

# Hall A/C Analysis Software Update

Ole Hansen

Jefferson Lab

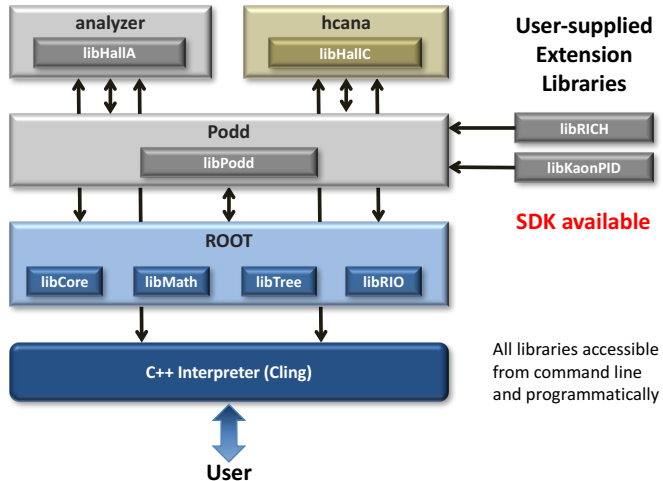
Hall A/C Summer Collaboration Meeting  
June 16, 2022

# Common Hall A/C Event Processing Framework: Podd

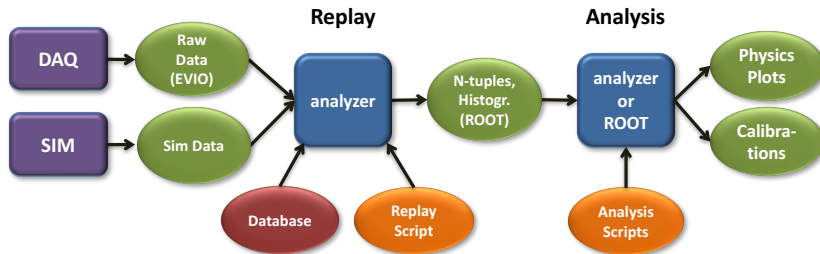
- C++ class library built on top of **ROOT**. Steering via ROOT interpreter.
- Developed in-house. Standard choice for Hall A reconstruction & analysis **since 2003**.
- Maintained on [GitHub](#). Documentation in [Redmine](#).
- **Shared development** with Hall C since 2012 ("[hcana](#)").
- Strengths
  - ▶ **Highly modular** to accommodate frequently changing experimental setups.
  - ▶ Intuitively conceptualizes analysis in terms of physical apparatuses (spectrometers, detectors) and physics calculations (kinematics, energy loss corrections, etc.)
  - ▶ **Light-weight**: minimal dependencies, small memory footprint.
  - ▶ Output & cuts **run-time configurable** via **text files**. Flat text file database.
- Limitations
  - ▶ Currently still single-threaded.
  - ▶ Designed for one-pass analysis: EVIO raw data → n-tuple-like ROOT trees + histograms
- Technical
  - ▶ Requirements: Linux or macOS. C++11. ROOT 6.
  - ▶ Recently modernized for C++11. Supports C++17 if available.

# Podd/hcana: Modular Architecture

- User interface: ROOT prompt (C++ interpreter)
- All loaded libraries (ROOT, Podd, etc.) accessible from command prompt for scripting
- Extension libraries for experiment-specific code can be loaded dynamically
  - ▶ Software Development Kit ([SDK](#)) to get started
- Entire **SBS software** package implemented as such an extension



# Reconstruction & Analysis Workflow



## 1 Reconstruction (Replay)

- ▶ Runs in ROOT interpreter (analyzer prompt)
- ▶ Calls mostly **Podd functions & classes**
- ▶ Scripts **set up by experiment experts** or advanced users
- ▶ After setup, runs in mass replay on the farm

## 2 Analysis

- ▶ Also runs in ROOT interpreter (analyzer prompt)
- ▶ Calls mostly **ROOT functions and classes** (but may need Podd classes)
- ▶ Done by **everyone** on the experiment
- ▶ **Calibration** and **final physics** usually done here

# Podd Source Code & Documentation

## GitHub

The screenshot shows the GitHub repository page for `JeffersonLab/analyzer`. The repository is public and has 44 forks and 6 stars. The main branch is `master`. The repository contains a list of files and folders, including `DB`, `Database`, `HallA`, `Podd`, `SDK`, `apps`, `cmake`, `docs`, and `examples`. The `Releases` section is highlighted with a red box, showing the latest release, `1.7.0`, dated 15 Dec 2021. The `packages` section shows that no packages have been published.

## JLab Redmine

The screenshot shows the JLab Redmine Wiki page for `Hall A Analyzer`. The page is titled `Hall A Analyzer` and has a search bar. The `Wiki` tab is selected. The `Introduction` section describes the software as the main Hall A physics analysis software, "Podd". The `Resources` section lists links to `Documentation`, `Release Notes`, `Git Repository`, and `Class Index`. The `Downloads` section lists the most recent source code, including `Analyzer 1.7.0 source code (production version)` dated 15 Nov 2021. The `Downloads` section is highlighted with a red box.

# Pre-Installed Podd

## farm/ifarm (works in Counting House, too)

```
$ module use /group/halla/modulefiles
$ module load analyzer
$ analyzer --version
Podd 1.7.0 Linux-3.10.0-1160.31.1.el7.x86_64-x86_64 git @e26c21d ROOT 6.22/06
```

## Counting House (local installation, faster, safer)

```
$ module use /adaqfs/apps/modulefiles
$ module load analyzer
$ analyzer --version
Podd 1.7.0 Linux-3.10.0-1160.31.1.el7.x86_64-x86_64 git @e26c21d ROOT 6.24/06
```

The SDK is located in `$ANALYZER/./src/SDK/`

# Podd Status & Roadmap

- Current release: **1.7.0** (16 Nov 2021)
  - ▶ Base software for SBS experiments and current Hall C `hcanal`.
  - ▶ Source-level backwards compatible (mostly), *i.e.* suitable for replaying older data as well.
  - ▶ Many speed improvements, CODA 3 support, etc. (see release notes)
  - ▶ Requires **C++11** compiler and ROOT 6. Installed in counting house and on the farm.
- Upcoming: **1.7.1** ("[real soon now](#)")
  - ▶ New "**Run**" **classes** supporting transparent input from multiple run segments and event streams
  - ▶ More info in output, e.g. event number in EPICS tree
- The Next Generation: **2.0** (Fall 2022)
  - ▶ **Multithreading!**
  - ▶ Will benefit SBS and Hall C, primarily for online replay
  - ▶ Requires **C++17** (e.g. gcc 9+, available on ifarm)
  - ▶ Existing code will need minor modifications

