

Stitched Together: Transitioning CMS to a Hierarchical Threaded Framework

CD Jones and E Sexton-Kennedy

Fermilab, P.O.Box 500, Batavia, IL 60510-5011, USA

E-mail: cdj@fnal.gov, sexton@fnal.gov

Abstract. Modern computing hardware is transitioning from using a single high frequency complicated computing core to many lower frequency simpler cores. As part of that transition, hardware manufacturers are urging developers to exploit concurrency in their programs via operating system threads. We will present CMS' effort to evolve our single threaded framework into a highly concurrent framework. We will outline the design of the new framework and how the design was constrained by the initial single threaded design. Then we will discuss the tools we have used to identify and correct thread unsafe user code. Finally we will end with a description of the coding patterns we found useful when converting code to being thread-safe.

1. Introduction

HEP computing has enjoyed the previous CPU hardware era where clock frequencies were doubling every 18 months. This allowed us to gain immediate benefits when using new computing hardware without even having to recompile our code. Unfortunately, those days are past and we now live in an era where CPU hardware vendors are using the exponential increase in transistor counts to increase the number of CPU cores. Therefore, to make use of this new hardware, our software must exploit concurrency.

Traditionally, HEP computing has utilized the increase CPU core count by increasing the number of single threaded jobs we run. Unfortunately, this does not scale for grid sites. As core counts double, grid sites are forced to double the number of batch slots, double the amount of memory on a machine, double the open file handles and double the number of network connections. By making the HEP applications concurrent, we can reduce these burdens on the sites and potentially target new sites whose single-threaded job resources were too limited for our single-threaded applications.

CMS' [1] challenge in switching to a concurrent framework is to avoid rewriting millions of lines of C++ legacy code to work in a new framework. In this paper we will discuss the design of the new threaded framework, emphasizing how it eases migration. After the design discussion we will cover in more depth how the framework accommodates code with various levels of thread safety. We will then end by detailing the thread safety tools we have found useful during the conversion.

2. Design

2.1. Legacy Design

Before describing the design for the threaded framework, we first need an understanding of the legacy framework design. There are two main concepts to address from the legacy framework: system state transitions and event processing.



Figure 1. The sequence of states traversed by an example job of the legacy framework.

The legacy framework has a series of guaranteed state transitions it progresses through. Algorithms run by the framework are allowed to do activities on each of these transitions. Figure 1 displays the sequence of states traversed by a short example job that could be run with the legacy framework. Once configuration has finished, the framework enters the *begin job* transition. The framework then switches into data processing. The CMS data model requires Events to be contained within a Luminosity block and Luminosity blocks are contained within a Run. The framework then proceeds through data driven transitions that reflect this hierarchy. The first such data transition is the *begin run*. Once completed, the framework moves to a *begin luminosity block* transition. After that the framework proceeds through a series of *event* transitions that end with an *end luminosity block* transition. The sequence of ‘*begin luminosity block, events, end luminosity block*’ transitions continue until we reach an *end run* transition. At that point the whole sequence, starts again at *begin run*, can occur, or the job can enter the *end job* state. Once the *end job* transition finishes, the application shuts down.

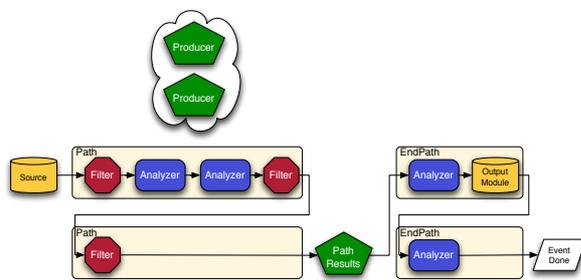


Figure 2. The time sequence of modules called during the processing of an event. Producers are modules that make new data products and are run the first time another module requests their data. Filter modules can halt the processing of all further modules on a path. Analyzer modules just read event data. When all paths have finished, the Path Results Producer records their success or failure. After that, each EndPath is run in turn. OutputModules write event data to storage and can only be used on an EndPath.

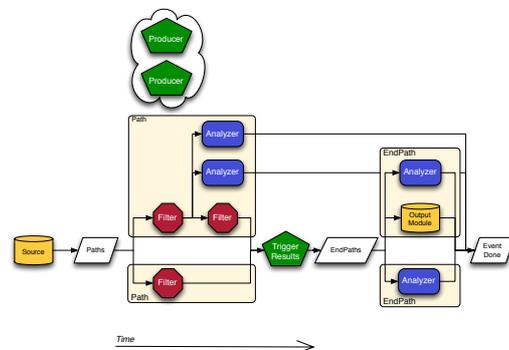


Figure 3. The concurrent ordering of modules during event processing in the new framework. Arrows show dependencies, which must be met before the next step can be run.

