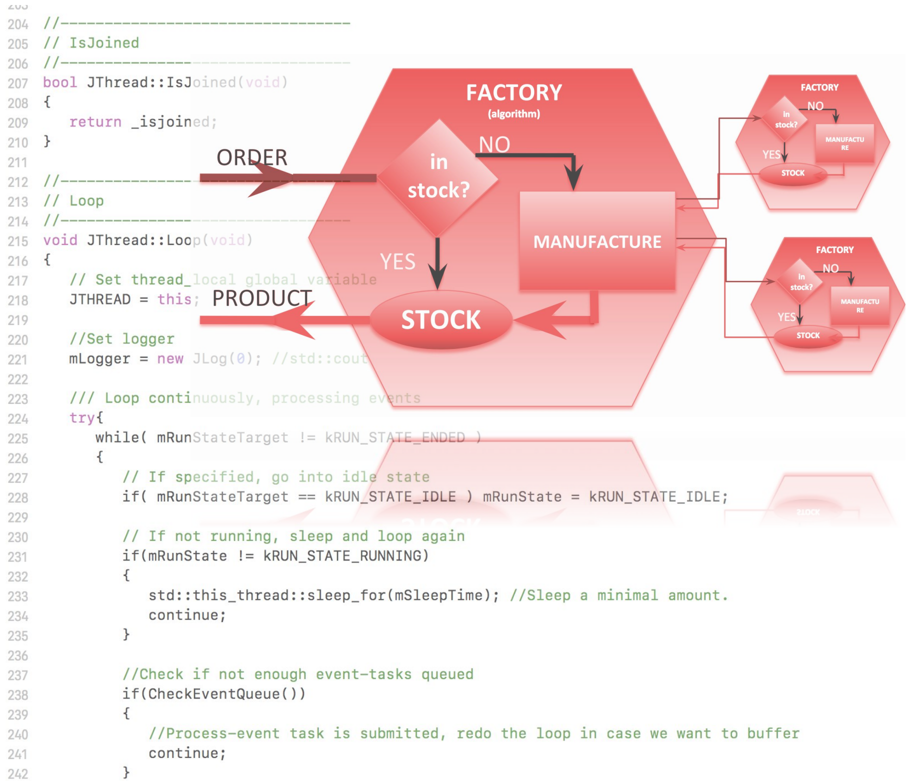
JANA2: Multi-threaded Event Reconstruction

David Lawrence
Jefferson Lab

September 21, 2022

HSF Framework Working Group



Jefferson Lab

U.S. DEPARTMENT OF ENERGY | Office of Science

JSA

The JANA Framework

- JANA is a multithreaded reconstruction framework project with nearly 2 decades of experience behind it

- JANA2 is a rewrite incorporating more modern coding and CS practices and improving on the original using lessons learned
  - Streaming DAQ and Heterogeneous hardware support strongly considered in redesign

Projects using JANA
- GlueX
- INDRA-ASTRA *(near-realtime calibrations using AI/ML)*
- BDX
- TriDAS (+ERSAP) + JANA2 Streaming DAQ

GlueX Computing Needs

	2017 (low intensity GlueX)	2018 (low intensity GlueX)	2019 (PrimEx)	2019 (high intensity GlueX)
Real Data	1.2PB	6.3PB	1.3PB	3.1PB
MC Data	0.1PB	0.38PB	0.16PB	0.3PB
Total Data	**1.3PB**	**6.6PB**	**1.4PB**	**3.4PB**
Real Data CPU	21.3Mhr	67.2Mhr	6.4Mhr	39.6Mhr
MC CPU	3.0Mhr	11.3MHr	1.2Mhr	8.0Mhr
Total CPU	**24.3PB**	**78.4Mhr**	**7.6Mhr**	**47.5Mhr**

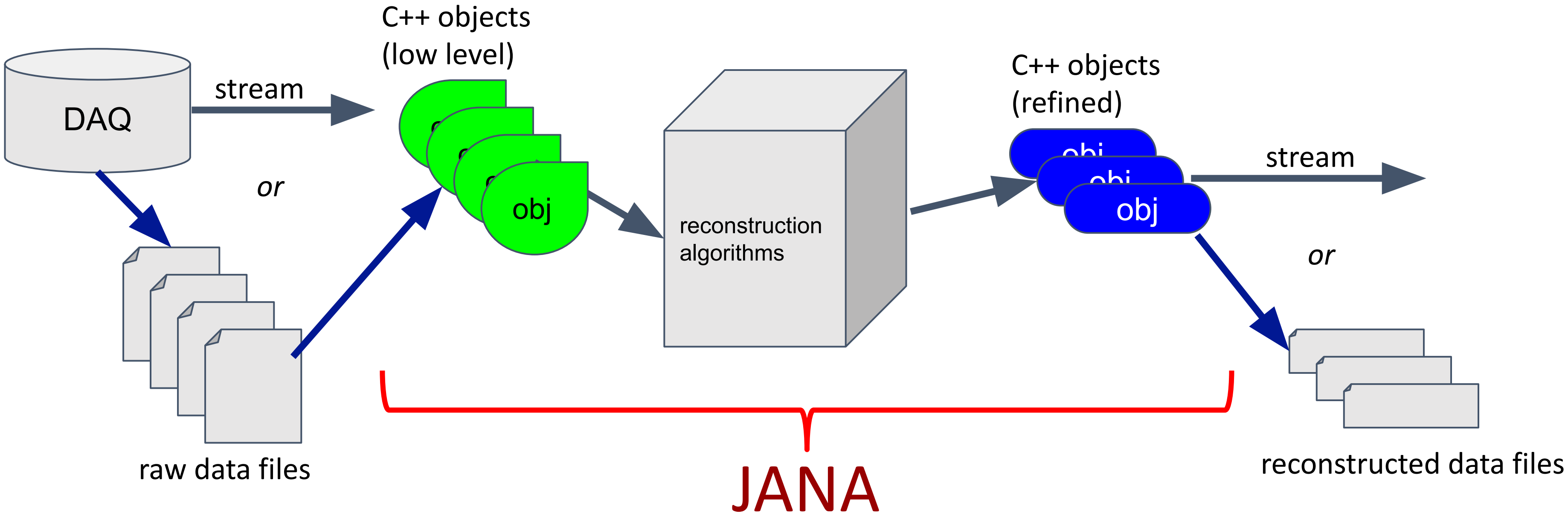
Anticipate 2018 data will be processed by end of summer 2019

Projection for out-years of GlueX High Intensity running at 32 weeks/year

	Out - years (high intensity GlueX)
Real Data	16.2PB
MC Data	1.4PB
Total Data	**17.6PB**
Real Data CPU	125.6Mhr
MC CPU	36.5Mhr
Total CPU	**162.1Mhr**

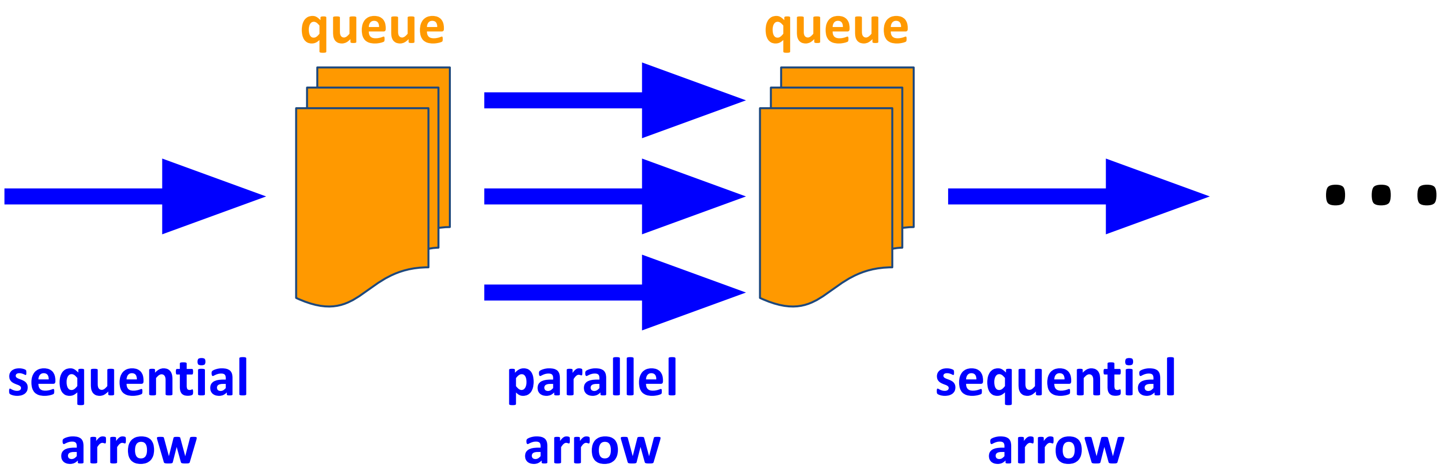
Event size:
12-13kB

JANA's Role in Data Processing



JANA2 arrows separate sequential and parallel tasks

- CPU intensive event reconstruction will be done as a parallel arrow
- Other tasks (e.g. I/O) can be done as a sequential arrow
- Fewer locks in user code allows framework to better optimize workflow