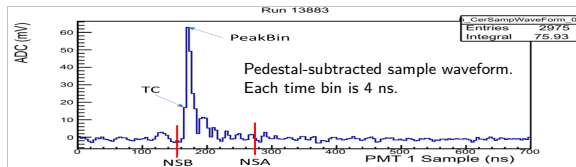
# Hall C Software Updates (hcana) (from Mark Jones)

- Feb 2022: Added **alternative drift-chamber tracking code**. Improves clustering and fitting for large- $\chi^2$  tracks ([slides](#)). Old algorithm still available, selectable through database parameter.
- Support for **updated FADC250 firmware** ([details](#)). Thanks to David Hamilton.
  - ▶ Old firmware: Incorrect pedestal if signal within first 4 samples. Approximately corrected with a workaround in `hcana`.
  - ▶ New firmware: FADC provides actual waveform samples. Waveform then analyzed in `hcana`.
  - ▶ Cherenkov readouts now also provide waveform data. Analyze pulses using waveform and store pulse data in `hcana`.



## Hall C Software Updates (cont.) (from Mark Jones)

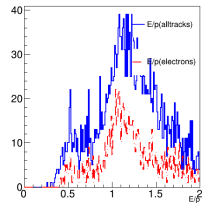
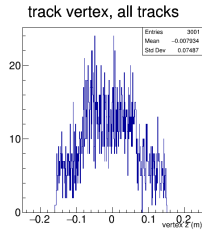
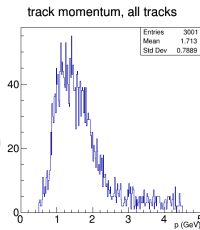
- Details on improved FADC250 waveform analysis:
  - ▶ Each detector's Decode() method now supports sample waveform data analysis. Mimics firmware algorithm.
  - ▶ Parameters (NSA, NSB, threshold, NSAT) settable per detector. Defaults from FADC250 config. info event.
  - ▶ Sliding-window search algorithm for peaks. Extracts peak amplitude (ADC(PeakBin)), integral (between NSB–NSA), and pulse time (TC), or best guesses if none found.



- New tree variables added to store waveform analysis results.

# SBS Software

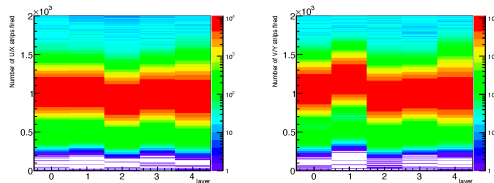
- Standalone [g4sbs](#) simulation package (Geant4-based). Well developed and tested. Typically run on JLab batch farm.
- Reconstruction routines implemented in a Podd library: [SBS-offline](#)
  - ▶ **GEM track reconstruction** for BigBite
  - ▶ BigBite shower/preshower cluster finding & PID
  - ▶ Timing hodoscope analysis
  - ▶ GRINCH Cherenkov PID
  - ▶ Hadron calorimeter (HCAL) cluster finding
  - ▶ Decoders for SBS-specific modules (MPD, VETROC)
- Calibration & analysis
  - ▶ First-order BigBite optics model
  - ▶ Time-of-flight and PID
  - ▶ GEM tracker alignment
- PID: Currently using Podd's standard Bayesian likelihood calculation
- GEM tracking performance
  - ▶ Typ. 10–20 Hz analysis rate
  - ▶  $\geq 70\%$  online efficiency with real data



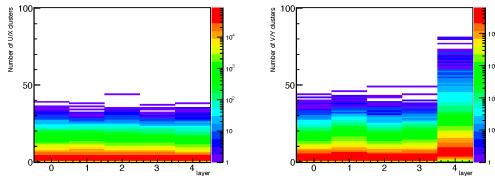
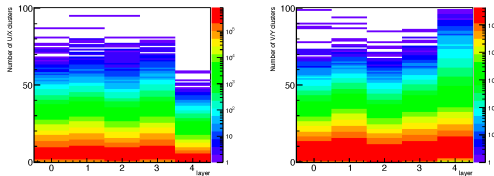
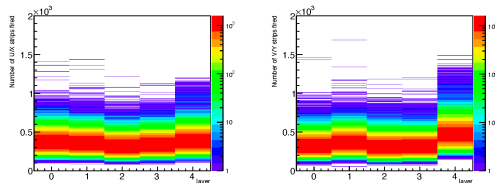
# SBS GEM Analysis — Recent Progress (from Andrew Puckett)

## Recent Highlights—GEM occupancy reduction with improved baseline calculation/subtraction

Summary Plots(Run #13452) 0: Strip and cluster multiplicities



Summary Plots(Run #13452) 1: Strip and cluster multiplicities



- Significant negative bias of baseline calculation used for online zero suppression during GMN led to much noise being recorded as “signal”—improved “histogramming method” calculation reduces bias, enables *post hoc* corrections (developed and tested by Sean Jeffas (UVA)) to the online calculated baseline for GMN analysis, even for (most) online zero-suppressed events for GMN analysis.
- **Preliminary results: 2-5X reduction of effective “raw” occupancy, speeds up tracking, improves efficiency, reduces fake tracks**
- **Implementation of “Histogramming method” for online baseline calculation could significantly reduce data rate/data volume for future SBS experiments**

# Open Software Tasks (most of this from Andrew Puckett)

- For SBS-GE<sub>n</sub>-II
  - ▶ SBS-arm GEM track reconstruction
  - ▶ HCAL constraint for charged-particle tracking in SBS
  - ▶ Coincidence timing analysis
  - ▶ SBS optics model and calibration (NB: no sieve slit)
  - ▶ Improve/consolidate online detector monitoring (too many plots)
- For SBS-GE<sub>n</sub>-RP
  - ▶ SBS polarimeter tracking and analysis.  $\vec{p}$  polarimetry.
- For SBS-GE<sub>p</sub>
  - ▶ Electron calorimeter (ECAL) and coordinate detector (CD) reconstruction
  - ▶ Elastic  $e$ - $p$  kinematic correlation analysis — may require new kind of “apparatus” class. (Required detector analysis sequence differs from standard Podd processing order.)
- Hall C NPS and LAD software (partly done)
- Multi-threading in Podd (benefits both halls). Better I/O and memory profile on farm. Proof-of-concept available.

