



# Multi-threaded *art* forum

Kyle J. Knoepfel

15 September 2017

# Why are we here?

- **Education**

- We are communicating to you what (using) a multi-threaded *art* will look like
- We are providing guidance for how you approach multi-threading issues that you will encounter

- **Your input**

- We need your perspective for what behaviors/features are needed/desired
- Each experiment/project using *art* has different requirements

- This is intended to be an ongoing discussion

# For today

- Motivation for a multi-threaded *art*
- (Some) complications with multi-threading
- *art's* approach to implementing multi-threading

## Motivations for a multi-threaded *art*

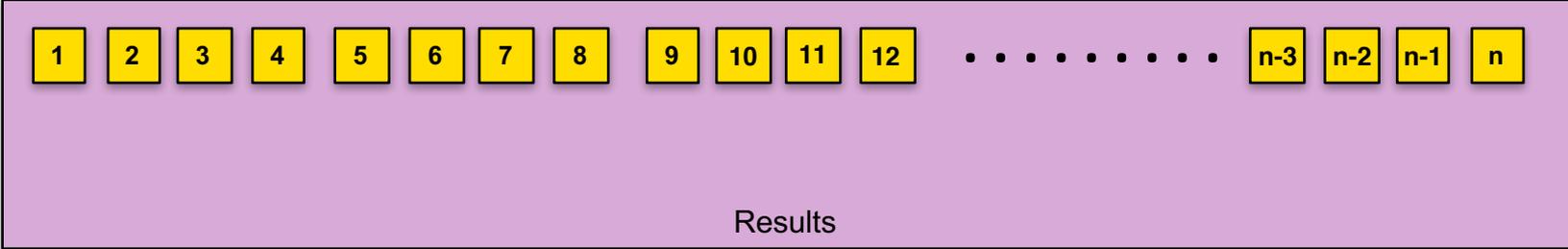
# Parallelism and particle physics event processing

- Particle physics relies on statistical independence of events.
  - It is fundamentally well-suited for parallel processing of events.



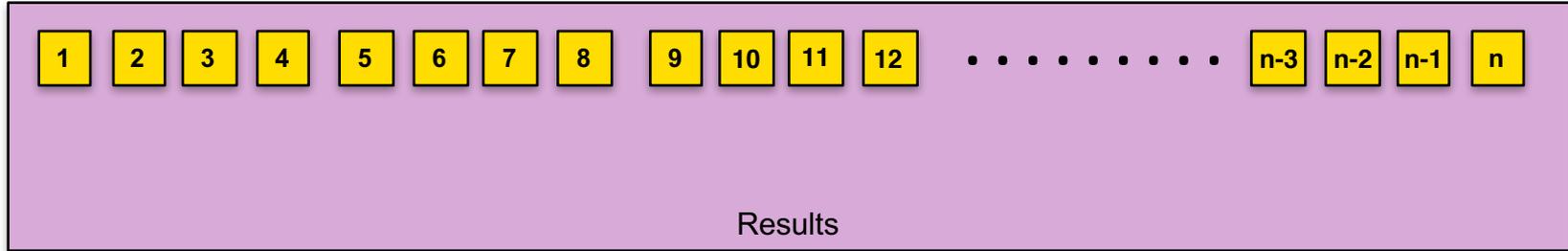
# Parallelism and particle physics event processing

- Particle physics relies on statistical independence of events.
  - It is fundamentally well-suited for parallel processing of events.



# Parallelism and particle physics event processing

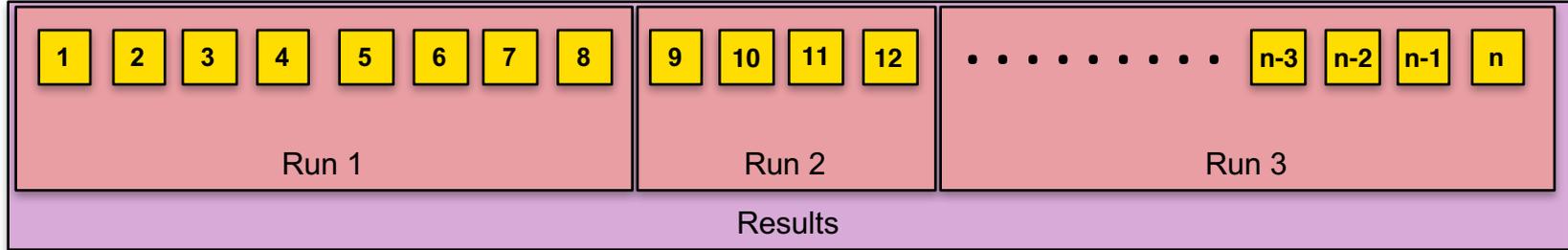
- Particle physics relies on statistical independence of events.
  - It is fundamentally well-suited for parallel processing of events.