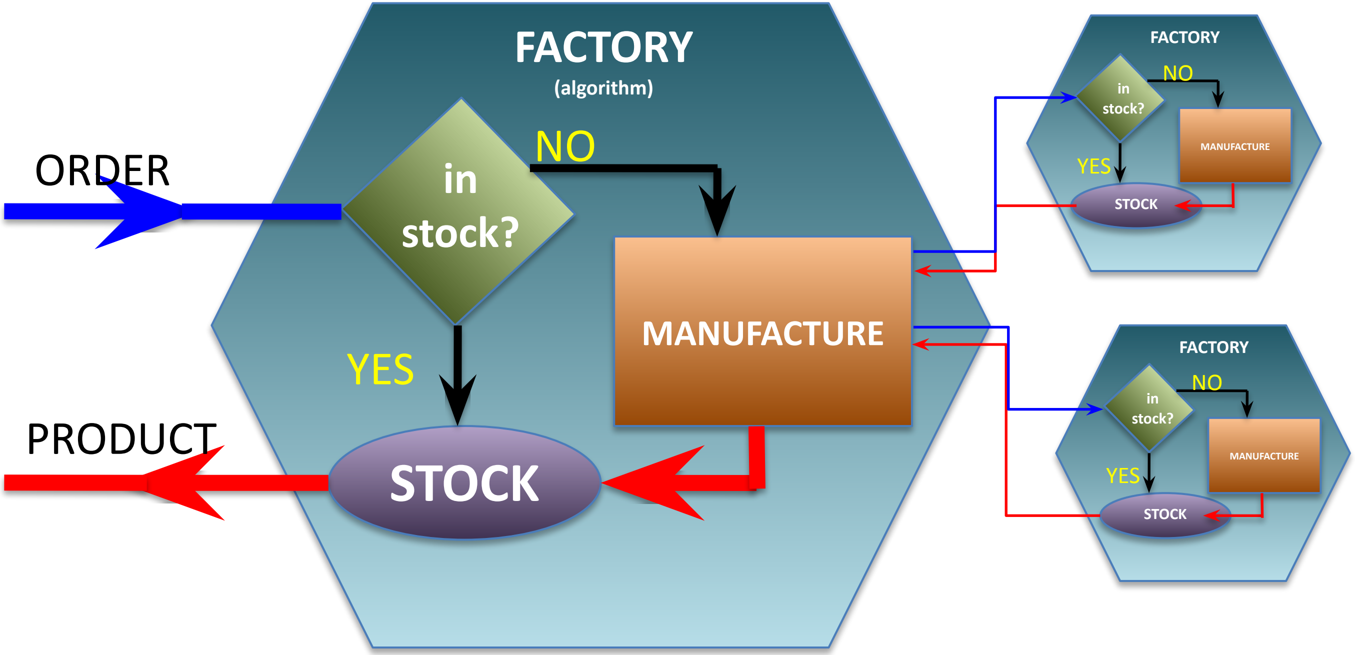


Reactive/Dataflow Programming

EPSCI
Experimental Physics Software and Computing Infrastructure

- Data is presented to arrow in the form of a queue
- Arrow transforms data and places it in downstream queue
- Minimal synchronization time spent in accessing queues
- Course tasks within arrow can eliminate most or all other synchronization points

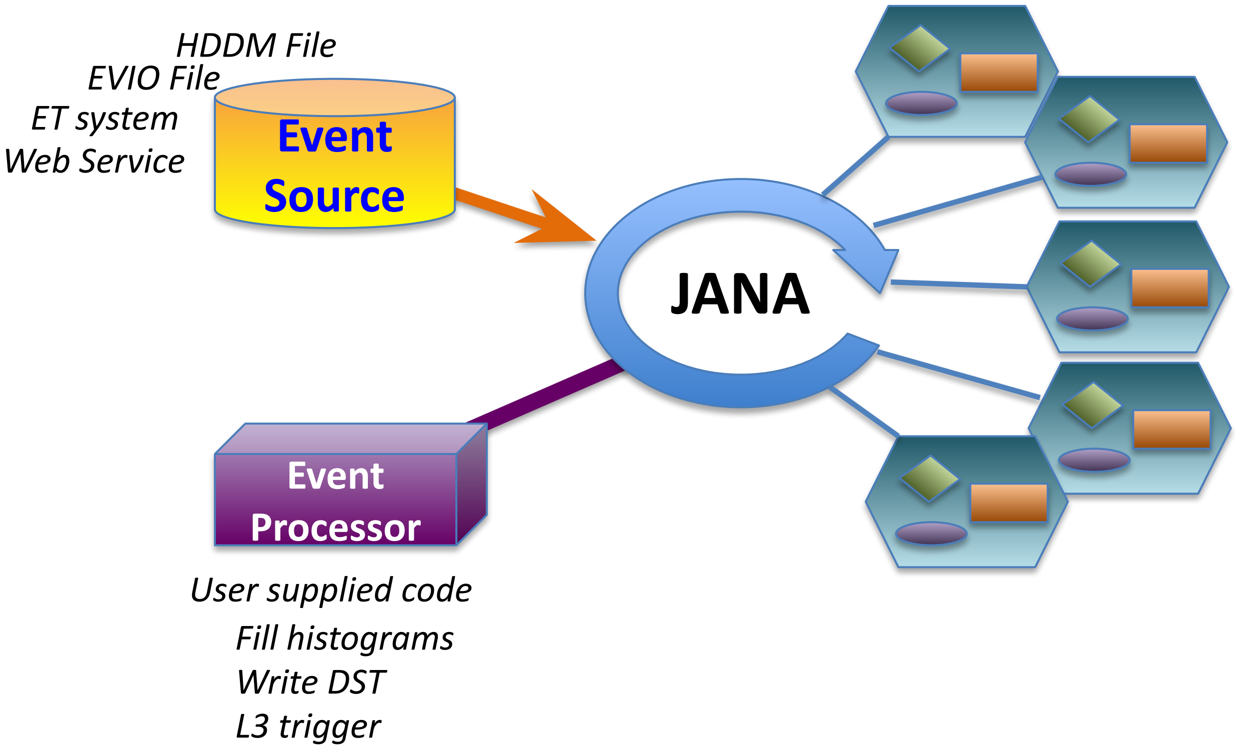
Factory Model



Data on demand = Don't do it unless you need it
Stock = Don't do it twice

Conservation of CPU cycles!

Complete Event Reconstruction in JANA



Framework has a layer that directs object requests to the factory that completes it

Multiple algorithms (factories) may exist in the same program that produce the same type of data objects

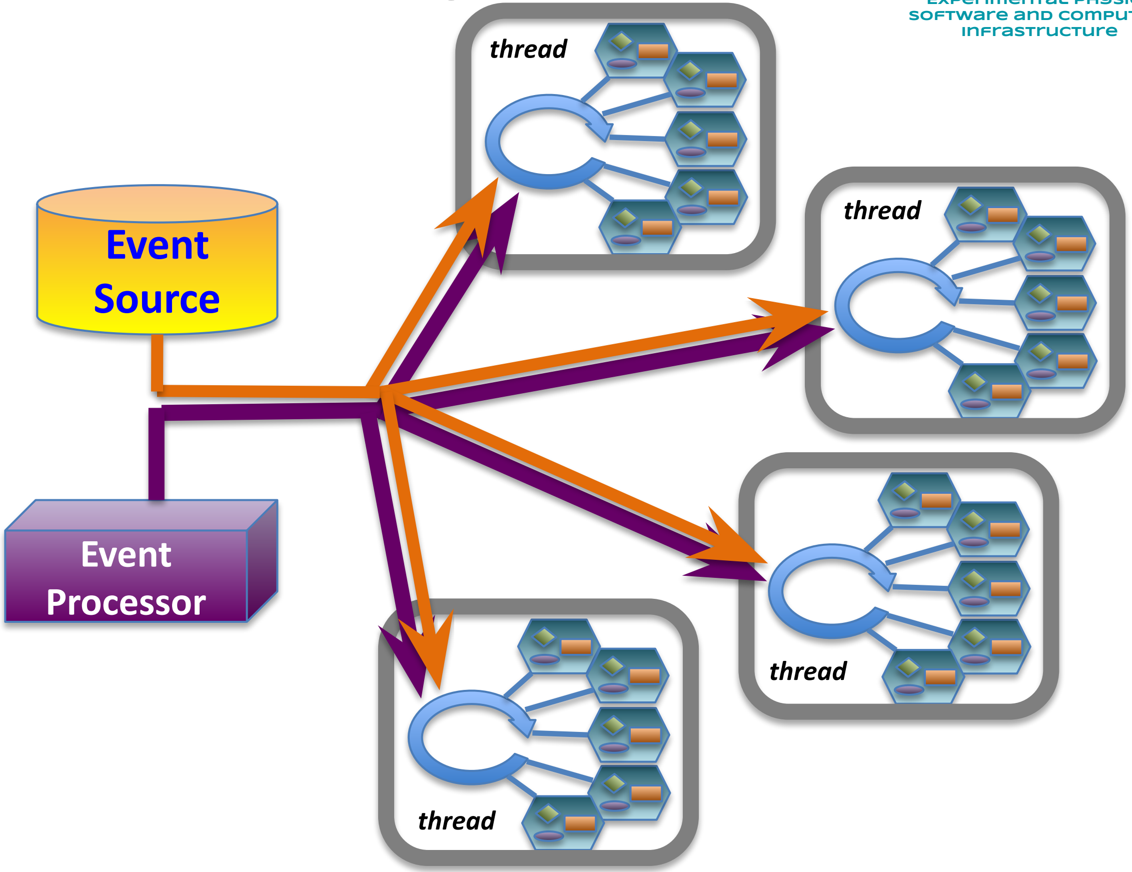
This allows the framework to easily redirect requests to alternate algorithms specified by the user at run time

Multi-threading

- A complete set of factories is assigned to an event giving it exclusive use while that event is processed

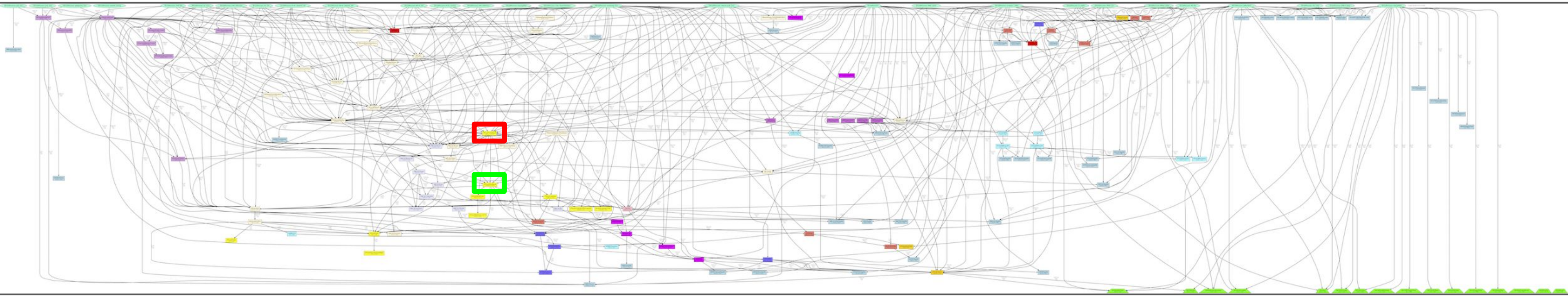
- Factories only work with other factories in the same thread eliminating the need for expensive mutex locking within the factories

- All events are seen by all Event Processors (multiple processors can exist in a program)