## Podd 2.0

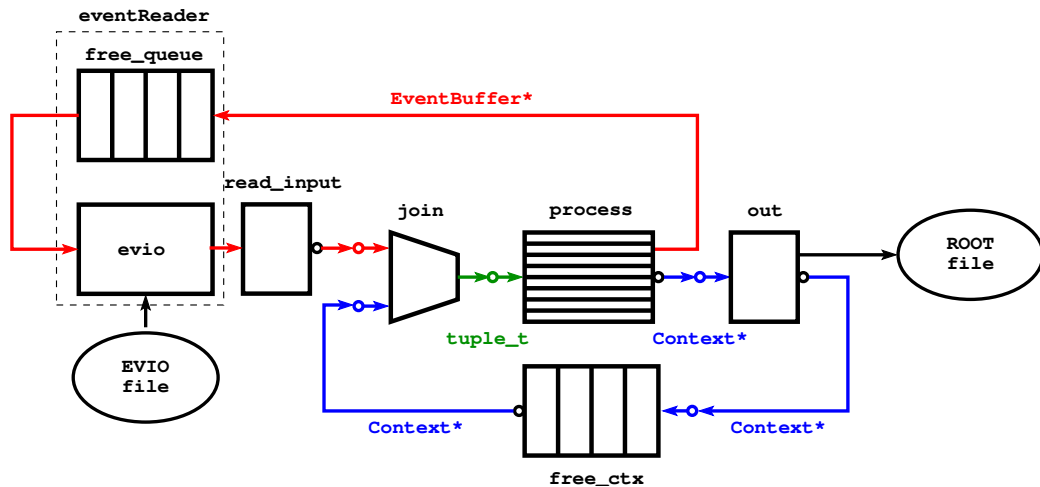
- Event-level **parallelization/multithreading**
  - ▶ Especially important for online replay
  - ▶ Reduced memory footprint compared to multiple individual jobs
  - ▶ Requires **thread safe** user code (→ only const or protected globals, statics)
- I/O improvements
  - ▶ Output system upgrade (full set of data types, object variables) — largely complete
  - ▶ ~~HIPO~~ or **PODIO** output file format support
  - ▶ **EVIO 6** input format support (HIPO-like raw data files) — once EVIO 6 stable
  - ▶ Goal: Make output easily usable with Python and Julia tools (e.g. **uproot**, **UnROOT**)

ETA: Aiming to have multithreading & output data typing ready for fall SBS run

# Parallel Podd Prototype

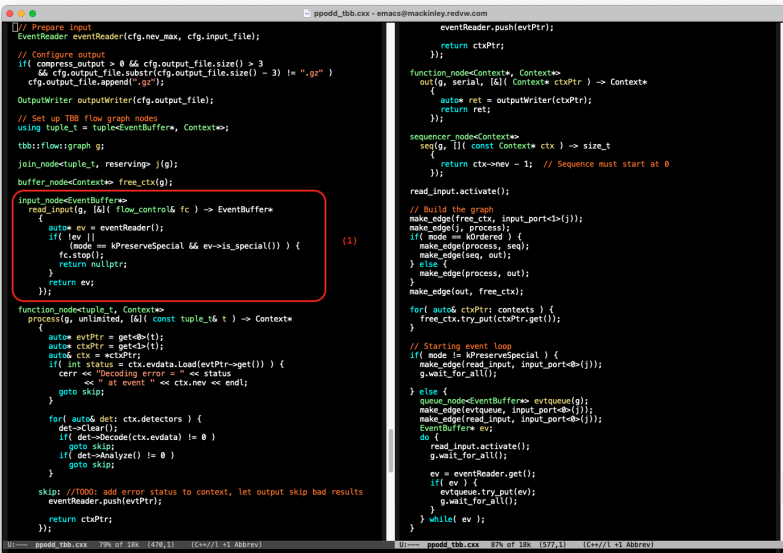
- <https://github.com/hansenjo/parallel>
- Mimics main components of Podd (e.g. decoder, analysis variables, output)
- A few example “detectors” included whose processing is intended to burn CPU cycles
- **Rewritten using oneAPI TBB** library (formerly Intel Thread Building Blocks)
- Three processing modes:
  - ① Unordered — event numbers may not be consecutive in output
  - ② Ordered — consecutive event numbers guaranteed
  - ③ Barriers — events guaranteed to stay between special barrier events (e.g. scalers)
- For now, **serialized output** (hard to avoid because of ROOT) — potential bottleneck

# TBB-based Parallel Podd — Unordered Mode Flow Graph



# Podd Parallel Processing Prototype

## 1 Read raw event (serial)



```
// Prepare input
EventReader eventReader(cfg.nev_max, cfg.input_file);

// Configure output
if( compress_output > 0 && cfg.output_file.size() > 3
    && cfg.output_file.substr(cfg.output_file.size() - 3) != ".gz" )
    cfg.output_file.append(".gz");

OutputWriter outputWriter(cfg.output_file);

// Set up TBB flow graph nodes
using tuple_t = tuple<EventBuffer*, Context*>;

tbb::flow::graph g;

join_node<tuple_t, reserving> j(g);

buffer_node<Context*> free_ctx(g);

input_node<EventBuffer*>
read_input(g, [&]( flow_control& fc ) -> EventBuffer*
{
    auto* ev = eventReader();
    if( !ev ||
        (mode == kPreserveSpecial && ev->is_special()) ) {
        fc.stop();
        return nullptr;
    }
    return ev;
});

function_node<tuple_t, Context*>
process(g, unlimited, [&]( const tuple_t& t ) -> Context*
{
    auto* evtPtr = get<0>(t);
    auto* ctxPtr = get<1>(t);
    auto& ctx = *ctxPtr;
    if( int status = ctx.evdata.Load(evtPtr->get()) ) {
        cerr << "Decoding error = " << status
            << " at event " << ctx.nev << endl;
        goto skip;
    }

    for( auto& det: ctx.detectors ) {
        det->Clear();
        if( det->Decode(ctx.evdata) != 0 )
            goto skip;
        if( det->Analyze() != 0 )
            goto skip;
    }

    skip: //TODO: add error status to context, let output skip bad results
    eventReader.push(evtPtr);

    return ctxPtr;
});

eventReader.push(evtPtr);
return ctxPtr;
});

function_node<Context*, Context*>
out(g, serial, [&]( Context* ctxPtr ) -> Context*
{
    auto* ret = outputWriter(ctxPtr);
    return ret;
});

sequencer_node<Context*>
seq(g, []( const Context* ctx ) -> size_t
{
    return ctx->nev - 1; // Sequence must start at 0
});

read_input.activate();

// Build the graph
make_edge(free_ctx, input_port<1>(j));
make_edge(j, process);
if( mode == kOrdered ) {
    make_edge(process, seq);
    make_edge(seq, out);
} else {
    make_edge(process, out);
}
make_edge(out, free_ctx);

for( auto& ctxPtr: contexts ) {
    free_ctx.try_put(ctxPtr.get());
}

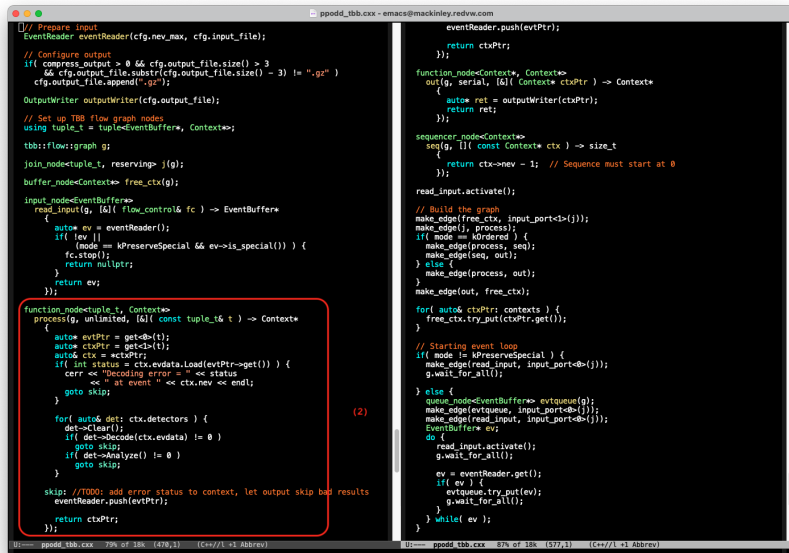
// Starting event loop
if( mode != kPreserveSpecial ) {
    make_edge(read_input, input_port<0>(j));
    g.wait_for_all();
} else {
    queue_node<EventBuffer*> evtqueue(g);
    make_edge(evtqueue, input_port<0>(j));
    make_edge(read_input, input_port<0>(j));
    EventBuffer* ev;
    do {
        read_input.activate();
        g.wait_for_all();

        ev = eventReader.get();
        if( ev ) {
            evtqueue.try_put(ev);
            g.wait_for_all();
        }
    } while( ev );
}
```