- Since events are statistically independent, disjoint collections of events are also statistically independent.

# Parallelism and particle physics event processing

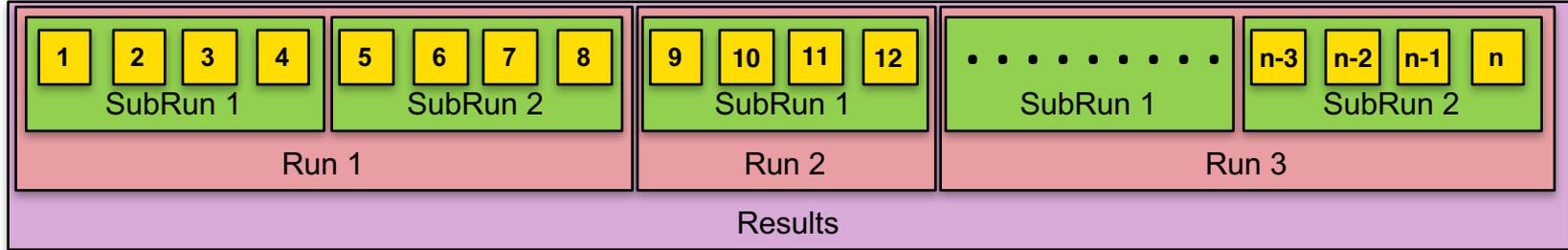
- Particle physics relies on statistical independence of events.
  - It is fundamentally well-suited for parallel processing of events.



- Since events are statistically independent, disjoint collections of events are also statistically independent.

# Parallelism and particle physics event processing

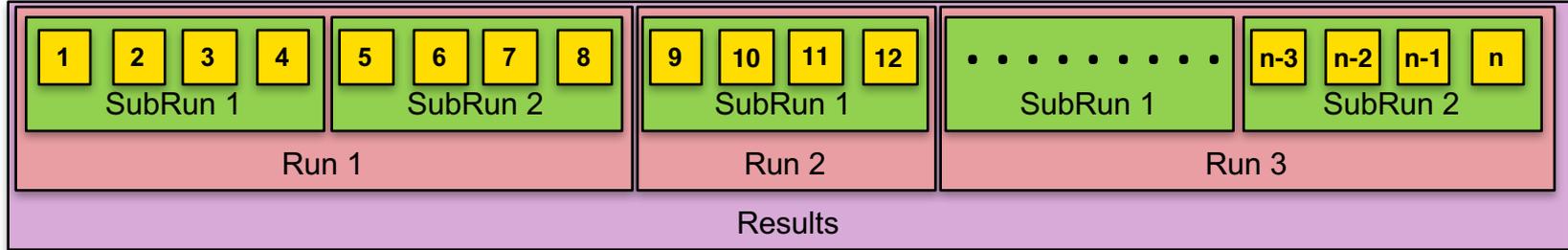
- Particle physics relies on statistical independence of events.
  - It is fundamentally well-suited for parallel processing of events.



- Since events are statistically independent, disjoint collections of events are also statistically independent.