Large experiments have complex call graphs

GlueX Reconstruction - automated rendering via janadot plugin



Run 42513:

Physics Production mode Trigger: FCAL_BCAL_PS_m9.conf

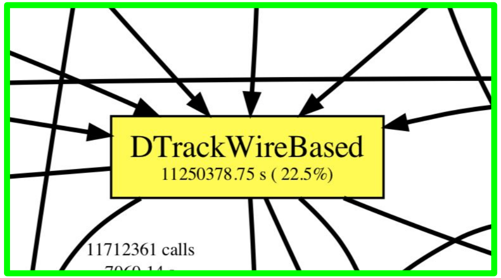
setup: hd_all.tsg

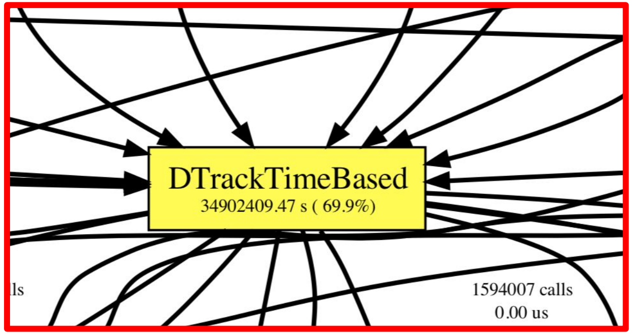
0/90 PERP 90

JD70-100 58um

TPOL Be 75um

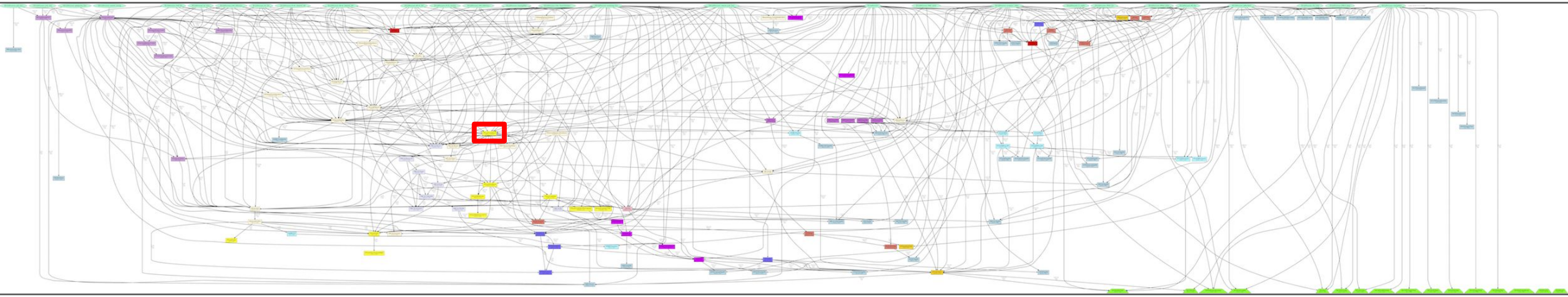
beam looks stable





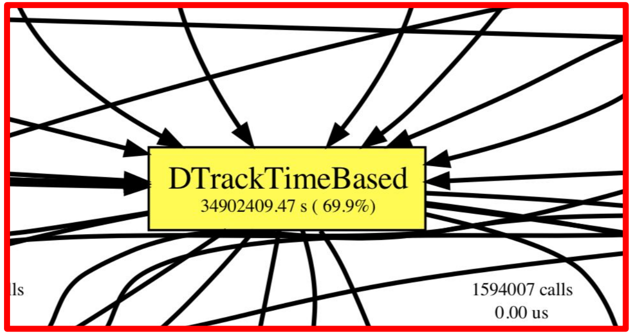
Large experiments have complex call graphs

GlueX Reconstruction - automated rendering via janadot plugin



Modular design:

- Factories (algorithms) need to know what they depend on
- Factories do ***not*** need to know what depends on them
- Dependencies do ***not*** need to be specified at higher level



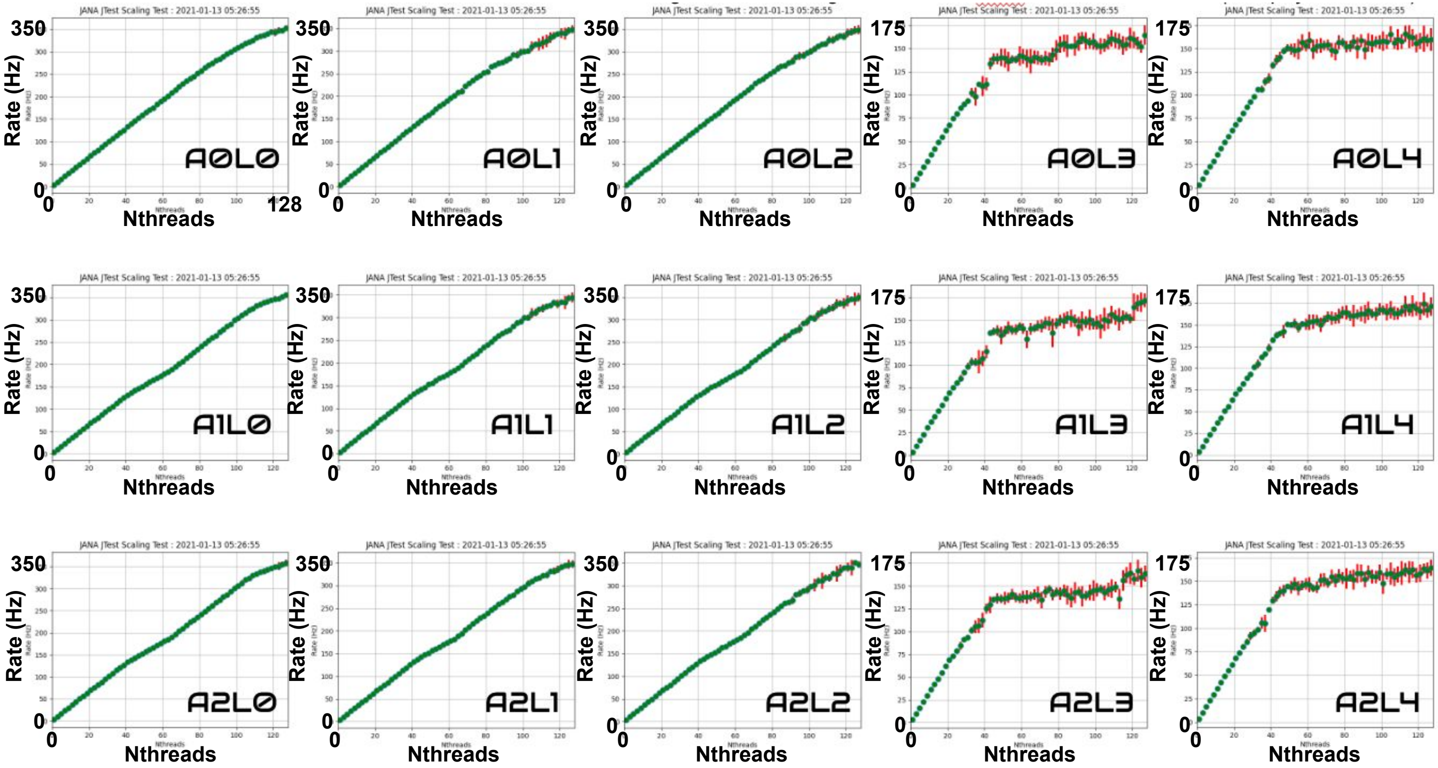
Features Added in JANA2

EPSCI
Experimental Physics Software and Computing Infrastructure

- Better use of “modern” C++ features
  - thread model via C++ language (introduced in c++11)
  - lock guards
  - shared pointers
  - lambda functions
- Generalized use of threads (pool)
  - multiple queues
  - arrows (sequential or parallel)
- NUMA awareness
- Python API (both embedded and as an extension)

Multiple Affinity and Locality strategies

OS, chip type, memory architecture, and nature of job all can affect which model yields optimal performance



JANA2 Scaling test: PSC Bridges-2 RM Two AMD EPYC 7742 CPUS (128 physical cores)

```
enum class
AffinityStrategy {
      None,
      MemoryBound,
      ComputeBound };
```

```
enum class
LocalityStrategy {
      Global,
      SocketLocal,
      NumaDomainLocal,
      CoreLocal,
      CpuLocal };
```

Configurable at run time via Config. Parameters

Gaudi Property = JANA Config. Parameter

EPSCI
Experimental Physics Software and Computing Infrastructure

Algorithms need parameters that can be configured at run time.

Gaudi stores configurable parameters in **`Gaudi::Property`** members

```
Gaudi::Property<double> m_timeResolution{this, "timeResolution", 10}; // todo : add units
Gaudi::Property<double> m_threshold{this, "threshold", 0. * Gaudi::Units::keV};
```

```
StatusCode sc          = m_gaussDist.initialize(randSvc, Rndm::Gauss(0.0, m_timeResolution.value()));
```

JANA stores configurable parameters in central service and copies to local variables (e.g. members)

```
        config.threshold = 0.0;
        config.timeResolution = 8.0 * dd4hep::ns; // correct units?

        app->SetDefaultParameter("RPOTS:ForwardRomanPotRawHits:threshold",      config.threshold );
        app->SetDefaultParameter("RPOTS:ForwardRomanPotRawHits:timeResolution",config.timeResolution );
```

JANA maintains list of all configuration parameters and their defaults.

