

Implementation of a Multi-threaded Framework for Large-scale Scientific Applications

E Sexton-Kennedy¹, Patrick Gartung¹, CD Jones¹, David Lange²

¹Fermilab, P.O.Box 500, Batavia, IL 60510-5011, USA

²Lawrence Livermore National Laboratory, 7000 East Ave, Livermore, CA 94550

E-mail: sexton@fnal.gov , cdj@fnal.gov, gartung@fnal.gov, David.Lange@cern.ch

Abstract. The CMS experiment has recently completed the development of a multi-threaded capable application framework. In this paper, we will discuss the design, implementation and application of this framework to production applications in CMS. For the 2015 LHC run, this functionality is particularly critical for both our online and offline production applications, which depend on faster turn-around times and a reduced memory footprint relative to before. These applications are complex codes, each including a large number of physics-driven algorithms. While the framework is capable of running a mix of thread-safe and "legacy" modules, algorithms running in our production applications need to be thread-safe for optimal use of this multi-threaded framework at a large scale. Towards this end, we discuss the types of changes, which were necessary for our algorithms to achieve good performance of our multi-threaded applications in a full-scale application. Finally performance numbers for what has been achieved for the 2015 run are presented.

1.Introduction and Goals

It has become clear that to make use of the new hardware architectures coming to market in the next 10 years, our software must exploit concurrency. This has been described in HEP computing conferences in the past, for instance the CHEP conference in 2011 [1].

Traditionally, HEP computing has utilized the increasing 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. Other goals include: 1. reduced processing latency for the larger event count files we expect to collect in Run 2, 2. more resource sharing between cores, including shared file handles, shared network connections, and sharing of infrequently updated memory holding calibrations, conditions, and I/O buffers 3. minimize changes to the existing framework. CMS' [2] 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 briefly review the design of the new threaded framework, for a more complete description of the design see [3]. We will then discuss the thread safety tools and migration strategies we have found useful during the conversion. Finally performance results for the Run 2 reconstruction application will be presented.

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 describe from the legacy framework: system state transitions and event processing. In the legacy framework, user supplied algorithms are encapsulated into modules. It is these modules that are informed when the framework changes state.

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.

Eventually all three levels of concurrency will be 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. TBB can then appropriately schedule all three levels of tasks. Module code is allowed to directly call TBB however it must follow well-defined rules. The framework requires that all tasks spawned by a module for a given transition call-back must complete before the module returns from that call-back.

2.2.1. Current Status

As of the writing of this paper, the framework supports parallel execution at the highest and lowest levels. Non-event transitions are serialized as in Figure 1. At the highest level multiple events will execute in parallel in different streams according to the number specified in job configuration. At the lowest level developers are allowed to use TBB directly inside of their modules to add concurrent calculations.

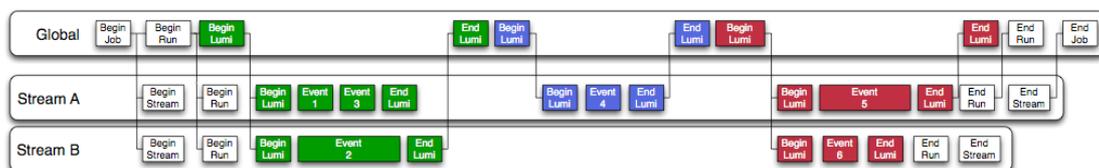


Figure 1. Note that only events are executed concurrently others are synchronised

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 products are required to be thread-safe. Fortunately, this exactly matches the C++ 11 library guarantee that states that calling only const functions of standard library objects from different threads is thread-safe.

The majority of physicist defined code utilizes the module component of the framework. As such, the thread safety requirements of modules will be the requirements that most affect physicists. The threaded framework uses four different varieties of modules, where each variety defines a different level of thread safety. The four module types are: Stream, Global, One and Legacy. The details of the thread safety requirements of each type of module is described in reference [3]. For the purposes of this paper, the work required of the collaboration was in converting modules from Legacy modules to Stream modules. In January, when we started, there were 20 conforming modules. In June we had 200 converted just in time for the first release of the multi-threaded framework. By the time of the ACAT conference we had 230, which is significant because this is the snapshot of the CMS software in which the performance measurements reported later were collected with.