# Parallelism and particle physics event processing

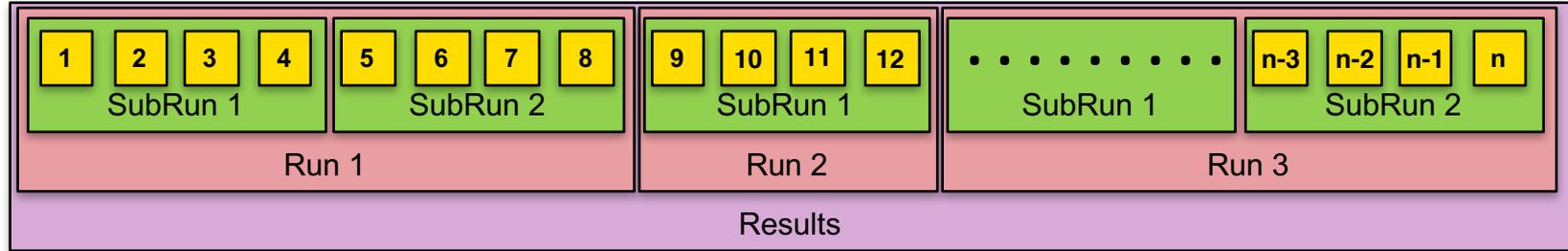
- Particle physics relies on statistical independence of events.
  - It is fundamentally well-suited for parallel processing of events.



- Since events are statistically independent, disjoint collections of events are also statistically independent.
- There are, thus, no fundamental limitations to processing concurrent Events, SubRuns, Runs, and Results/input files.

# Parallelism and particle physics event processing

- Particle physics relies on statistical independence of events.
  - It is fundamentally well-suited for parallel processing of events.



- Since events are statistically independent, disjoint collections of events are also statistically independent.
- There are, thus, no fundamental limitations to processing concurrent Events, SubRuns, Runs, and Results/input files.
- We are designing a system that recognizes this.

## Other motivations

1. Hardware is moving to greater number of cores, and less RAM per core.
2. Reduction in memory use of simulation, reconstruction, and analysis programs using *art* by allowing  $n$  working cores to share data between threads.
3. Mitigation of the impact of rare events that are unusually large.
4. Simplification of the description of workflows that process large amounts of data.
5. Improved efficiency of workload scheduling.
6. Reduction of the load on the workflow management system.

# Multi-threading concepts

# Multi-threading pitfalls

- **Data race:**
  - when two or more threads attempt to update the state of an object at the same time
  - when one thread is reading an object while another thread is updating it
- Data races result of sharing memory among threads.
- If there are no shared objects in your code, then you have no concerns.
- If you *do* have a shared object *and* it has mutable state, then you must take steps to ensure that data races cannot occur (e.g.):
  - Design your code so that there cannot be a data race.
  - Use structures that provide atomic operations on the shared data.
  - Consider using mutual exclusion to protect the critical regions.
- Easier said than done:
  - Any libraries you use may share memory among threads (e.g. ROOT)

# Multi-threading pitfalls

- **Data race:**
  - when two or more threads attempt to update the state of an object at the same time
  - when one thread is reading an object while another thread is updating it
- Data races result of sharing memory among threads.
- If there are no shared objects in your code, then you have no concerns.
- If you *do* have a shared object *and* it has mutable state, then you must take steps to ensure that data races cannot occur (e.g.):
  - Design your code so that there cannot be a data race.
  - Use locks to ensure mutual exclusion.
  - Cache coherency.
- Easiest way to avoid data races is to avoid shared memory among threads (e.g., use local variables).
- Always use locks when you do share memory among threads (e.g., use mutexes).

Multi-threaded programs are frequently non-deterministic—  
*i.e.* the ordering of concurrently-executing tasks will not be the  
same from one program execution to the next

# Is your module a shared object?

- Who owns your module?
- *art* owns the module objects, which are created at run-time based on the configuration you provide.
- You provide the *definition* of the module class:
  - *art* knows very little of your module's definition
  - ***art*** calls module functions via C++ polymorphism
- Suppose *art* were to call your `produce` function concurrently on multiple events.

# Module examples

- We want to create a track from a collection of hits

```
void TrackMaker::produce(art::Event& e)
{
    auto const& hits = e.getValidHandle<Hits>(tag_);
    unique_ptr<Track> track = trackFromHits(*hits);
    e.put(move(track));
}
```

# Module examples

- We want to create a track from a collection of hits

```
void TrackMaker::produce(art::Event& e)
{
    auto const& hits = e.getValidHandle<Hits>(tag_);
    unique_ptr<Track> track = trackFromHits(*hits);
    e.put(move(track));
}
```

- Assuming `trackFromHits` does not update any state, then this `produce` function is thread-safe—i.e. it can be called concurrently with different `art::Event` objects.
- Why?
  - *art* guarantees that product retrieval and insertion is thread-safe
  - the `produce` function above modifies no state of the `TrackMaker` object

# Module examples

- Suppose we want to add an event counter:

```
void TrackMaker::produce(art::Event& e)
{
    ++nEvents_;
    // ...
}
```

- Is this thread-safe?