# Podd Parallel Processing Prototype

- 1 Read raw event (serial)
- 2 Parallel processing like Podd: RawDecode, Decode, Process



```
// Prepare input
EventReader eventReader(cfg.nev_max, cfg.input_file);

// Configure output
if( compress_output > 0 && cfg.output_file.size() > 3
    && cfg.output_file.substr(cfg.output_file.size() - 3) != ".gz" )
    cfg.output_file.append(".gz");

OutputWriter outputWriter(cfg.output_file);

// Set up TBB flow graph nodes
using tuple_t = tuple<EventBuffer*, Context*>;

tbb::flow::graph g;

join_node<tuple_t, reserving> j(g);

buffer_node<Context*> free_ctx(g);

input_node<EventBuffer*>
read_input(g, [&]( flow_control& fc ) -> EventBuffer*
{
    auto* ev = eventReader();
    if( !ev ||
        (mode == kPreserveSpecial && ev->is_special()) ) {
        fc.stop();
        return nullptr;
    }
    return ev;
});

function_node<tuple_t, Context*>
process(g, unlimited, [&]( const tuple_t& t ) -> Context*
{
    auto* evtPtr = get<0>(t);
    auto* ctxPtr = get<1>(t);
    auto& ctx = *ctxPtr;
    if( int status = ctx.evdata.Load(evtPtr->get()) ) {
        cerr << "Decoding error = " << status
            << " at event " << ctx.nev << endl;
        goto skip;
    }

    for( auto& det: ctx.detectors ) {
        det->Clear();
        if( det->Decode(ctx.evdata) != 0 )
            goto skip;
        if( det->Analyze() != 0 )
            goto skip;
    }

    skip: //TODO: add error status to context, let output skip bad results
    eventReader.push(evtPtr);

    return ctxPtr;
});

eventReader.push(evtPtr);
return ctxPtr;
});

function_node<Context*, Context*>
out(g, serial, [&]( Context* ctxPtr ) -> Context*
{
    auto* ret = outputWriter(ctxPtr);
    return ret;
});

sequencer_node<Context*>
seq(g, []( const Context* ctx ) -> size_t
{
    return ctx->nev - 1; // Sequence must start at 0
});

read_input.activate();

// Build the graph
make_edge(free_ctx, input_port<1>(j));
make_edge(j, process);
if( mode == kOrdered ) {
    make_edge(process, seq);
    make_edge(seq, out);
} else {
    make_edge(process, out);
}
make_edge(out, free_ctx);

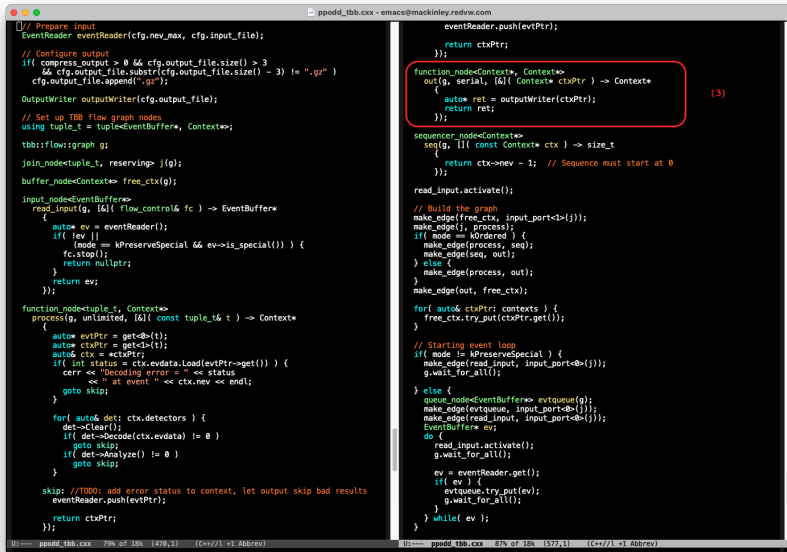
for( auto& ctxPtr: contexts ) {
    free_ctx.try_put(ctxPtr.get());
}

// Starting event loop
if( mode != kPreserveSpecial ) {
    make_edge(read_input, input_port<0>(j));
    g.wait_for_all();
} else {
    queue_node<EventBuffer*> evtqueue(g);
    make_edge(evtqueue, input_port<0>(j));
    make_edge(read_input, input_port<0>(j));
    EventBuffer* ev;
    do {
        read_input.activate();
        g.wait_for_all();

        ev = eventReader.get();
        if( ev ) {
            evtqueue.try_put(ev);
            g.wait_for_all();
        }
    } while( ev );
}
```