3.1. Required changes for Stream modules

All modules in the multi-threaded framework are presumed to be thread unsafe and therefor are declared to the new frame work as Legacy modules. In order to convert them to a Stream module the author has to remove the use of non-const static member data, change the inheritance to `stream::Producer<>`, and remove any use of `begin/end _job` callbacks. If those are still needed they can be converted to `begin/end stream` callbacks, however this was rare.

3.2. Thread safety patches for 3rd party software

In order to successfully run the full CMS reconstruction application in the multi-threaded framework, we had to make sure that all of the 3rd party software that we use was thread-safe. We used git repositories forked off of the official repositories of products like ROOT, CLHEP and fastjet to make minimal changes to them that allowed us to use them in our thread-safe 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 defined a framework that gives users the ability to extend it's compile time checking by adding user defined 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 modules, and one that finds problematic data products. 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 `edm::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 where 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. 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 both the implementation of the threaded framework, externals, and CMS software.

5. Performance

5.1. Amdahl's Law

Amdahl's law is a model for the relationship between the expected speedup of parallelized implementations of an algorithm relative to the serial algorithm, under the assumption that the problem size remains the same when parallelized. This means that with small enough overheads within the framework execution engine, it should be possible to approximate how much of the code is executing in parallel given the observed utilization and the number of cores the framework has been configured to allow. If P is the parallel fraction and n is the number of cores then the theoretical average utilization is $= [n(1-P)+P]^{-1}$. See [5] for a derivation and further discussion of Amdahl's Law.

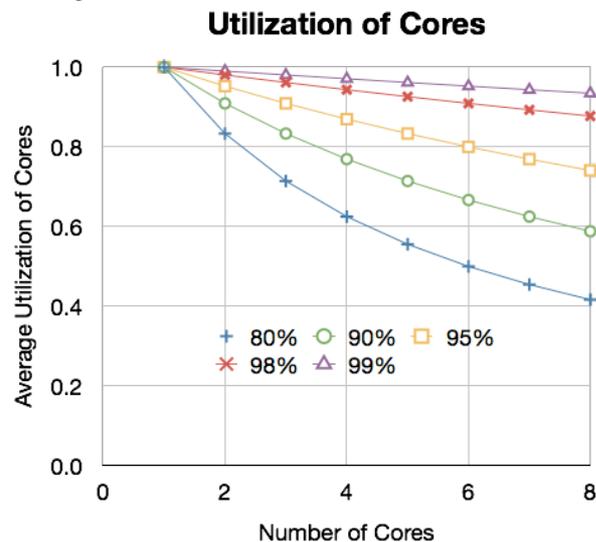


Figure 2. This figure plots the relationship between the theoretical average utilization of processing cores as a function of 2 variables: the number of allowed parallel tasks, and the fraction of the code that can be executed in parallel. The blue curve traces out this dependency when that fraction is 80%, green 90%, yellow 95%, red 98% and purple 99%.

In a multi-threaded CMS reconstruction application run on a simulation of the most challenging data taking conditions we expect in Run 2, we observe a 90% average utilization at 16 allowed parallel streams. This corresponds to a time weighted fraction of code which can be run in parallel, used in this application, of 99.3%. This highlights the high penalty for strictly serial code. Even if your profiler tells you that a function takes very little of the total time of your application, it can cause serious performance penalties if it requires serialization. A lot of effort went into finding serialization bottlenecks. We developed a log scanning tool that runs on the output of a tracer service that reports begin and end times of a modules execution as well as the stream that it was executing on. In this way it was possible to find gaps in time where the framework was forced to wait for some bit of serial code. A heuristic that worked well was to fix the module that was waiting to be executed. Where fix means make the module conform to the earlier described thread safety requirements and the convert the module to a stream module.