- values can be dumped to config. file at end of job for use on subsequent jobs
- values that differ from defaults can be flagged
- values that have no implementation in code (typos) can be flagged

Connecting Algorithms

EPSCI
Experimental Physics Software and Computing Infrastructure

Gaudi: *reconstruction.py*

explicit piping

```python
## Roman pots
ffi_romanpot_coll = SimTrackerHitsCollector(
    "ffi_romanpot_coll",
    inputSimTrackerHits=forward_romanpot_collections,
    outputSimTrackerHits="ForwardRomanPotAllHits",
)
algorithms.append(ffi_romanpot_coll)
ffi_romanpot_digi = TrackerDigi(
    "ffi_romanpot_digi",
    inputHitCollection=ffi_romanpot_coll.outputSimTrackerHits,
    outputHitCollection="ForwardRomanPotRawHits",
    timeResolution=8,
)
algorithms.append(ffi_romanpot_digi)

ffi_romanpot_reco = TrackerHitReconstruction(
    "ffi_romanpot_reco",
    inputHitCollection=ffi_romanpot_digi.outputHitCollection,
    outputHitCollection="ForwardRomanPotRecHits",
)
algorithms.append(ffi_romanpot_reco)

ffi_romanpot_parts = FarForwardParticles(
    "ffi_romanpot_parts",
    inputCollection=ffi_romanpot_reco.outputHitCollection,
    outputCollection="ForwardRomanPotParticles",
)
algorithms.append(ffi_romanpot_parts)
```

JANA: *factory C++*

implicit piping

```cpp
    //------------------------------------------
    // Process
    void Process(const std::shared_ptr<const JEvent> &event) override {
        // Get inputs
        auto sim_hits1 = event->Get<edm4hep::SimTrackerHit>("ForwardRomanPotHits1");
        auto sim_hits2 = event->Get<edm4hep::SimTrackerHit>("ForwardRomanPotHits2");
        auto sim_hits_all = sim_hits1;
        sim_hits_all.insert( sim_hits_all.end(), sim_hits2.begin(), sim_hits2.end() );

        // Call Process for generic algorithm
        auto rawhits = produce( sim_hits_all );

        // Hand owner of algorithm objects over to JANA
        Set(rawhits);
    }
```

```cpp
    //------------------------------------------
    // Process
    void Process(const std::shared_ptr<const JEvent> &event) override {
        // Get inputs
        auto rawhits = event->Get<edm4eic::RawTrackerHit>("ForwardRomanPotRawHits");

        // Call Process for generic algorithm
        std::vector<edm4eic::TrackerHit*> trackerhits;
        for( auto rawhit : rawhits) trackerhits.push_back(produce(rawhit));

        // Hand owner of algorithm objects over to JANA
        Set(trackerhits);
    }
```

Primary Key for Connecting Algorithms

Gaudi uses collection name (string) as the primary key

python code

```
ffi_romanpot_digi = TrackerDigi(
    "ffi_romanpot_digi",
    inputHitCollection=ffi_romanpot_coll.outputSimTrackerHits,
    outputHitCollection="ForwardRomanPotRawHits",
    timeResolution=8,
)
```

C++ code

```
DataHandle<edm4hep::SimTrackerHitCollection> m_inputHitCollection{"inputHitCollection", Gaudi::DataHandle::Reader,
                                                                   this};
```

JANA uses typeid as the primary key and string as secondary key (“tag”)

```
auto sim_hits1 = event->Get<edm4hep::SimTrackerHit>("ForwardRomanPotHits1");
```

primary key

secondary key

errors in primary key are caught at compile time.

errors in secondary key are caught at run time.

n.b. C++ linker does ***not*** link algorithms together. Run time list is searched using typeid

Summary

- JANA is a multithreaded framework project with nearly 2 decades of experience behind it
- JANA2 is a rewrite incorporating more modern coding and CS practices and improving on the original using lessons learned
  - Streaming DAQ and Heterogeneous hardware support strongly considered in redesign
- JANA2 has been selected for use with the EIC first detector (ePIC) and is currently being implemented there

Github: https://github.com/JeffersonLab/JANA2
Documentation: https://jeffersonlab.github.io/JANA2/
Example project: https://github.com/faustus123/EIC_JANA_Example

Publications:

https://arxiv.org/abs/2202.03085 *Streaming readout for next generation electron scattering experiments*
https://doi.org/10.1051/epjconf/202125104011 *Streaming Readout of the CLAS12 Forward Tagger Using TriDAS and JANA2*
https://doi.org/10.1051/epjconf/202024501022 *JANA2 Framework for Event Based and Triggerless Data Processing*
https://doi.org/10.1051/epjconf/202024507037 *Offsite Data Processing for the GlueX Experiment*
https://iopscience.iop.org/article/10.1088/1742-6596/119/4/042018 *Multi-threaded event reconstruction with JANA*
https://pos.sissa.it/070/062 *Multi-threaded event processing with JANA*
https://iopscience.iop.org/article/10.1088/1742-6596/219/4/042011 *The JANA calibrations and conditions database API*
https://iopscience.iop.org/article/10.1088/1742-6596/1525/1/012032 *JANA2: Multithreaded Event Reconstruction*

Backups

Python support in JANA2

EPSCI
Experimental Physics Software and Computing Infrastructure

As pure python script

python3 jana.py

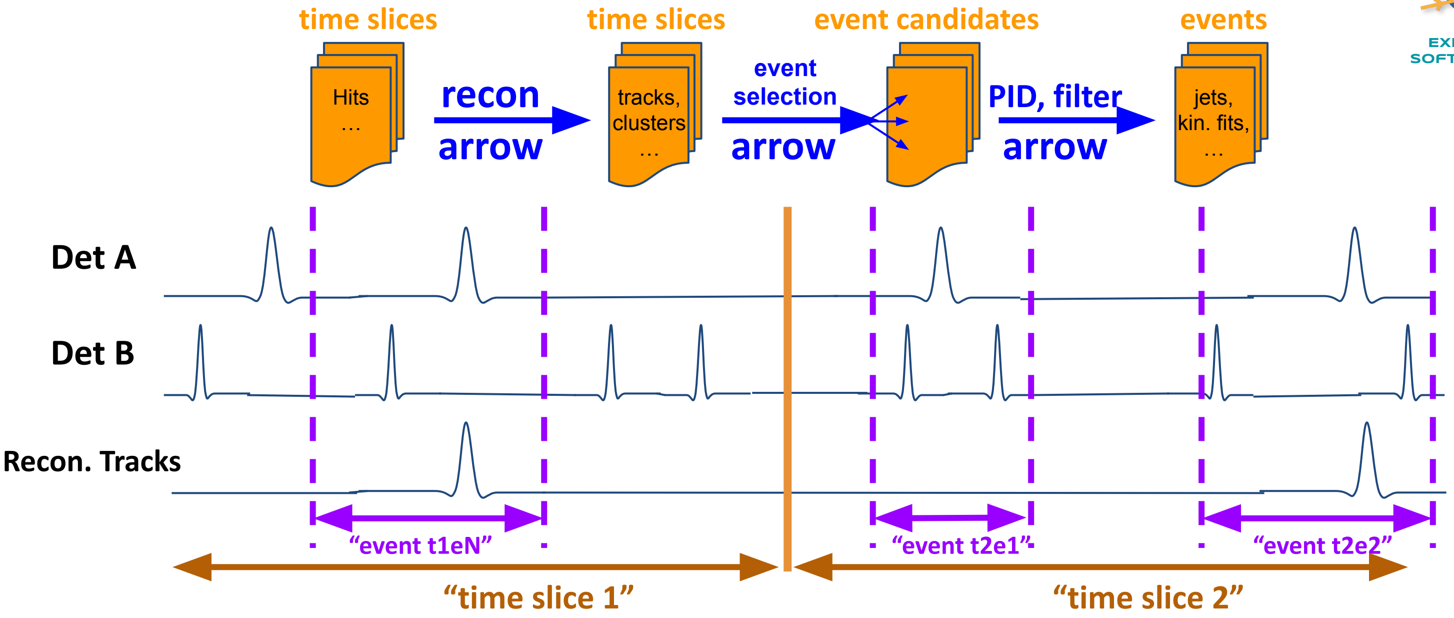
```
# This example JANA python script
import time
import jana

print('Hello from jana.py!!!')

# Turn off JANA's standard ticker so we can print our own updates
jana.SetTicker(False)

# Wait for 4 seconds before allowing processing to start
for i in range(1,5):
        time.sleep(1)
        print(" waiting ... %d" % (4-i))

# Start event processing
jana.Start()

# Wait for 5 seconds while processing events
for i in range(1,6):
        time.sleep(1)
        print(" running ... %d  (Nevents: %d)" % (i, jana.GetNeventsProcessed()))

# Tell program to quit gracefully
jana.Quit()
```

As embedded interpreter

jana -PPLUGINS=janapy -PJANA_PYTHON_FILE=myfile.py

```
# This is a simple example JANA python script. It shows how to add plugins
# and set configuration parameters. Event processing will start once this
# script exits.
import jana

jana.AddPlugin('JTest')
jana.SetParameterValue('jana:nevents', 200)

jana.Run()
```

Streaming Data

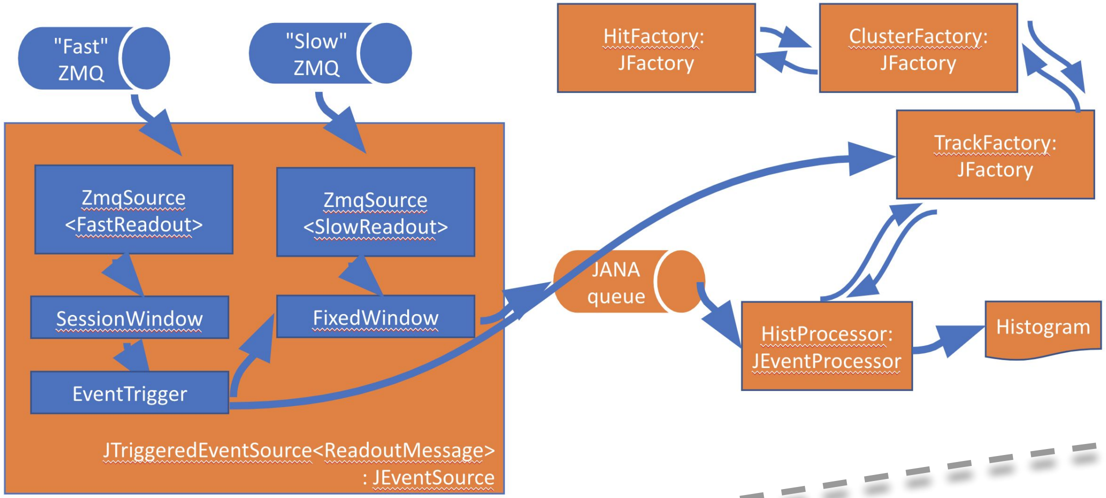


- JANA2 has streaming readout features tested under multiple detector setups and in beam conditions*
- EPSCI has multiple experts working on streaming DAQ systems in same group as JANA2 developers
- EPSCI works closely with the *JLab Fast Electronics Group* (Chris Cuevas) and partners routinely in performance testing in the DAQ Lab.