# Module examples

- Suppose we want to add an event counter:

```
void TrackMaker::produce(art::Event& e)
{
    ++nEvents_;
    // ...
}
```

- Is this thread-safe?
- Answer: it depends on the **type** of `nEvents_`:
  - If the type is an integral fundamental type (e.g. `unsigned int`), then **no**, it is not thread-safe, since `operator++` requires a read and then write.
  - If the type is an `std::atomic<unsigned int>`, then **yes**, it is thread-safe.

# Module examples

- Suppose we want to add an event counter:

```
void TrackMaker::produce(art::Event& e)
{
    ++nEvents_;
    // ...
}
```

- Is this thread-safe?

- A
  - Determining thread-safety of module code takes analysis.
  - *art* will provide various module types that tell the framework whether a given module can support multi-threaded processing

if the type is an `std::atomic<unsigned int>`, then **yes**, it is thread safe.

# Indications of thread-unsafe C++ code

- Functions (free or member) which access a global object whose state can change, including non-`const` function-scope `static` data.
- Functions (free or member) which change the state of objects which were passed as `const` function arguments (e.g. casting away `const` on an argument).
- `const` non-`static` member functions which modify the state of the object on which they are called (e.g. `mutable` members, or casting away `const` on this).
- Pointer member data or data held by member data being passed as a non-`const` argument to functions.
- `const` member functions returning values of member variables which are pointers to non-`const` items.

# General guidelines

- Apply `const` liberally
- Avoid using non-`const static` variables in functions/classes
- To the extent possible, do not use the `mutable` keyword
- Use as few global objects as possible
- If you must use a global object, provide only `const`-qualified interface

# General guidelines

- Apply `const` liberally
- Avoid using non-`const static` variables in functions/classes
- To the extent possible, do not use the `mutable` keyword
- Use as few global objects as possible
- If you must use a global object, provide only `const`-qualified interface

All of these are good ideas for single-threaded code.  
You can do this now!

## ***art's* approach to implementing multi-threading**

# Approaching the design

- The design of a multi-threaded framework should be based on fundamental principles, **not** on the limitations of external dependencies.
  - The relevant questions are:
    - In what contexts does multi-threading make sense?
    - In what contexts does multi-threading not make sense?
    - Not, when *can* we do multi-threading and when *can we not*.
  - The implementation must accommodate any limitations, not cater to them.

# Approaching the design

- The design of a multi-threaded framework should be based on fundamental principles, **not** on the limitations of external dependencies.
  - The relevant questions are:
    - In what contexts does multi-threading make sense?
    - In what contexts does multi-threading not make sense?
    - Not, when *can* we do multi-threading and when *can we not*.
  - The implementation must accommodate any limitations, not cater to them.
- We have striven for a balance between complexity and efficiency:
  - Our preference is to have a slightly less efficient, easier-to-understand system than a slightly more efficient, difficult-to-understand system.

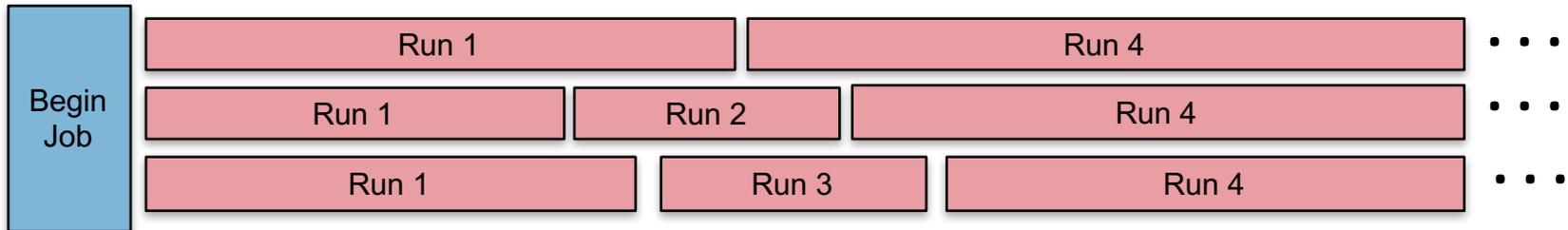
# User interface

- *“No framework code is so precious that it must be saved; nothing is untouchable. Our users' code is precious, and we should break as little of it as possible.”*
- To the extent possible, the user interface should reflect only physically meaningful concepts—i.e. those concepts that are already known to *art* users: `Results`, `Run`, `SubRun`, and `Event`. It should not reflect implementation details.
  - Exceptions to this could be for those developing services or other more-expert facilities than modules.