# Podd Parallel Processing Prototype

- 1 Read raw event (serial)
- 2 Parallel processing like Podd: RawDecode, Decode, Process
- 3 Output (serial)



```
// Prepare input
EventReader eventReader(cfg.nev_max, cfg.input_file);

// Configure output
if( compress_output > 0 && cfg.output_file.size() > 3
    && cfg.output_file.substr(cfg.output_file.size() - 3) != ".gz" )
    cfg.output_file.append(".gz");

OutputWriter outputWriter(cfg.output_file);

// Set up TBB flow graph nodes
using tuple_t = tuple<EventBuffer*, Context*>;

tbb::flow::graph g;

join_node<tuple_t, reserving> j(g);

buffer_node<Context*> free_ctx(g);

input_node<EventBuffer*>
read_input(g, [&]( flow_control& fc ) -> EventBuffer*
{
    auto* ev = eventReader();
    if( !ev ||
        (mode == kPreserveSpecial && ev->is_special()) ) {
        fc.stop();
        return nullptr;
    }
    return ev;
});

function_node<tuple_t, Context*>
process(g, unlimited, [&]( const tuple_t& t ) -> Context*
{
    auto* evtPtr = get<0>(t);
    auto* ctxPtr = get<1>(t);
    auto& ctx = *ctxPtr;
    if( int status = ctx.evdata.Load(evtPtr->get()) ) {
        cerr << "Decoding error = " << status
            << " at event " << ctx.nev << endl;
        goto skip;
    }

    for( auto& det: ctx.detectors ) {
        det->Clear();
        if( det->Decode(ctx.evdata) != 0 )
            goto skip;
        if( det->Analyze() != 0 )
            goto skip;
    }

    skip: //TODO: add error status to context, let output skip bad results
    eventReader.push(evtPtr);

    return ctxPtr;
});

eventReader.push(evtPtr);
return ctxPtr;
});

function_node<Context*, Context*>
out(g, serial, [&]( Context* ctxPtr ) -> Context*
{
    auto* ret = outputWriter(ctxPtr);
    return ret;
});

sequencer_node<Context*>
seq(g, []( const Context* ctx ) -> size_t
{
    return ctx->nev - 1; // Sequence must start at 0
});

read_input.activate();

// Build the graph
make_edge(free_ctx, input_port<1>(j));
make_edge(j, process);
if( mode == kOrdered ) {
    make_edge(process, seq);
    make_edge(seq, out);
} else {
    make_edge(process, out);
}
make_edge(out, free_ctx);

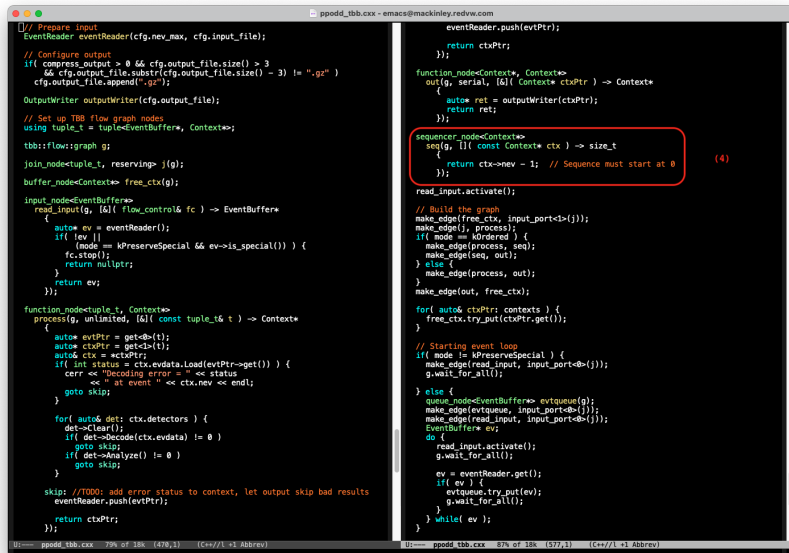
for( auto& ctxPtr: contexts ) {
    free_ctx.try_put(ctxPtr.get());
}

// Starting event loop
if( mode != kPreserveSpecial ) {
    make_edge(read_input, input_port<0>(j));
    g.wait_for_all();
} else {
    queue_node<EventBuffer*> evtqueue(g);
    make_edge(evtqueue, input_port<0>(j));
    make_edge(read_input, input_port<0>(j));
    EventBuffer* ev;
    do {
        read_input.activate();
        g.wait_for_all();

        ev = eventReader.get();
        if( ev ) {
            evtqueue.try_put(ev);
            g.wait_for_all();
        }
    } while( ev );
}
```