In the legacy framework, user supplied algorithms are encapsulated into modules. It is these modules that are informed when the framework changes state. The exact order in which modules are called is dictated by the schedule explicitly declared in the configuration. A module is guaranteed to be called

for all state transitions, except for *event* transitions. Figure 2 illustrates how module processing is achieved for an event. Modules are placed in sequences called Paths and EndPaths. All Paths are guaranteed to finish before any EndPath is started. In addition, Paths may be stopped before reaching the last module in their sequence while EndPaths always run to completion. An Event is processed as follows. The Event is first read from the Source and then passed to the first Path. The Path passes the Event to each module it holds in the given sequence order. A Filter module is allowed to halt all further progress in a Path so that modules later in the sequence never see that particular Event. An Analyzer module is only allowed to read data from the Event and cannot alter the Event or affect the processing of the Event. Modules called Producers create new data associated with the Event. Producers run the first time another module requests their data, for that Event. Once the last Path in the sequence of Paths finishes, the framework runs a special “Path Results” Producer that adds the success or failure information about each Path into the Event. Once this has completed, the framework runs the first EndPath. The EndPath contains a sequence of Analyzers and OutputModules. OutputModules read data from the Event and write it out to storage. Once the last EndPath completes, the processing for that Event is done.

2.2. Threaded design

The threaded framework design makes use of the naturally concurrent elements of the legacy framework. These elements occur at three different conceptual levels. At the highest level, the threaded framework runs multiple data transitions, e.g. events, concurrently. This is supported by introducing two new concepts to the framework: Stream and Global. The middle level supports running multiple modules concurrently while processing one transition, e.g. an event. To do that correctly, one must take into account the inter-module dependencies. To aid transitioning to the new framework, we also want to minimize any required changes to module code. At the lowest level, the threaded framework supports running multiple concurrent tasks within one module.

All three levels of concurrency are implemented using Intel’s Threaded Building Blocks [2] (TBB) library. In TBB, one breaks down concurrent work into *tasks* which TBB can then run in parallel. The framework therefore decomposes processing of transitions and modules into appropriate tasks, and then hands these tasks at the appropriate time to TBB. Following well defined rules; Module code is allowed to directly call TBB. TBB can then appropriately schedule all three levels of tasks.

2.2.1. Concurrent transitions

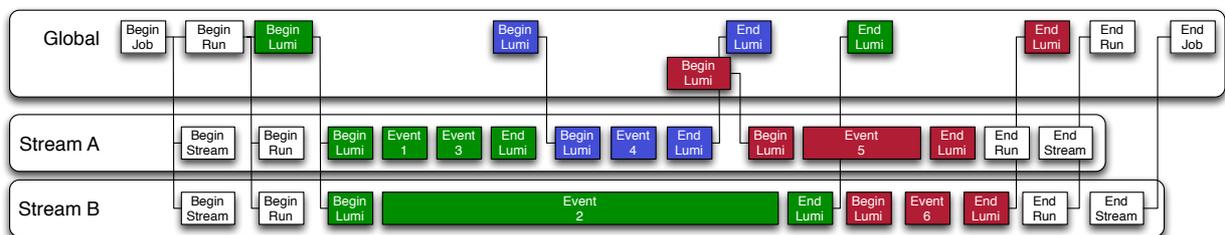


Figure 4. The concurrent sequence of states traversed by an example job of the threaded framework.

In the threaded framework, transitions are separated into Global and Stream types. Modules see Global transitions on a ‘global’ scale. For example, modules see *global begin run* and *global begin luminosity block* when the source first reads them. Similarly, modules see *global end run* and *global end luminosity block* once all processing has finished for that particular Run or Luminosity block. Global transitions can run concurrently (see the red *begin luminosity block* and blue *end luminosity block* in figure 4) and can even end in a different order than they began (see the green and blue Luminosity

blocks in figure 4). Although there are Run and Luminosity block global transitions, there are no global event transitions. Events are only Stream transitions.

A Stream processes data transitions serially: begin run, begin luminosity block, events, end luminosity block and finally end run. The sequence of transitions is identical to what one sees in the legacy framework. In addition, multiple Streams can be running concurrently, each with its own events. Therefore one Stream only sees a subset of the Events processed by the job. The legacy CMS framework is equivalent to running the threaded framework with only one Stream. In the threaded framework, Paths and EndPaths are a per Stream construct because Paths and EndPaths are only used for Event filtering and Events are only seen by Streams. Also, the same module instance can be shared across Streams and it is the Stream's responsibility to know if a module was run already for a particular Event.