*Publications relevant to Streaming:

https://arxiv.org/abs/2202.03085 *Streaming readout for next generation electron scattering experiments*
https://doi.org/10.1051/epjconf/202125104011 *Streaming Readout of the CLAS12 Forward Tagger Using TriDAS and JANA2*
https://doi.org/10.1051/epjconf/202024501022 *JANA2 Framework for Event Based and Triggerless Data Processing*
https://doi.org/10.2172/1735849 *Evaluation & Development of Algorithms & Techniques for Streaming Detector Readout*

Streaming Readout



INDRA-ASTRA initiative:

- Software trigger
- Multi-flavored stream merging
- Event building

Support for Heterogeneous Hardware

- Sub-event level parallelism
  - Run ML on GPU or TPU

21

What the user needs to know:

EPSCI
Experimental Physics Software and Computing Infrastructure

```
auto tracks = jevent->Get<DTrack>();

for(auto t : tracks){

  // ... do something with const DTrack* t

}
```

*vector<const *DTrack> tracks*

22

Data on Demand => Software Trigger

EPSCI
Experimental Physics Software and Computing Infrastructure

Event by event decision on whether to activate a factory:

Software triggers may have multiple “keep” or “discard” conditions that may be probed in order of CPU cost

```
// Getting hit objects is cheap so we check that first
auto NcaloHits = jevent->Get<CaloHit>().size();
if( NcaloHits>minCaloHits ){

    keep_event = true;

// Tracks factory only activated if not already keeping event
}else if( jevent->Get<Tracks>().size() > minTrackHits ) {

    keep_event = true;

}
```

If an alternate factory is desired:
(i.e. algorithm)

```
auto tracks = jevent->Get<DTrack>("MyTest");
```

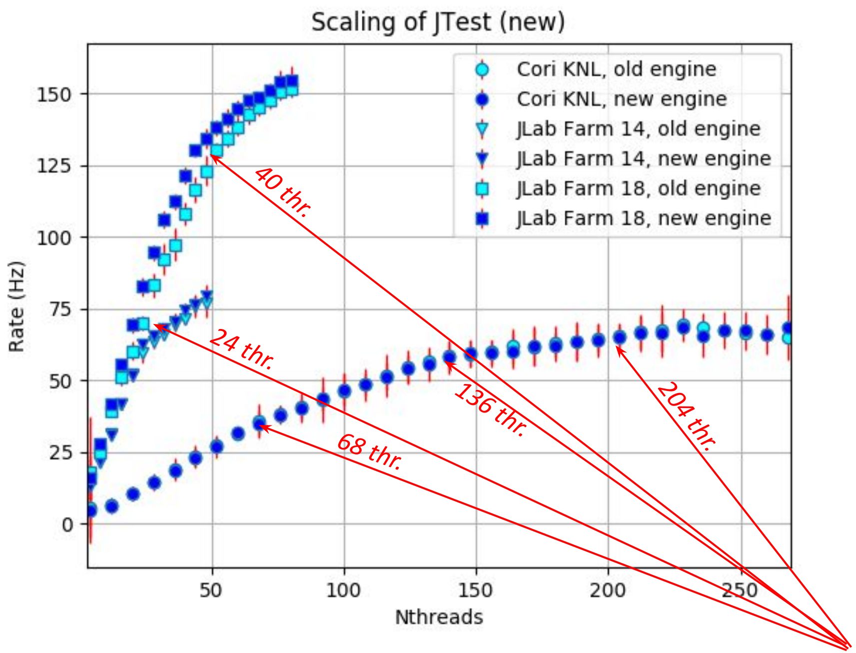
or, even better

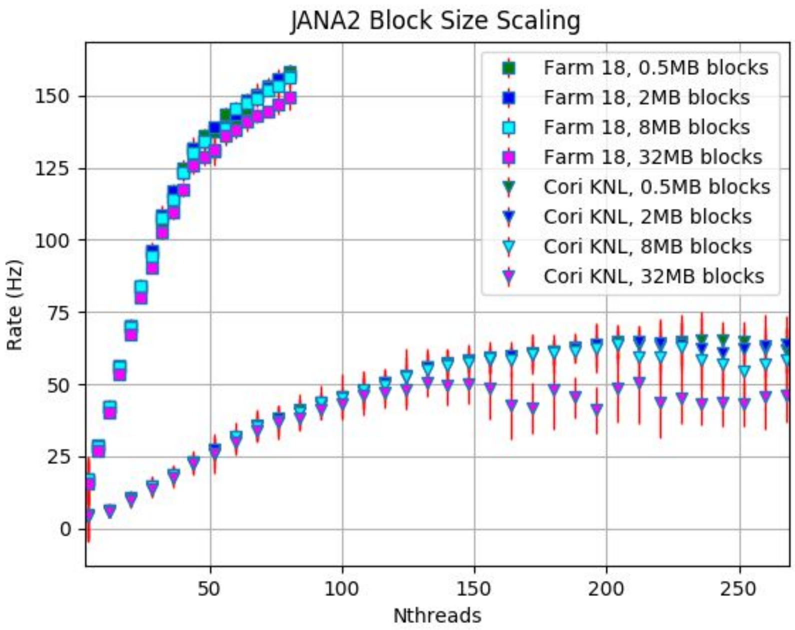
set configuration parameter: **DTrack:DEFTAG=MyTest**

- Configuration parameters are set at run time
- NAME:DEFTAG is special and tells JANA to re-route ALL requests for objects of type NAME to the specified factory.

24

JANA2 Scaling Tests (JLab + NERSC)





kinks indicate hardware boundaries

25

```
TOPOLOGY STATUS
---------------
Thread team size [count]:    4
Total uptime [s]:            50.09
Uptime delta [s]:            0.5002
Completed events [count]:    587
Inst throughput [Hz]:        14
Avg throughput [Hz]:         11.7
Sequential bottleneck [Hz]:  335
Parallel bottleneck [Hz]:    11.9
Efficiency [0..1]:           0.986

+-------------------------+------------+--------+-----+---------+-------+--------+---------+-------------+
|          Name           |   Status   |  Type  | Par | Threads | Chunk | Thresh | Pending |  Completed  |
+-------------------------+------------+--------+-----+---------+-------+--------+---------+-------------+
| dummy_evt_src           | Running    | Source |  F  |       0 |    16 |      - |       - |         672 |
| processors              | Running    | Sink   |  T  |       4 |     1 |    500 |      81 |         587 |
+-------------------------+------------+--------+-----+---------+-------+--------+---------+-------------+

+-------------------------+-------------+--------------+----------------+--------------+----------------+
|          Name           | Avg latency | Inst latency | Queue latency  | Queue visits | Queue overhead |
|                         | [ms/event]  |  [ms/event]  |   [ms/visit]   |   [count]    |     [0..1]     |
+-------------------------+-------------+--------------+----------------+--------------+----------------+
| dummy_evt_src           |        2.98 |         1.03 |        0.00415 |           42 |       8.71e-05 |
| processors              |         337 |          321 |        0.00883 |         1450 |       6.48e-05 |
+-------------------------+-------------+--------------+----------------+--------------+----------------+

+----+-------------------------+-------------+------------+-----------+----------------+------------------+
| ID | Last arrow name         | Useful time | Retry time | Idle time | Scheduler time | Scheduler visits |
|    |                         |    [ms]     |    [ms]    |   [ms]    |      [ms]      |     [count]      |
+----+-------------------------+-------------+------------+-----------+----------------+------------------+
|  0 | processors              |         623 |          0 |         0 |       0.000576 |               76 |
|  1 | processors              |         622 |          0 |         0 |       0.000624 |              138 |
|  2 | processors              |         668 |          0 |         0 |       0.000553 |              131 |
|  3 | processors              |         734 |          0 |         0 |       0.000606 |              125 |
+----+-------------------------+-------------+------------+-----------+----------------+------------------+
```

EPSCI
Experimental Physics Software and Computing Infrastructure

JANA2 now has much better built-in diagnostics compared to the original JANA.

This helps pinpoint bottlenecks, especially in more complex systems

Boilerplate code generation

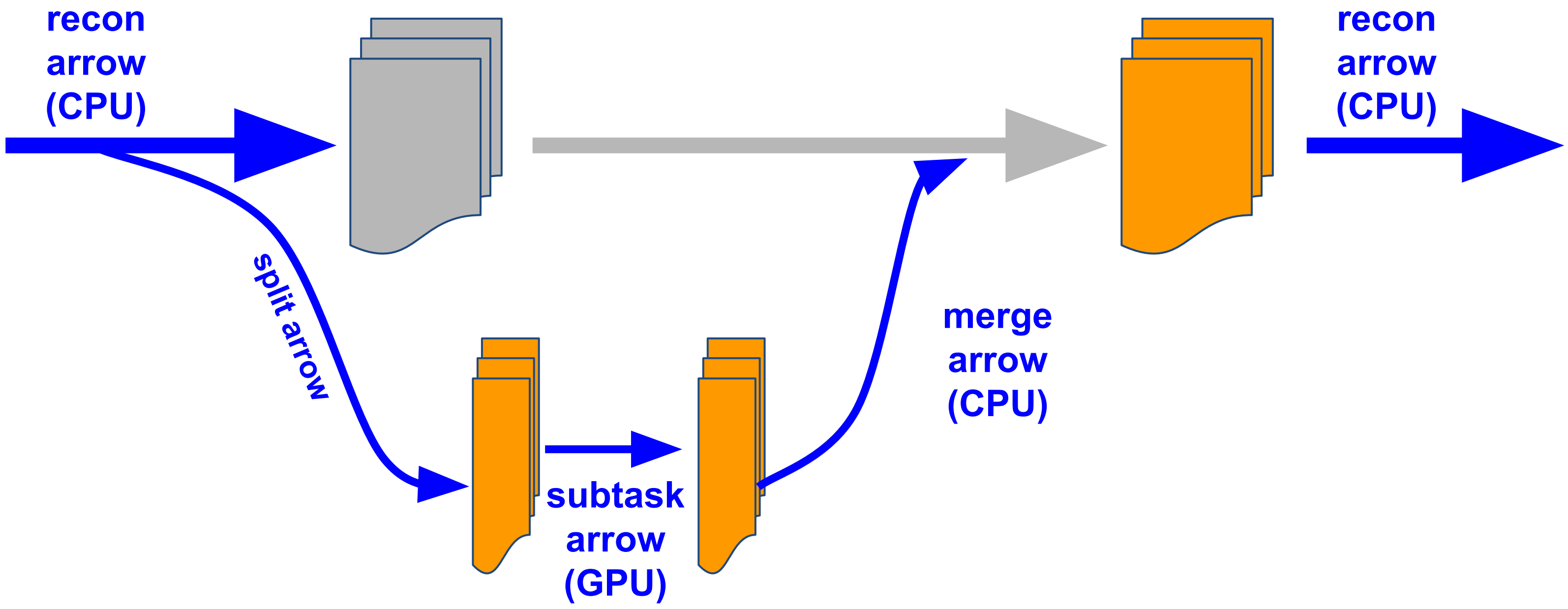
EXPERIMENTAL PHYSICS SOFTWARE AND COMPUTING INFRASTRUCTURE

```
> jana-generate.py
Usage: jana-generate.py [-h|--help] [type] [args...]
  type: JObject JEventSource JEventProcessor RootEventProcessor JEventProcessorTest JFactory Plugin Project
```