# Podd Parallel Processing Prototype

- 1 Read raw event (serial)
- 2 Parallel processing like Podd: RawDecode, Decode, Process
- 3 Output (serial)
- 4 Optional sequencer for event ordering



```
// Prepare input
EventReader eventReader(cfg.nev_max, cfg.input_file);

// Configure output
if( compress_output > 0 && cfg.output_file.size() > 3
    && cfg.output_file.substr(cfg.output_file.size() - 3) != ".gz" )
    cfg.output_file.append(".gz");

OutputWriter outputWriter(cfg.output_file);

// Set up TBB flow graph nodes
using tuple_t = tuple<EventBuffer*, Context*>;

tbb::flow::graph g;

join_node<tuple_t, reserving> j(g);

buffer_node<Context*> free_ctx(g);

input_node<EventBuffer*>
read_input(g, [&]( flow_control& fc ) -> EventBuffer*
{
    auto* ev = eventReader();
    if( !ev ||
        (mode == kPreserveSpecial && ev->is_special()) ) {
        fc.stop();
        return nullptr;
    }
    return ev;
});

function_node<tuple_t, Context*>
process(g, unlimited, [&]( const tuple_t& t ) -> Context*
{
    auto* evtPtr = get<0>(t);
    auto* ctxPtr = get<1>(t);
    auto& ctx = *ctxPtr;
    if( int status = ctx.evdata.Load(evtPtr->get()) ) {
        cerr << "Decoding error = " << status
            << " at event " << ctx.nev << endl;
        goto skip;
    }

    for( auto& det: ctx.detectors ) {
        det->Clear();
        if( det->Decode(ctx.evdata) != 0 )
            goto skip;
        if( det->Analyze() != 0 )
            goto skip;
    }

    skip: //TODO: add error status to context, let output skip bad results
    eventReader.push(evtPtr);

    return ctxPtr;
});

eventReader.push(evtPtr);
return ctxPtr;
});

function_node<Context*, Context*>
out(g, serial, [&]( Context* ctxPtr ) -> Context*
{
    auto* ret = outputWriter(ctxPtr);
    return ret;
});

sequencer_node<Context*>
seq(g, [&]( const Context* ctx ) -> size_t
{
    return ctx->nev - 1; // Sequence must start at 0
});

read_input.activate();

// Build the graph
make_edge(free_ctx, input_port<1>(j));
make_edge(j, process);
if( mode == kOrdered ) {
    make_edge(process, seq);
    make_edge(seq, out);
} else {
    make_edge(process, out);
}
make_edge(out, free_ctx);

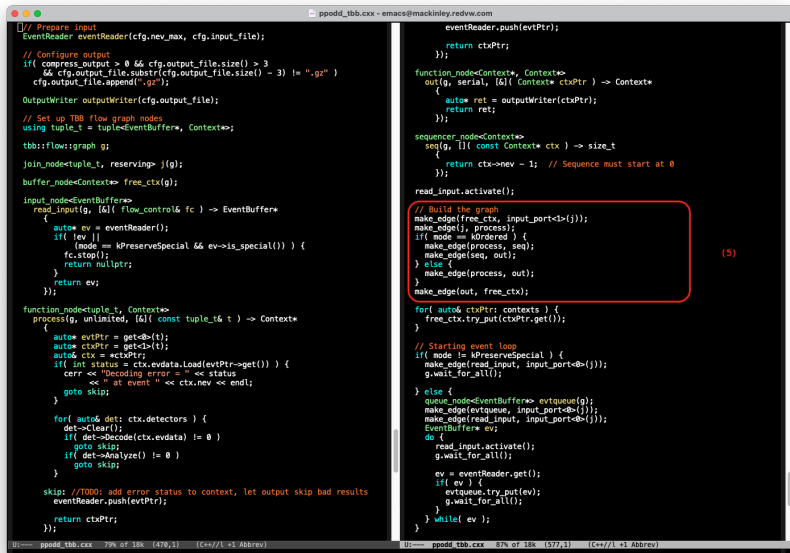
for( auto& ctxPtr: contexts ) {
    free_ctx.try_put(ctxPtr.get());
}

// Starting event loop
if( mode != kPreserveSpecial ) {
    make_edge(read_input, input_port<0>(j));
    g.wait_for_all();
} else {
    queue_node<EventBuffer*> evtqueue(g);
    make_edge(evtqueue, input_port<0>(j));
    make_edge(read_input, input_port<0>(j));
    EventBuffer* ev;
    do {
        read_input.activate();
        g.wait_for_all();

        ev = eventReader.get();
        if( ev ) {
            evtqueue.try_put(ev);
            g.wait_for_all();
        }
    } while( ev );
}
```

# Podd Parallel Processing Prototype

- 1 Read raw event (serial)
- 2 Parallel processing like Podd: RawDecode, Decode, Process
- 3 Output (serial)
- 4 Optional sequencer for event ordering
- 5 Build graph according to mode



```
// Prepare input
EventReader eventReader(cfg.nev_max, cfg.input_file);

// Configure output
if( compress_output > 0 && cfg.output_file.size() > 3
    && cfg.output_file.substr(cfg.output_file.size() - 3) != ".gz" )
    cfg.output_file.append(".gz");

OutputWriter outputWriter(cfg.output_file);

// Set up TBB flow graph nodes
using tuple_t = tuple<EventBuffer*, Context*>;

tbb::flow::graph g;

join_node<tuple_t, reservo> j(g);
buffer_node<Context*> free_ctx(g);

input_node<EventBuffer*>
read_input(g, [&]( flow_control& fc ) -> EventBuffer*
{
    auto* ev = eventReader();
    if( !ev ||
        (mode == kPreserveSpecial && ev->is_special()) ) {
        fc.stop();
        return nullptr;
    }
    return ev;
});

function_node<tuple_t, Context*>
process(g, unlimited, [&]( const tuple_t& t ) -> Context*
{
    auto* evtPtr = get->0(t);
    auto* ctxPtr = get->1(t);
    auto& ctx = *ctxPtr;
    if( int status = ctx.evdata.Load(evtPtr->get()) ) {
        cerr << "Decoding error = " << status
            << " at event " << ctx.nev << endl;
        goto skip;
    }

    for( auto& det: ctx.detectors ) {
        det->Clear();
        if( det->Decode(ctx.evdata) != 0 )
            goto skip;
        if( det->Analyze() != 0 )
            goto skip;
    }

    skip: //TODO: add error status to context, let output skip bad results
    eventReader.push(evtPtr);

    return ctxPtr;
});

eventReader.push(evtPtr);
return ctxPtr;
});

function_node<Context*, Context*>
out(g, serial, [&]( Context* ctxPtr ) -> Context*
{
    auto* ret = outputWriter(ctxPtr);
    return ret;
});

sequencer_node<Context*>
seq(g, []( const Context* ctx ) -> size_t
{
    return ctx->nev - 1; // Sequence must start at 0
});

read_input.activate();

// Build the graph
make_edge(free_ctx, input_port->{j});
make_edge(j, process);
if( mode == kOrdered ) {
    make_edge(process, seq);
    make_edge(seq, out);
} else {
    make_edge(process, out);
}
make_edge(out, free_ctx);

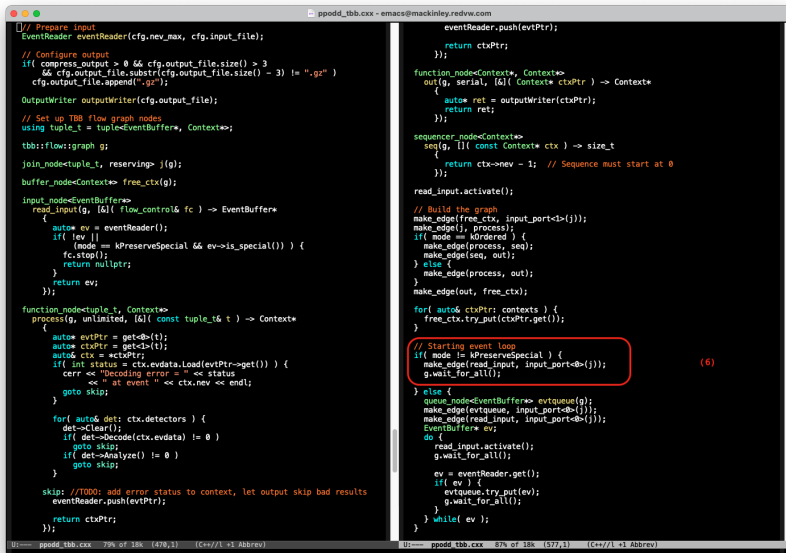
for( auto& ctxPtr: contexts ) {
    free_ctx.try_put(ctxPtr.get());
}

// Starting event loop
if( mode != kPreserveSpecial ) {
    make_edge(read_input, input_port->{j});
    g.wait_for_all();
} else {
    queue_node<EventBuffer*> evtqueue(g);
    make_edge(evtqueue, input_port->{j});
    make_edge(read_input, input_port->{j});
    EventBuffer* ev;
    do {
        read_input.activate();
        g.wait_for_all();

        ev = eventReader.get();
        if( ev ) {
            evtqueue.try_put(ev);
            g.wait_for_all();
        }
    } while( ev );
}
```