2.2.2. Concurrent modules

Figure 3 illustrates how modules are concurrently run when doing Event processing. Just as in the legacy framework case, the Event is first obtained from the source. In this case, however, all Paths are scheduled to run concurrently. We cannot run all modules within a Path simultaneously if any of the modules is a Filter since a Filter is meant to stop any further processing of an Event for the Path. However, once a Filter has made its decision, all Analyzers between that Filter and the next Filter on the Path can be run concurrently with the next Filter (as shown in figure 3). Producers are run concurrently based on data requests from other modules. The framework waits for all Paths to finish before running the Trigger Results Producer. Once that has occurred, the framework runs all modules on all EndPaths simultaneously. The framework once again waits until all EndPaths have finished before declaring that Event done. Only once the Event is done can a Stream request a new Event from the Source.

2.2.3. Concurrent tasks

Developers are allowed to use TBB directly inside of their modules to add concurrent calculations. TBB efficiently handles scheduling tasks for both modules and sub-modules using only the number of threads for which it was configured. The easiest way to use TBB is simply to use one of its convenience functions such as `tbb::parallel_for`. More complex algorithms can be parallelized by creating new task classes by inheriting from `tbb::task` and then telling TBB to spawn those tasks. No matter which method is used by a developer, the framework requires that all tasks spawned by a module for a given transition call-back must complete before the module returns from the call-back.

3. Thread Safety

The legacy framework is composed of individual modules which communicate indirectly with each other by publishing data products to a shared container, the `edm::Event`, and then reading those published data products from the container. The threaded framework maintains the module and data product design but requires that both types obey some thread-safety rules.

In both the legacy and threaded frameworks, data products are passed to modules via the `const` modifier. This is done to enforce the rule that once published, data products cannot be changed. Therefore only `const` member functions of a data product are required to be thread-safe. Fortunately, this exactly matches the C++ 11 library guarantee that states calling only `const` functions of standard library objects from different threads is thread-safe.

Modules are the framework components that utilize the majority of physicist defined code. As such, the thread safety requirements of modules will be the requirements that most affect physicists. The threaded framework uses four different varieties of module, where each variety defines a different level of thread safety. The four module types are: Stream, Global, One and Legacy.

3.1. Stream modules

A Stream module is declared once in the job configuration but then replicated for each Stream. For example, if the job configuration specified 8 Streams then there would be 8 replicas of each Stream module. This is done since each Stream only processes one transition, e.g. one event, at a time. Therefore a Stream module replica will never be called concurrently which means a Stream module's member data does not need to be thread safe. The only requirement is the Stream module code does not interact with any non-const static C++ objects.

One thing to keep in mind, is one Stream only sees a subset of all the Events processed in a job. Therefore Stream modules also only see a subset. This means algorithms that require seeing all Events cannot be implemented as Stream modules. Fortunately, most Producer and Filter modules only operate on one Event at a time and therefore can be easily implemented as Stream modules.

We made it easy to convert from our present modules to Stream modules. The only modification needed is to change which class one inherits from. So for Producers, one just changes from inheriting from `Producer` to inheriting from `stream::Producer<>`.

3.2. Global modules

Unlike Stream modules, there is only one instance of each Global module per module declared in the configuration. All Streams share the single instance. Therefore, a Global module can see all run, Luminosity blocks and Event transitions. However, this means a Global module will be called concurrently. This requires all member functions and member data of Global modules must be thread-safe. To emphasize this requirement, all member functions called on a framework data transition are declared `const`. This requires the developer to either never change the values during the function call or declare the variables as `mutable`. Such mutable variables are easy to find by a static analyzer that can check that the members are intrinsically thread-safe. The interface for the Global module provides ways to help developers with thread-safety. The primary way is to allow users to declare per transition caches which are then managed by the module's base class. For example, a module could declare it wants to cache an `int` for each Stream. Then when processing an Event the module can request from the base class the `int` which is appropriate for that Stream.

Global modules are intended only for specialized needs. One such need would be to share as much memory across Streams as possible in order to run on limited hardware. Another such need would be if an algorithm must see all Runs, Luminosity blocks or Events and must be as performing as possible. We anticipate using a Global module as the way to write a highly performant `OutputModule`.

3.3. One modules

The One module has that name since there is only one instance shared by all Streams and they only see one transition at a time. Similar to Global modules, One modules see all framework data transitions. However, unlike a Global module instance, a One module instance is serialized by the framework so that the instance only sees one transition at a time. Since they are serialized, it means the member data of One modules does not have to be thread-safe.

A major difference between a One module and both Stream and Global modules is One modules can safely share resources between instance of the same module class type or different module class types. This is done to allow the framework to utilize algorithms that internally contain 'global' data, e.g. legacy FORTRAN based monte carlo event generators. Each One module declares which shared resources it will be using. The framework then guarantees that only one module using that shared resource is run at a time. Therefore access to the shared resource is serialized.