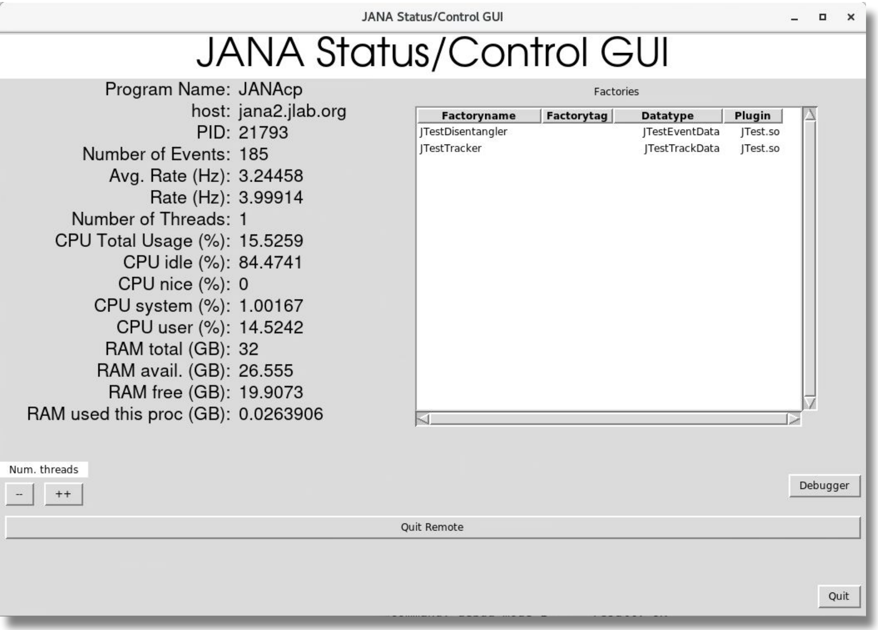
```
> jana-generate.py --help
…
Plugin
Create a code skeleton for a plugin in its own directory. Takes the following positional arguments:
          name              The name of the plugin, e.g. "trk_eff" or "TrackingEfficiency"
         [is_standalone]   Is this a new project, or are we inside the source tree of an existing CMake project? (default=True)
         [is_mini]          Reduce boilerplate and put everything in a single file? (default=True)
         [include_root]    Include a ROOT dependency and stubs for filling a ROOT histogram? (default=True)

       Example: `jana_generate.py Plugin TrackingEfficiency 1 0 0`
…
```

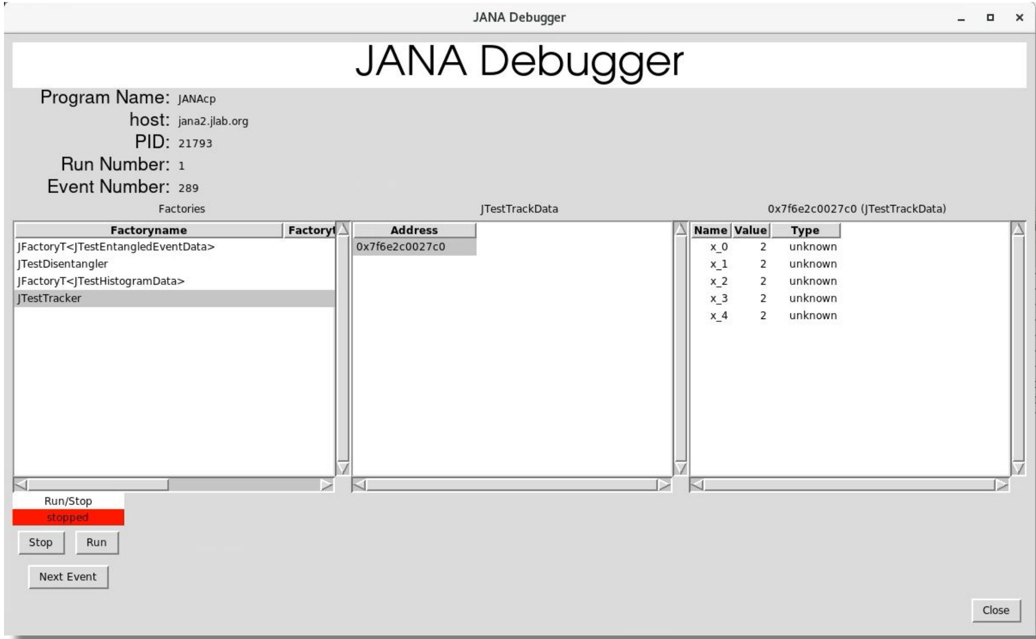
```
> jana-generate.py Plugin DaveTest
> ls DaveTest/
CMakeLists.txt  DaveTest.cc
> mkdir DaveTest/build
> cd DaveTest/build/
> cmake ..
…
> make install
[ 50%] Building CXX object CMakeFiles/DaveTest_plugin.dir/DaveTest.cc.o
[100%] Linking CXX shared library DaveTest.so
[100%] Built target DaveTest_plugin
Install the project...
-- Install configuration: ""
-- Installing: /Users/davidl/builds/JANA2/JANA2/plugins/DaveTest.so
```

Heterogeneous Hardware Support



Inspection Tools






```
> jana -Pplugins=JTest,janacontrol
```

Add *janacontrol* plugin to any process

```
> jana-control.py [--host host] [--port port]
```

Run GUI from remote (or same) node

JANA Command Line Debugging w/ gdb

EPSCI
Experimental Physics Software and Computing Infrastructure

```
davidl@jana2:JANA
File  Edit  View  Search  Terminal  Help
Class name:    JTestParser
Sequential:    0

JANA: [INFO] Status: 0 events processed  0.0 Hz (0.0 Hz avg)


JANA: h
  ----------------------------------------
  Available commands
  ----------------------------------------
  pe    PrintEvent
  pf    PrintFactories [filter_level <- {0,1,2,3}]
  pfd   PrintFactoryDetails fac_idx
  po    PrintObjects fac_idx
  po    PrintObject fac_idx obj_idx
  pfp   PrintFactoryParents fac_idx
  pop   PrintObjectParents fac_idx obj_idx
  poa   PrintObjectAncestors fac_idx obj_idx
  vt    ViewAsTable
  vj    ViewAsJson
  x     Exit
  h     Help
  ----------------------------------------

JANA: p
```

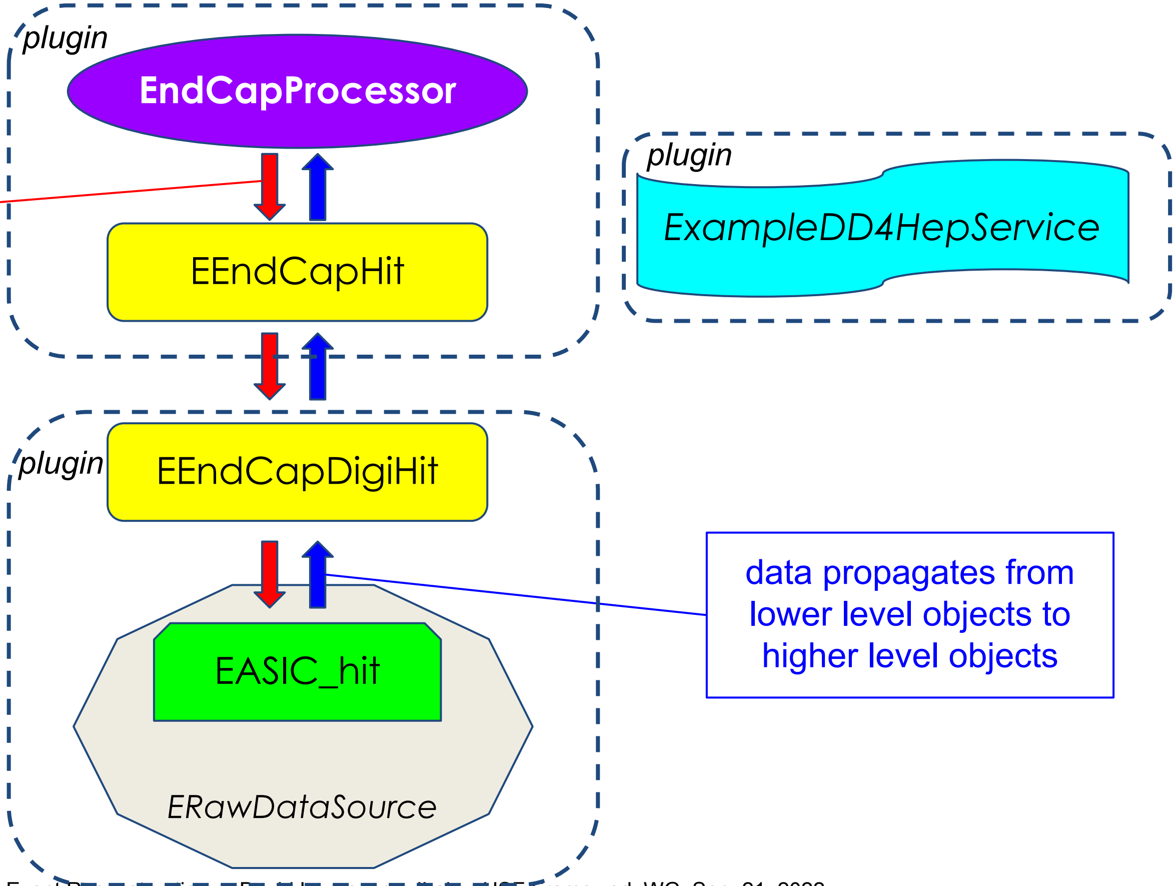
Certain JANA methods are written with the intention of being called from debugger.