5.1 Throughput measurements

At the time of ACAT we ran our benchmark, described above, and obtained the following results presented in Figure 3. Note that there is good scaling up to 8 allowed streams on this 16 core, 64 GB AMD machine. At 8 cores, this application is 95% CPU efficient. Therefore running 2 processes in which each allowed 8 streams, CMS could utilize the full machine with an acceptable efficiency hit.

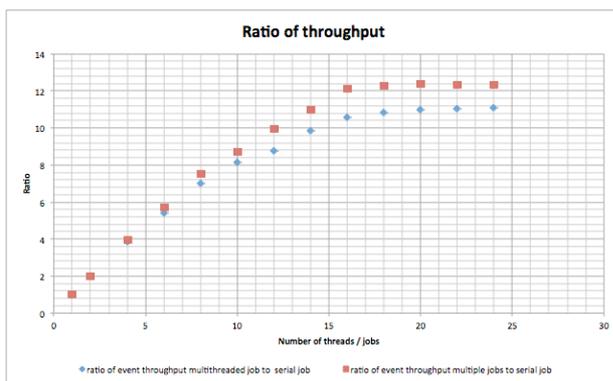


Figure 3. These two curves plot the ratio of throughput for multiple jobs to a serial job in red and a single multithreaded job to a serial job in blue

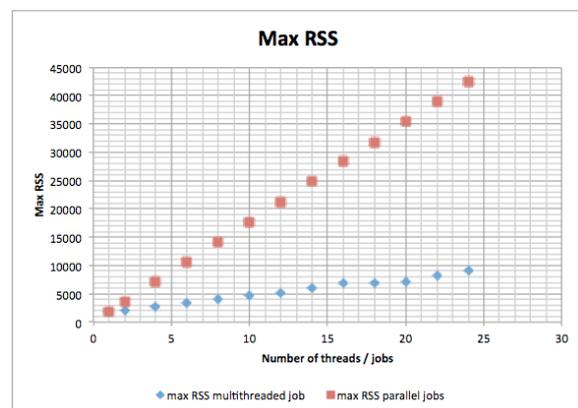


Figure 4. These two curves plot the maximum RSS used on the machine for the corresponding two cases in figure 3.

5.2. Memory, File Handle, and Network measurements

Figure 4 demonstrates that the big win is in the memory use of the multi-threaded application. At 8 cores it requires 4 GB of memory where as running 8 simultaneous jobs requires 14 GB. The number of network connections also scales linearly with the number of jobs, however in this case the network load of the multi-threaded application is the same as that for a single job of the serial application. A similar statement can also be made about open file handles.

6. 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 `edm::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. Over the past year CMS has managed to make 99.3% of its reconstruction code compliant enough to work efficiently in our multi-threaded framework (in other words only 0.7% of the code must be serialized by the framework). At the target value of an 8 core multi-threaded job, our reconstruction application has a CPU efficiency of 95%.

7.References

1. Sverre Jarp *et al* 2011 *J. Phys.: Conf. Ser.* **331** 052009, “Evaluating the scalability of HEP software and multi-core hardware” [doi:10.1088/1742-6596/331/5/052009](https://doi.org/10.1088/1742-6596/331/5/052009)
2. CMS Collaboration R. Adolphi *et al.*, “The CMS experiment at the CERN LHC”, JINST, 0803, S08004 (2008)
3. C D Jones and E Sexton-Kennedy 2014 *J. Phys.: Conf. Ser.* **513** 022034, “Stitched Together: Transitioning CMS to a Hierarchical Threaded Framework” [doi:10.1088/1742-6596/513/2/022034](https://doi.org/10.1088/1742-6596/513/2/022034)
4. <http://valgrind.org>, Jan. 25, 2014
5. Amdahl’s Law http://en.wikipedia.org/wiki/Amdahl's_law

Acknowledgments

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