# Podd Parallel Processing Prototype

- 1 Read raw event (serial)
- 2 Parallel processing like Podd: RawDecode, Decode, Process
- 3 Output (serial)
- 4 Optional sequencer for event ordering
- 5 Build graph according to mode
- 6 Connect input and start processing



```
// Prepare input
EventReader eventReader(cfg.nev_max, cfg.input_file);

// Configure output
if (compress_output > 0 && cfg.output_file.size() > 3
    && cfg.output_file.substr(cfg.output_file.size() - 3) != ".gz" )
    cfg.output_file.append(".gz");

OutputWriter outputWriter(cfg.output_file);

// Set up TBB flow graph nodes
using tuple_t = tuple<EventBuffer*, Context*>;

tbb::flow::graph g;

join_node<tuple_t, reserving> j(g);

buffer_node<Context*> free_ctx(g);

input_node<EventBuffer*>
read_input(g, [&]( flow_control& fc ) -> EventBuffer*
{
    auto* ev = eventReader();
    if ( !ev ||
        (mode == kPreserveSpecial && ev->is_special()) ) {
        fc.stop();
        return nullptr;
    }
    return ev;
});

function_node<tuple_t, Context*>
process(g, unlimited, [&]( const tuple_t& t ) -> Context*
{
    auto* evtPtr = get<0>(t);
    auto* ctxPtr = get<1>(t);
    auto& ctx = *ctxPtr;
    if ( int status = ctx.evdata.Load(evtPtr->get()) ) {
        cerr << "Decoding error = " << status
            << " at event " << ctx.nev << endl;
        goto skip;
    }

    for( auto& det: ctx.detectors ) {
        det->Clear();
        if ( det->Decode(ctx.evdata) != 0 )
            goto skip;
        if ( det->Analyze() != 0 )
            goto skip;
    }

    skip: //TODO: add error status to context, let output skip bad results
    eventReader.push(evtPtr);

    return ctxPtr;
});

eventReader.push(evtPtr);
return ctxPtr;
});

function_node<Context*, Context*>
out(g, serial, [&]( Context* ctxPtr ) -> Context*
{
    auto* ret = outputWriter(ctxPtr);
    return ret;
});

sequencer_node<Context*>
seq(g, [( const Context* ctx ) -> size_t
{
    return ctx->nev - 1; // Sequence must start at 0
}]);

read_input.activate();

// Build the graph
make_edge(free_ctx, input_port<1>(j));
make_edge(j, process);
if ( mode == kOrdered ) {
    make_edge(process, seq);
    make_edge(seq, out);
} else {
    make_edge(process, out);
}
make_edge(out, free_ctx);

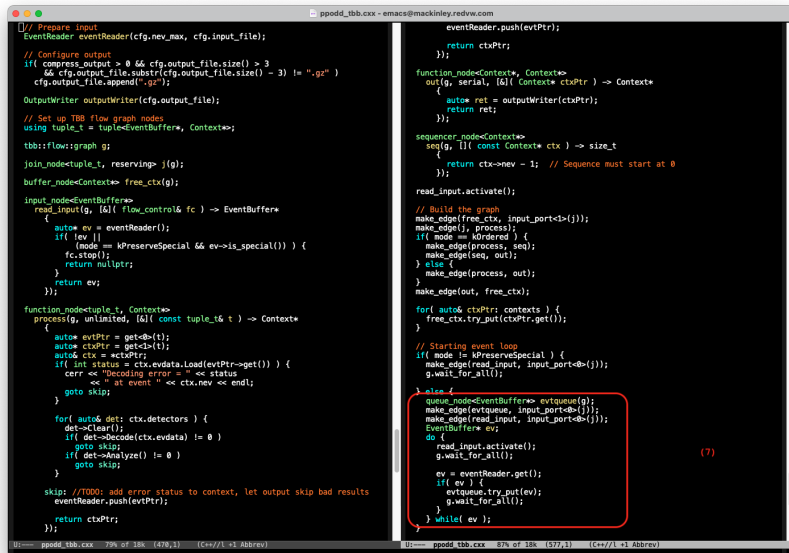
for( auto& ctxPtr: contexts ) {
    free_ctx.try_put(ctxPtr.get());
}

// Starting event loop
if ( mode != kPreserveSpecial ) {
    make_edge(read_input, input_port<0>(j));
    g.wait_for_all();
} else {
    queue_node<EventBuffer*> evtqueue(g);
    make_edge(evtqueue, input_port<0>(j));
    make_edge(read_input, input_port<0>(j));
    EventBuffer* ev;
    do {
        read_input.activate();
        g.wait_for_all();

        ev = eventReader.get();
        if ( ev ) {
            evtqueue.try_put(ev);
            g.wait_for_all();
        }
    } while ( ev );
}
```