# The design

- Largely based off of CMSSW's design
  - We use Intel's Threading Building Blocks (TBB)
  - Steps to be performed are factorized into *tasks*
  - You can think of a call to your module's "produce" function as performing a task
- Users specify the number of concurrent event loops and (optionally) the maximum number of threads that the process can use.
- **Each loop processes one event at a time.**

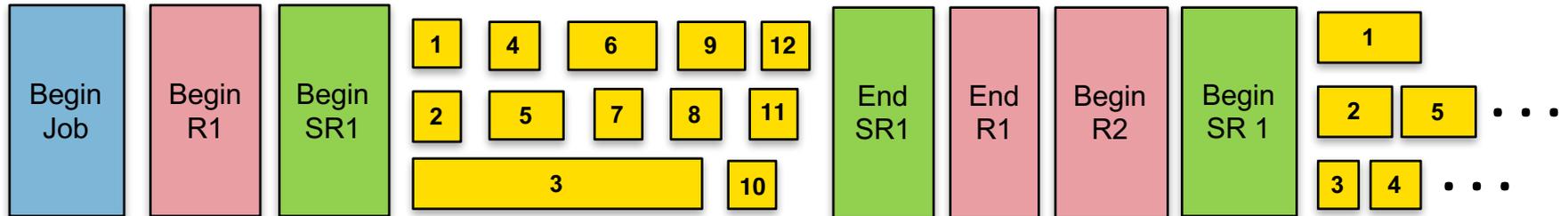
## Our goal:



# The design

- Largely based off of CMSSW's design
  - We use Intel's Threading Building Blocks (TBB)
  - Steps to be performed are factorized into *tasks*
  - You can think of a call to your module's "produce" function as performing a task
- Users specify the number of concurrent event loops and (optionally) the maximum number of threads that the process can use.
- **Each loop processes one event at a time.**

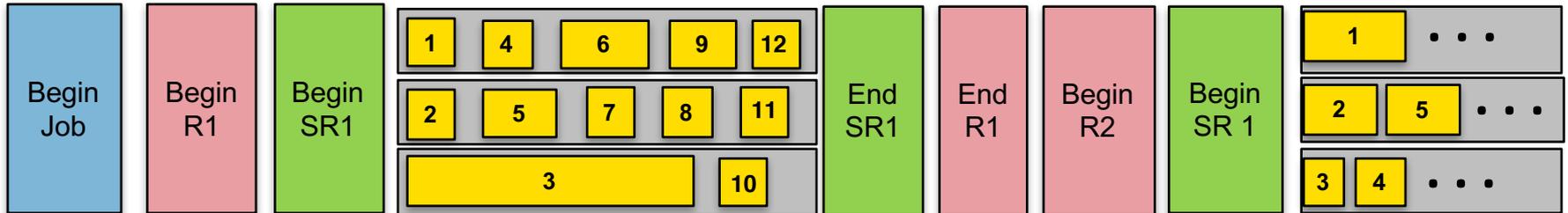
## Currently implemented:



# The design

- Largely based off of CMSSW's design
  - We use Intel's Threading Building Blocks (TBB)
  - Steps to be performed are factorized into *tasks*
  - You can think of a call to your module's "produce" function as performing a task
- Users specify the number of concurrent event loops and (optionally) the maximum number of threads that the process can use.
- **Each loop processes one event at a time.**

## Currently implemented:



# The design

- Largely based off of CMSSW's design
  - We use Intel's Threading Building Blocks (TBB)
  - Steps to be performed are factorized into *tasks*
  - You can think of a call to your module's "produce" function as performing a task
- Users specify the number of concurrent event loops and (optionally) the maximum number of threads that the process can use.
- **Each loop processes one event at a time.**
- Different modules will also be able to be run in parallel on the *same* event.
- Users are allowed to use TBB's parallel facilities within their own modules.