This allows easier browsing from the framework point of of view.

Example with Geometry Service

https://github.com/faustus123/EIC_JANA_Example

requests start with higher level objects and propagate to lower level objects



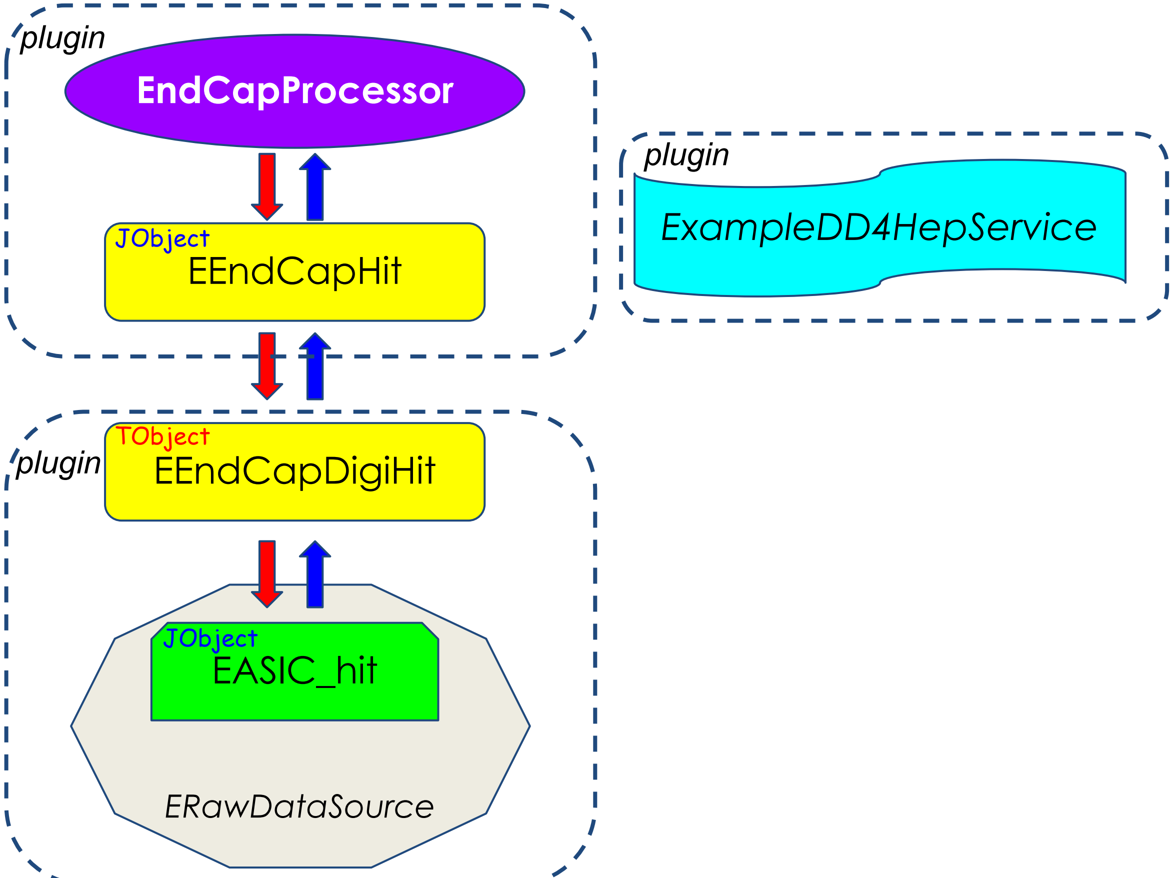
data propagates from lower level objects to higher level objects

Wouter suggested example:

"…[take] a collection of hits and selecting those hits that are on a particular endcap tracking detector and have a position outside a minimum radial range."

Example with Mixed TObject and JObject

https://github.com/faustus123/EIC_JANA_Example/tree/TObject_example



JFactory_EEndCapHit

```
#ifndef _JFactory_EEndCapHit_h_
#define _JFactory_EEndCapHit_h_

#include <JANA/JFactoryT.h>
#include <ExampleDD4HepService/ExampleDD4HepService.h>
#include "EEndCapHit.h"

class JFactory_EEndCapHit : public JFactoryT<EEndCapHit> {

    // Insert any member variables here

public:
    JFactory_EEndCapHit();
    void Init() override;
    void ChangeRun(const std::shared_ptr<const JEvent> &event) override;
    void Process(const std::shared_ptr<const JEvent> &event) override;

protected:
    double min_radius;

    const ExampleDD4HepService *geomservice=nullptr;

};

#endif // _JFactory_EEndCapHit_h_
```

```
void JFactory_EEndCapHit::Init() {
    auto app = GetApplication();

    // Just for fun, create a configuration parameter named
    // EndCap:min_radius so we can set the threshold at run time.
    min_radius = 15.0;
    app->SetDefaultParameter("EndCap:min_radius", min_radius, "The mini

    /// Acquire geometry service pointer (see ExampleDD4HepService plugin)
    geomservice = app->GetService<ExampleDD4HepService>().get();
}
```

boilerplate

added for this example

JFactory_EEndCapHit::Process

```
//-------------------------
// Process
//-------------------------
void JFactory_EEndCapHit::Process(const std::shared_ptr<const JEvent> &event) {

    /// JFactories are local to a thread, so we are free to access and modify
    /// member variables here. However, be aware that events are _scattered_ to
    /// different JFactory instances, not _broadcast_: this means that JFactory
    /// instances only see _some_ of the events.

    // The EEndCapDigiHit objects are made by a factory in the EICRawData plugin.
    // That factory uses the low-level EASIC_hit objects coming from the event source
    auto endcapdigihits = event->Get<EEndCapDigiHit>();

    // Loop over the EEndCapDigiHit objects and create calibrated hits
    // objects with geometry info.
    std::vector<EEndCapHit *> hits;
    for( auto digihit : endcapdigihits ){

        auto pos = geomservice->GetVTXPixelLocation( digihit->layer, digihit->chip, digihit->pixel );
        auto r = pos.Perp();
        if( r > min_radius ){

            auto hit = new EEndCapHit();
            hit->x = pos.X();
            hit->y = pos.Y();
            hit->z = pos.Z();
            hit->t = ((double)digihit->t - 125.0)*2.50E-1;  // Here we would apply calibrations read from DB
            hits.push_back(hit);
        }
    }

    /// Publish outputs
    Set(hits);

    // n.b. if we created additional types of objects we could also add them to the event using event->Insert() )
}
```

ExampleDD4HepService

EPSCI
Experimental Physics Software and Computing Infrastructure

```
class ExampleDD4HepService: public JService {

    // The geometry service needs to be sensitive to the exact data being processed since subtle
    // alignment changes or even significant changes to the detector could appear between one
    // data set and the next. The most versatile system would allow data from multiple different
    // geometry definitions to exist at the same time.
    //
    // For this to return the correct geometry, it needs information from the data stream itself
    // on when it was acquired so it can access the correct DB. I do not try and add that
    // complication here right now. I do demonstrate though that the JEvent reference would be
    // passed in so that the needed info can be extracted. Note that this should not be called for
    // every event, but rather from the ChangeRun method of a factory or processor indicating a
    // new calibration region of the stream has been reached.
    const dd4hep::Assembly* GetDD4hepAssembly(const std::shared_ptr<const JEvent> &event) const {

        // Retrieve the correct Assembly based on when the given
        // JEvent was acquired.

        return _assembly;
    }

    // There is a lot of freedom in how this class could be organized. One is to simply provide a
    // reference to the DD4hep Assembly object as above and let all of the algorithms speak "DD4hep".
    // A more practical approach would be to augment that with some dedicated methods that answer
    // common questions about the geometry for specific detectors. Here is an example of this:
    TVector3 GetVTXPixelLocation( int layer, int chip, int pixel ) const {

        // This is where the code to extract the location information given the layer,chip, and pixel
        // values would reside. This could either be directly from the dd4hep reference or from some
        // cached value.
        assert( layer>=1 && layer<=9 );

        double x = (double)chip*2.7;  // Totally unrealistic. Just for demo
        double y = (double)pixel*1.2; // Totally unrealistic. Just for demo
        double z = z_layer[layer-1];  // Lookup table (this should actually be close to correct!)

        return TVector3( x, y, z);
    }

private:
    dd4hep::Assembly *_assembly = nullptr;
    double z_layer[9] = {-106.0, -79.0, -52.0, -25.0, 25.0, 49.0, 73.0, 106.0, 125.0};

};
```

Service is added to application with single line:

```
extern "C" {
    void InitPlugin(JApplication *app) {
        InitJANAPlugin(app);
        app->ProvideService( std::make_shared<ExampleDD4HepService>() );
    }
}
```

The following are some notes I made a while back when trying to understand how JANA, Gaudi, and Fun4all approach the basic function of the framework. It is terribly incomplete, but may give some insight so I included it here in the backup slides.

Here I try and breakdown some example reconstruction code from ATHENA's juggler framework based on GAUDI. At the same time I try and compare this to what an equivalent JANA2 implementation would look like.

This is the first algorithm I looked at in the ATHENA repository and can be found here:

https://eicweb.phy.anl.gov/EIC/juggler/-/blob/master/JugReco/src/components/SimpleClustering.cpp

I looked at it first since the name "SimpleClustering" seemed like a good place to start.