# Podd Parallel Processing Prototype

- 1 Read raw event (serial)
- 2 Parallel processing like Podd: RawDecode, Decode, Process
- 3 Output (serial)
- 4 Optional sequencer for event ordering
- 5 Build graph according to mode
- 6 Connect input and start processing
- 7 Barriers at special events: process events in batches



```
// Prepare input
EventReader eventReader(cfg.nev_max, cfg.input_file);

// Configure output
if (compress_output > 0 && cfg.output_file.size() > 3
    && cfg.output_file.substr(cfg.output_file.size() - 3) != ".gz" )
    cfg.output_file.append(".gz");

OutputWriter outputWriter(cfg.output_file);

// Set up TBB flow graph nodes
using tuple_t = tuple<EventBuffer*, Context*>;
tbb::flow::graph g;

join_node<tuple_t, reserving> j(g);
buffer_node<Context*> free_ctx(g);

input_node<EventBuffer*>
read_input(g, [&]( flow_control& fc ) -> EventBuffer*
{
    auto* ev = eventReader();
    if ( !ev ||
        (mode == kPreserveSpecial && ev->is_special()) ) {
        fc.stop();
        return nullptr;
    }
    return ev;
});

function_node<tuple_t, Context*>
process(g, unlimited, [&]( const tuple_t& t ) -> Context*
{
    auto* evtPtr = get<0>(t);
    auto* ctxPtr = get<1>(t);
    auto& ctx = *ctxPtr;
    if ( int status = ctx.evdata.Load(evtPtr->get()) ) {
        cerr << "Decoding error = " << status
            << " at event " << ctx.nev << endl;
        goto skip;
    }
    for( auto& det: ctx.detectors ) {
        det->Clear();
        if ( det->Decode(ctx.evdata) != 0 )
            goto skip;
        if ( det->Analyze() != 0 )
            goto skip;
    }
    skip: //TODO: add error status to context, let output skip bad results
    eventReader.push(evtPtr);

    return ctxPtr;
});

eventReader.push(evtPtr);
return ctxPtr;
});

function_node<Context*, Context*>
out(g, serial, [&]( Context* ctxPtr ) -> Context*
{
    auto* ret = outputWriter(ctxPtr);
    return ret;
});

sequencer_node<Context*>
seq(g, [( const Context* ctx ) -> size_t
{
    return ctx->nev - 1; // Sequence must start at 0
}]);

read_input.activate();

// Build the graph
make_edge(free_ctx, input_port<1>(j));
make_edge(j, process);
if ( mode == kOrdered ) {
    make_edge(process, seq);
    make_edge(seq, out);
} else {
    make_edge(process, out);
}
make_edge(out, free_ctx);

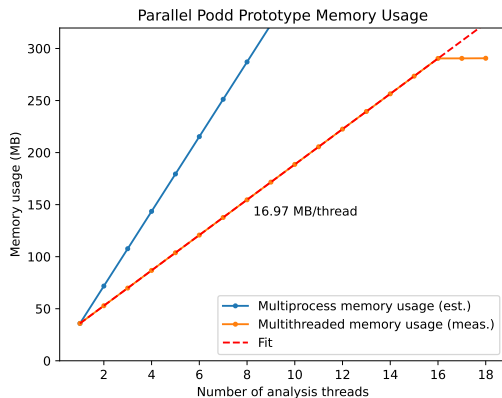
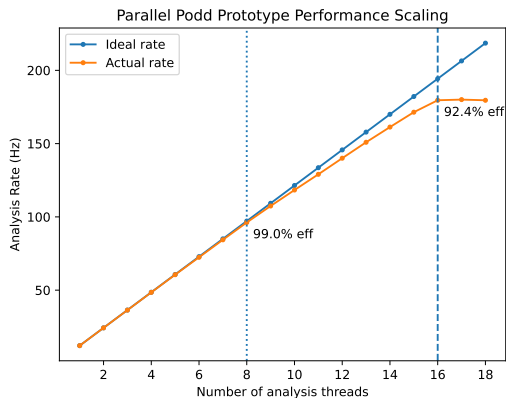
for( auto& ctxPtr: contexts ) {
    free_ctx.try_put(ctxPtr.get());
}

// Starting event loop
if ( mode != kPreserveSpecial ) {
    make_edge(read_input, input_port<0>(j));
    g.wait_for_all();
} else {
    queue_node<EventBuffer*> evtqueue(g);
    make_edge(evtqueue, input_port<0>(j));
    make_edge(read_input, input_port<0>(j));
    EventBuffer* ev;
    do {
        read_input.activate();
        g.wait_for_all();

        ev = eventReader.get();
        if ( ev ) {
            evtqueue.try_put(ev);
            g.wait_for_all();
        }
    } while ( ev );
}
```

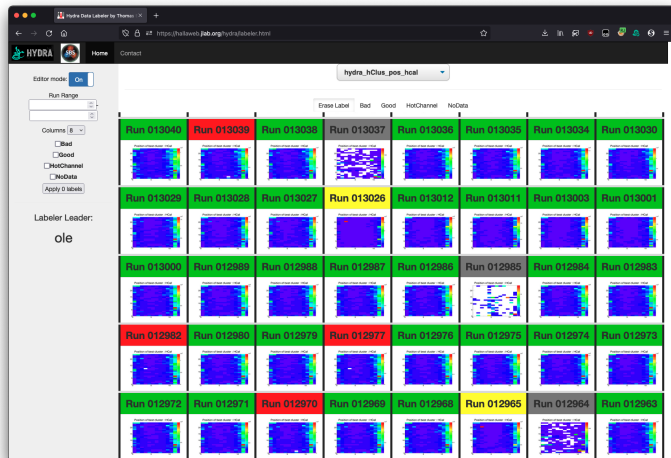
# TBB-based Parallel Podd Performance Scaling Benchmark

- Unordered mode. (Other modes are naturally less performant.)
- Processing rate and real memory usage (resident set size) as function of number of analysis threads.
- Test system: Intel i7-10700K (8C/16T) @ 3.80 GHz, 32 GB RAM, macOS 11, idle.
- 16 MB per-thread event buffer size for illustration purposes.



# AI-Assisted Online Monitoring (Hydra) — credits to Thomas Britton

- EPSCI group has offered support to deploy the Hall D Hydra system in Hall A for **automated data quality monitoring**.
- Will tap into online histograms generated by panguin.
- Test installation & database set up (see screenshot).
- One-time human review ("**labeling**") required.
- Alex Camsonne has one summer student working on bringing system fully up to speed.
- Will port to Hall C as well.





## Scientific Computing Resources — credits to Bryan Hess *et al.*

- Farm/ifarm still on **CentOS 7.9**. RHEL 8 clones being evaluated.
- Farm batch system running new **slurm** and **swif2** job scheduler. See the [Farm Users Guide](#).
- Current farm resources
  - ▶ Disk: **Lustre: 4.1 PB**, Work: 1.4 PB (recent upgrade).
  - ▶ CPU: **14192 cores / 28384 threads**. Total capacity **249 M-core-hours/year**
  - ▶ Almost half the capacity is on **AMD EPYC 7502** 64C/128T systems (speed demons!)
  - ▶ 6 nodes with Nvidia TitanRTX GPUs dedicated for ML applications
- Mass storage system (as of May 2022)
  - ▶ Throughput  $\approx$  **10 GB/s** (24 LTO-8 drives, uncompressed, theoretical)
  - ▶  $\approx$  150 PB capacity (LTO-8, uncompressed),  $\approx$  85 PB used (23.4 raw, 26.7 rawdup).
  - ▶ Significant capacity headroom (more frames, LTO-9) with current silo, up to  $\approx$  325 PB.

# Summary

- “Podd” analysis software continues to be **actively maintained** and used by current experiments in Halls A & C.
- Significant **modernization** work (multithreading etc.) ongoing.
- Various **software tasks remaining**, especially for later SBS experiments.
- The **large data volumes** from SBS are putting Hall A in the same league as Halls B & D in terms of resource needs. This requires **careful planning** for full-scale farm replays.
- Online monitoring will soon be AI-assisted, which may improve the shift taking experience.