
- Plugins can also be useful for optionally integrating common analysis tasks, e.g. monitoring plots, and integration tests.

Looking under the hood: Dataflow parallelism

- JANA2 avoids locks in favor of a network of ‘queues and arrows’, analogous to an assembly line in a real-world factory
- Each worker thread is assigned an arrow, conceptually like a conveyor belt
- Arrows can be either sequential (only one worker at a time) or parallel (many workers can perform these tasks at once)
- Between each arrow lies a queue. Workers don’t need to coordinate with each other, just pick up new tasks off their input queue and put finish tasks onto their output queue.
- If a queue backs up, some of the workers move to a different arrow (backpressure!)

A graphical notation for queues and arrows



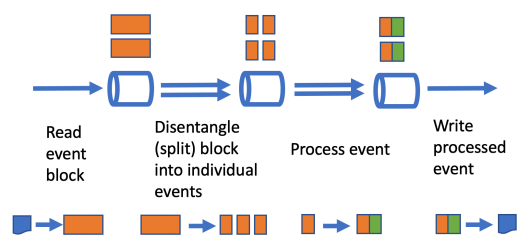
- Arrows are allowed to produce a different number of output objects than they consume, and to choose from different output queues
- This allows us to do things like
  - Split/merge
  - Gather/scatter
  - Filtering
  - Streaming joins
- This lets us have different units of parallelism at different points in our computation
  - Event blocks
  - Subevents

Example: Event blocks

Problem: The GlueX event builder stores 'entangled' events in blocks of 40. Disentangling is expensive and needs to be done in parallel. In JANA1, this is accomplished by a separate thread team that JANA knows nothing about.

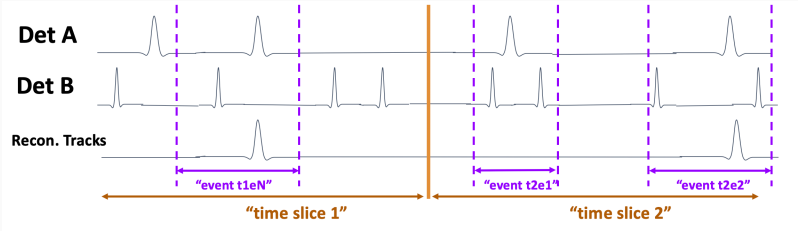
Solution: Flatmap pattern:

```
block_source.next :: () -> block
block_source.disentangle :: block -> [event]
```



Example: SRO event selection and filtering

Problem: A ‘triggerless’ streaming system receives timeslices directly from the DAQ (instead of events from the event builder), and performs partial reconstruction directly on the timeslices in order to identify event boundaries.

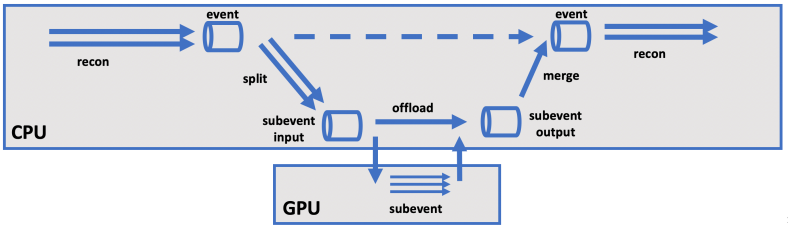


Example: Subevents + GPU offloading

Problem: Offloading the data to a GPU/TPU/etc is a sequential bottleneck! We want to batch our data so that we can utilize the full parallelism of the heterogeneous hardware.

Solution: Split/merge pattern:

- Split each event into subtasks
- Batch subtasks together, possibly across event boundaries
- Send the batched subtasks to the hardware and wait for the results
- Attach the subtask results to their corresponding event



Thank you! That is all.