SimpleClustering.cpp 6.21 KB

```cpp
#include <algorithm>

#include "Gaudi/Property.h"
#include "GaudiAlg/GaudiAlgorithm.h"
#include "GaudiAlg/GaudiTool.h"
#include "GaudiAlg/Transformer.h"
#include "GaudiKernel/PhysicalConstants.h"
#include "GaudiKernel/RndmGenerators.h"
#include "GaudiKernel/ToolHandle.h"

#include "DDRec/CellIDPositionConverter.h"
#include "DDRec/Surface.h"
#include "DDRec/SurfaceManager.h"

#include "JugBase/DataHandle.h"
#include "JugBase/IGeoSvc.h"
#include "JugBase/UniqueID.h"

// Event Model related classes
#include "dd4pod/CalorimeterHitCollection.h"
#include "eicd/CalorimeterHitCollection.h"
#include "eicd/ClusterCollection.h"
#include "eicd/ProtoClusterCollection.h"
#include "eicd/RawCalorimeterHitCollection.h"

using namespace Gaudi::Units;

namespace Jug::Reco {
```

This is a preamble to the file. Nothing remarkable here.

```
/** Simple clustering algorithm.
 *
 * \ingroup reco
 */
class SimpleClustering : public GaudiAlgorithm, AlgorithmIDMixin<> {
public:
```

Class is defined in implementation file in a Java-like way. This may be a stylistic choice, but definitely something allowed by GAUDI. Without a header file, the class cannot be directly used in code outside of this. Any use would have to come from properties of the class coming through one of its base classes.

```
DECLARE_COMPONENT(SimpleClustering)

} // namespace Jug::Reco
```

The class is declared to GAUDI by the DECLARE_COMPONENT *call at the bottom of the file. This is defined through a few files but eventually gets to this file and the following line:*

Gaudi/GaudiPluginService/include/Gaudi/PluginServiceV2.h

Registry::instance().add(id, { libraryName(), std::move(f), std::move(props) });

At this point I don't know if that is instantiating an object of this class or otherwise generating code that can be used to instantiate SimpleClustering objects later.

```
class SimpleClustering : public JFactoryT<Cluster> {
```

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

The JANA equivalent here would be to create a class inheriting from JFactory and then report that to JANA by instantiating a JFactoryGenerator class via template.

JANA will use the JFactoryGenerator class to instantiate multiple SimpleClustering objects later.

```cpp
/** Simple clustering algorithm.
 *
 * \ingroup reco
 */
class SimpleClustering : public GaudiAlgorithm, AlgorithmIDMixin<> {
public:
  using RecHits  = eic::CalorimeterHitCollection;
  using ProtoClusters = eic::ProtoClusterCollection;
  using Clusters = eic::ClusterCollection;

  DataHandle<RecHits>       m_inputHitCollection{"inputHitCollection", Gaudi::DataHandle::Reader, this};
  DataHandle<ProtoClusters> m_outputProtoClusters{"outputProtoCluster", Gaudi::DataHandle::Writer, this};
  DataHandle<Clusters>      m_outputClusters{"outputClusterCollection", Gaudi::DataHandle::Writer, this};

  Gaudi::Property<std::string> m_mcHits{this, "mcHits", ""};

  Gaudi::Property<double>   m_minModuleEdep{this, "minModuleEdep", 5.0 * MeV};
  Gaudi::Property<double>   m_maxDistance{this, "maxDistance", 20.0 * cm};

  /// Pointer to the geometry service
  SmartIF<IGeoSvc> m_geoSvc;

  // Monte Carlo particle source identifier
  const int32_t m_kMonteCarloSource{uniqueID<int32_t>("mcparticles")};
  // Optional handle to MC hits
  std::unique_ptr<DataHandle<dd4pod::CalorimeterHitCollection>> m_inputMC;

  SimpleClustering(const std::string& name, ISvcLocator* svcLoc)
    : GaudiAlgorithm(name, svcLoc)
    , AlgorithmIDMixin<>(name, info()) {
    declareProperty("inputHitCollection", m_inputHitCollection, "");
    declareProperty("outputProtoClusterCollection", m_outputClusters, "Output proto clusters");
    declareProperty("outputClusterCollection", m_outputClusters, "Output clusters");
  }
```

Convenience declarations

Data objects in Gaudi are contained in DataHandle templated classes. It looks like these wrappers are instantiated with a pointer to the algorithm object they belong to.

Gaudi Property objects look to similarly wrap variables in a class and register it with the Gaudi system. This will allow Gaudi to know and set these values externally.

The JANA equivalent to these properties are configuration parameters. It is not clear if Gaudi expects to change these after event processing has started, but in JANA they are not expected to change. A comparable JANA call would be:

*double m_minModuleEdep = 5.0 * MeV;*
app->SetDefaultParameter("minModuleEdep", m_minModuleEdep, "...");

typo?

Input and output objects are declared explicitly in the constructor. It is not clear why this is needed in addition to the DataHandle constructors above.

```cpp
  StatusCode initialize() override
  {
    if (GaudiAlgorithm::initialize().isFailure()) {
      return StatusCode::FAILURE;
    }
    // Initialize the MC input hit collection if requested
    if (m_mcHits != "") {
      m_inputMC =
          std::make_unique<DataHandle<dd4pod::CalorimeterHitCollection>>(m_mcHits, Gaudi::DataHandle::Reader, this);
    }
    m_geoSvc = service("GeoSvc");
    if (!m_geoSvc) {
      error() << "Unable to locate Geometry Service. "
              << "Make sure you have GeoSvc and SimSvc in the right order in the configuration." << endmsg;
      return StatusCode::FAILURE;
    }
    return StatusCode::SUCCESS;
  }
```

Gaudi initialization method. This returns a value indicating if the initialization succeeds or fails.

Here, a string property of the class is used to determine if an input container should be made for MC hits.

```cpp
void SimpleClustering::Init() {
    auto app = GetApplication();

    /// Acquire any parameters
    // app->GetParameter("parameter_name", m_destination);

    /// Acquire any services
    // m_service = app->GetService<ServiceT>();

    /// Set any factory flags
    // SetFactoryFlag(JFactory_Flags_t::NOT_OBJECT_OWNER);
}
```

JANA initialization method. Unlike Gaudi, JANA does not emit a return value. In JANA, Init() is only called at event processing time if/when an algorithm is first used and so it is assumed to be required. Fatal errors in the Init() method are expected to emit errors to the logging service and to tell the application to quit via a call to app->Quit(). One may also explicitly set an exit code with app->SetExitCode(val).

```cpp
    StatusCode execute() override
    {
      // input collections
      const auto& hits = *m_inputHitCollection.get();
      // Create output collections
      auto& proto = *m_outputProtoClusters.createAndPut();
      auto& clusters = *m_outputClusters.createAndPut();
      // Optional MC data
      const dd4pod::CalorimeterHitCollection* mcHits = nullptr;
      if (m_inputMC) {
        mcHits = m_inputMC->get();
      }

      std::vector<std::pair<uint32_t, eic::ConstCalorimeterHit>> the_hits;
      std::vector<std::pair<uint32_t, eic::ConstCalorimeterHit>> remaining_hits;
```

This is the top of the execute() method which is called for every event for which the algorithm is active. The first lines are used to get the inputs for the algorithm and to create the output containers for the algorithm.

This mechanism uses the existence of a container that may or may not have been created in the init() method to determine whether to get the actual hits into the container.

JANA method that is called for every event.

```cpp
void SimpleCluster_factory::Process(const std::shared_ptr<const JEvent> &jevent){

    auto calohits = jevent->Get<DFCALHit>(); // Get input objects

    // .... Create cluster objects ....
    {
        auto cluster = new DFCALCluster( a, b, c );
        for( auto hit : myhits )cluster->AddAssociatedObject( hit );
        Insert( cluster ); //pass ownership to framework
    }
}
```

Input objects obtained as vector<const DFCALHit> calohits*

Algorithm creates cluster objects and “Inserts” them into the event using the Insert() method. One could also fill a local std::vector<> of pointers and publish those with the Set() method.

If the DFCALCluster class inherits from JObject, then the AssociatedObject mechanism can be used. This allows the framework to know about which hit objects were used to make the cluster.

Here is a comparison with Fun4All. This is taken from the following:

https://github.com/ECCE-EIC/coresoftware/blob/master/offline/packages/CaloReco/RawClusterBuilderFwd.h

I wanted to use another calorimeter clustering algorithm and this was the best I could locating with a quick search.

To start with, I should note that some of the code dealing with this is spread over a few classes:

RawClusterDefs ← *Namespace. Defines RawClusterDefs::keytype*
RawCluster ← *Inherits from PHObject*
RawClusterContainer ← *Inherits from PHObject*
RawClusterBuilderFwd ← *Inherits from SubsysReco*

```
#ifndef CALOBASE_RAWCLUSTERDEFS_H
#define CALOBASE_RAWCLUSTERDEFS_H

namespace RawClusterDefs
{
  typedef unsigned int keytype;
}

#endif
```

This is just a namespace used to define the keytype used for the RawCluster objects. Presumably this is useful for object persistence since the unique id can be reproduced if the data were replayed.

JANA has removed support for object ids in JANA2. This is due to almost never being used in JANA1. This is likely due to the heavy use of pointers which also provide unique ids within the event, but don't require lookup tables to get at the object data.

```
class RawClusterContainer : public PHObject
{
 public:
  typedef std::map<RawClusterDefs::keytype, RawCluster *> Map;
  typedef Map::iterator Iterator;
  typedef Map::const_iterator ConstIterator;
  typedef std::pair<Iterator, Iterator> Range;
  typedef std::pair<ConstIterator, ConstIterator> ConstRange;

  RawClusterContainer() {}
  ~RawClusterContainer() override {}

  void Reset() override;
  int isValid() const override;
  void identify(std::ostream &os = std::cout) const override;

  ConstIterator AddCluster(RawCluster *clus);

  RawCluster *getCluster(const RawClusterDefs::keytype id);
  const RawCluster *getCluster(const RawClusterDefs::keytype id) const;

  //! return all clusters
  ConstRange getClusters(void) const;
  Range getClusters(void);
  const Map &getClustersMap() const { return _clusters; }
  Map &getClustersMap() { return _clusters; }

  unsigned int size() const { return _clusters.size(); }
  double getTotalEdep() const;

 protected:
  Map _clusters;
```

The RawClusterContainer class is interesting because it really serves as a customized container class for RawCluster objects. It has several methods like AddCluster, getCluster, getClusters, … that include the word “cluster” in their names. These do not seem to be doing anything special that any other container class would not already be doing. It is unclear why a more general (templated) container class is not used which could provide more uniformity in the code.

n.b. getTotalEdep() looks to be the only method that has functionality that would not be provided by a generic container class.

In JANA, the JFactory (i.e. algorithm) class that produces the data objects owns them and serves the combined purpose of the RawClusterContainer and RawClusterBuilderFwd classes. The JFactory class is actually a template itself where the template parameter is the specific type of primary data object the factory produces.
n.b. More than one object type can be produced by a JFactory. The supplementary types would use Insert() to add them to the event and would no longer be owned by the factory. This would make no difference to the end user. The emphasis on having a factory produce a single, primary object type is meant to encourage modularity in the overall design by having more, smaller algorithms.