## General statements—*art*

- The first release with multi-threaded *art* will be version **3.0**.
- Our intention is that all currently-provided features will still be supported
- Standard guidance for *art* usage is very important:
  - No communication between modules
  - The order in which paths are executed is unspecified—you cannot rely on any specific ordering
- Take advantage of the *consumes* interface:
  - catch data-dependency errors in your workflow
  - *art* may be able to optimize your program execution based on the information you provide via *consumes*.

# General statements—plugin configuration/construction

- If you are using configuration validation, all validation must happen at plugin construction time (the general use case)
- If you are invoking `art::make_tool`, that must happen at plugin construction time

# General statements—modules

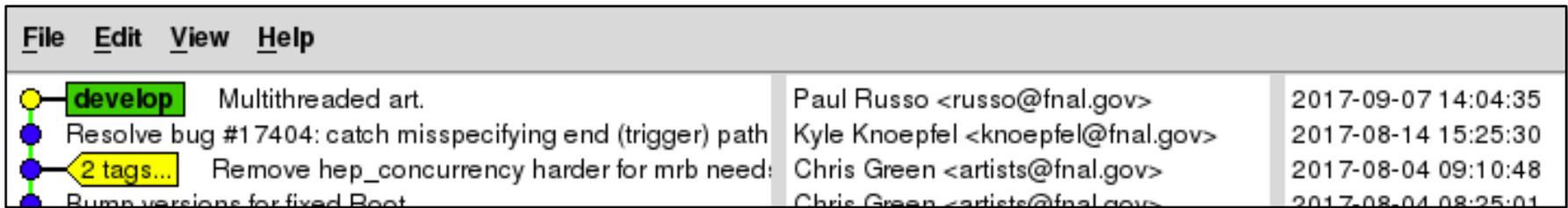
- *art* guarantees that any currently-existing modules (to within some interface changes) will be usable in a multi-threaded execution of *art*.
  - No multi-threading benefits will be realized with such “legacy” modules
- To take advantage of *art*'s multi-threading capabilities, users will need to change the kind of module they use:
  - **Serialized module**: use if the facilities you are using do not allow for concurrent execution and you must see all events
  - **Per-event loop module**: for a configured module, one copy of that module is produced per event loop—each module copy sees one event at a time. Use if moving to a fully concurrent module is unfeasible
  - **Fully concurrent module**: module functions can be called concurrently without any data races

## General statements—services

- The activity registry is likely to change (adjustments to some callbacks, addition of others)
- All *art*-provided services that are intended to be used in a multi-threaded context are thread-safe.
- There are some services that cannot be thread-safe (e.g. `TFileService`)
  - We will provide you with instructions as to how they can be used
- Generally speaking, all of your services must be thread safe

# art's status

- Many preparatory changes over the last year
  - State machine has been removed
  - Relevant services have made thread-safe
  - All registries have been made thread-safe
  - consumes interface has been introduced
- Multi-threaded implementation was just committed last week:



The screenshot shows a Git commit log with a menu bar at the top containing 'File', 'Edit', 'View', and 'Help'. The log entries are as follows:

Commit Hash	Message	Author	Date
develop	Multithreaded art.	Paul Russo <russo@fnal.gov>	2017-09-07 14:04:35
	Resolve bug #17404: catch misspecifying end (trigger) path	Kyle Knoepfel <knoepfel@fnal.gov>	2017-08-14 15:25:30
	Remove hep_concurrency harder for mrb need:	Chris Green <artists@fnal.gov>	2017-08-04 09:10:48
	Bump versions for fixed Boost	Chris Green <artists@fnal.gov>	2017-08-04 08:25:01

## Next steps

- Before we release *art 3.0*, we need your input:
  - What behavior will be needed for the concurrent-aware modules?
  - Are there specific constraints your experiment has regarding processing sub-runs concurrently?
  - What is the desired reproducibility/replayability behaviors for the `RandomNumberGenerator` service?
  - Do you explicitly use `TriggerResults` objects?
  - *etc.*

Over the next few weeks, we'll explore these issues.