It is very easy to convert an existing module to a One module. Just like in the Stream module case, all one has to do is change the inheritance. To change a Producer one would switch from inheriting from `Producer` to inheriting from `one::Producer<>`. We believe One modules will be good for OutputModules which are not performance critical and for Analyzers which are making ROOT ntuples.

3.4. Legacy modules

Legacy modules are just modules which have not been explicitly ported to the threaded framework. The only work that needs to be done to make them work in the threaded framework is to recompile them. However, the threaded framework assumes that a Legacy module can possible interfere with any other Legacy module. Because of that assumption, the framework only allows one Legacy module to run at a time. This serialization avoids possible data races but leads to performance problems when using more than one thread.

The reason Legacy modules are supported in the threaded framework is to easy code migration to the new framework. Modules can be ported to the Stream, Global, or One variety, one at a time while still maintaining a fully running framework.

4. Tools

In order to meet the challenge of migrating a large body of code like CMSSW into thread-safe implementations, we have developed tools to automate the discovery of non-conforming code. In addition, we found the use of run time tools for finding errors very important.

4.1. Static Code Analysis: Clang

The Clang tool suite [3] has a framework that gives users the ability to extend it's run checking by adding user define rules to the default set provided with the tool. We have extended the clang static analyzer with customizations that find coding patterns that violate the rules of the threaded framework. The checkers fall into two different categories; one that finds problematic data products, and one that finds problematic modules. The framework needs to be able to run multiple instances of any particular module concurrently. Non-const static member data represents shared state between different instances of the same module. We can use our custom extensions to Clang to look for modules that make use of non-const static data. The search has to include looking at all functions that the module calls, and each function they call, all the way down the stack.

Data products are shared between modules. They are the sole source of communication between the modules. Once a producer makes a data product and gives ownership of it to the Event, the framework can only allow const access to it. Problematic products contain non-const statics, mutable member data which is not `std::atomic<>`, member functions that cast away const on member data, pointer member data that is returned from a const function, and pointer member data that is passed as a

non-const argument to a function. All of these are checked by our extensions to Clang. The checks have to be done recursively on all data members that are classes.

4.2. Run time checking: Helgrind

Helgrind is a tool within the Valgrind tool suite [4]. It can be used to search for possible race conditions between threads without requiring the race condition to happen. It does this by recording every memory read/write done in an executing program by each thread and flags if multiple threads use the same memory address if one of them is doing a write to that address. The tool understands posix synchronization mechanisms that protect memory, i.e. mutex, semaphore, pthread_join, so that it can ignore these addresses. The following output of Helgrind illustrates this:

Possible data race during write of size 1 at 0x8D878A0 by thread #7

Locks held: none

at 0x8E7B62C: MessageLogger::establishModule(...) (in libFWCoreMessageService.so)...
by 0x49CF6E9: EventProcessor::processEvent(unsigned int) (in libFWCoreFramework.so)

This conflicts with a previous write of size 1 by thread #2

Locks held: none

at 0x8E7B62C: MessageLogger::establishModule(...) (in libFWCoreMessageService.so)...
by 0x49CF6E9: EventProcessor::processEvent(unsigned int) (in libFWCoreFramework.so)

Address 0x8D878A0 is 144 bytes inside a block of size 152 alloc'd at 0x4807A85:

operator new(unsigned long) (in vgpreload_helgrind-amd64-linux.so)...

The first paragraph of the Helgrind output says where the second read/write to the same memory location occurred; followed by the previous use, and ends with a stack trace of where the memory was initially allocated. Unfortunately, Helgrind currently does not understand lock-free designs, e.g. designs that use `std::atomic<>`, generating lots of false positives in those cases. Even with this deficiency, we found Helgrind to be an invaluable tool in debugging the implementation of the threaded framework.

5. Conclusions

The challenge of converting millions of lines of code, from serial to concurrent execution, is a huge task, and in some ways more difficult than the move from Fortran to C++ a decade ago. We have chosen a design that minimizes the work of the current framework users. By following well-defined rules about the subset of C++ that is allowed, their code can run in a concurrent framework that allows parallel execution of Events and modules. The rules are simple enough that with the help of automated tools we can find violations of these rules and get them fixed as they appear.

6. References

- [1] CMS Collaboration R. Adolphi et al., The CMS experiment at the CERN LHC, JINST, 0803, S08004 (2008)
- [2] <https://www.threadingbuildingblocks.org>, Jan. 25, 2014
- [3] <http://clang.llvm.org>, Jan. 25, 2014
- [4] <http://valgrind.org>, Jan. 25, 2014

Acknowledgments

Operated by Fermi Research Alliance, LLC under Contract No. DE-AC02-07CH11359 with the United States Department of Energy.