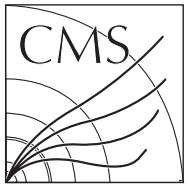
Available on CMS information server

CMS CR -2018/277



The Compact Muon Solenoid Experiment

Conference Report

Mailing address: CMS CERN, CH-1211 GENEVA 23, Switzerland

15 October 2018 (v2, 21 November 2018)

Implementing Concurrent Non-Event Transitions in CMS

Christopher Jones for the CMS Collaboration

Abstract

Since the beginning of the LHC Run 2 in 2016 the CMS data processing framework, CMSSW, has been running with multiple threads during production of data and simulation via the use of Intels Thread Building Blocks (TBB) library. The TBB library utilizes tasks as concurrent units of work. CMS used these tasks to allow both concurrent processing of events as well as concurrent running of modules processing the same event. This design has served CMS well and has allowed jobs to utilize less memory per core as well as reduce the number of jobs that must be tracked by CMSs worflow management system. As CMS has begun to utilize greater number of threads in a job, the effect of serialization points in the framework has decreased jobs CPU efficiency. One major serialization point occurs when the framework processes a non-Event transition. These transitions occur when a new Run or LuminosityBlock is to be processed. In this talk we will discuss how the different transitions define the processing model for CMSSW and how we were able to successfully allow concurrent processing of those transitions using TBB via task queues. We will also show CPU efficiency comparisons between the same work being done with and without the additional concurrency.

Presented at *CHEP 2018 Computing in High-Energy Physics 2018*

Implementing Concurrent Non-Event Transitions in CMS

Christopher Jones[1,*]

[1]Fermi National Accelerator Laboratory, Batavia, IL, USA

Abstract. Since the beginning of the LHC Run 2 in 2016 the CMS data processing framework, CMSSW, has been running with multiple threads during production of data and simulation via the use of Intel's Thread Building Blocks (TBB) library. The TBB library utilizes tasks as concurrent units of work. CMS used these tasks to allow both concurrent processing of events as well as concurrent running of modules processing the same event. This design has served CMS well and has allowed jobs to utilize less memory per core as well as reduce the number of jobs that must be tracked by CMS's workflow management system. As CMS has begun to utilize greater number of threads in a job, the effect of serialization points in the framework has decreased job's CPU efficiency. One major serialization point occurs when the framework processes a non-Event transition. These transitions occur when a new Run or LuminosityBlock is to be processed. In this paper we will discuss how the different transitions define the processing model for CMSSW and how we were able to successfully allow concurrent processing of those transitions using TBB via task queues. We will also show CPU efficiency comparisons between the same work being done with and without the additional concurrency.

1 Introduction

The CMS experiment at the LHC has used a multi-thread enabled data processing framework, CMSSW [1], for large scale data processing since the start of LHC Run 2 in 2016. CMS has successfully processed both data and Monte Carlo workflows using multiple threads. Initially, CMS only ran jobs using up to four threads. Over time, the threading efficiency of the CMS software was improved to the extent that CMS can now routine use eight or more threads for processing.

The CMS multi-threaded framework originally only supported two levels of concurrency. The first allowed concurrent processing of different events. The second allowed concurrent processing of different modules within an event. Both levels of concurrency can operate at the same time. There are additional transitions supported by the CMS framework beyond ones associated with events. Our goal is to allow all such transitions to be processed concurrently.

In this paper we will begin by describing the different data related transitions supported by the CMS framework. We will then explain how task queues can be used to constrain the resources used to support concurrency. We will end with performance measurements based on the changes made to support additional levels of concurrency.

* Corresponding author: cdj@fnal.gov

2 Data Processing Transitions

CMS used a strict hierarchical data organization. At the highest level are Runs. A Run represents a contiguous data taking session. Contained with Runs are LuminosityBlocks. A LuminosityBlock is a twenty-three second time period which is sufficient to measure the integrated luminosity during that time period. Finally, contained within LuminosityBlocks are Events. An Event is created when an 'interesting' collision occurred within the CMS detector causing the data acquired for the whole detector to be readout.

The CMS framework supports software callbacks triggered on transitioning from one data state to a different state. Specifically

- when a new Run is encountered, the framework issues a *beginRun* callback
- when a new LuminosityBlock is encountered, the framework issues a *beginLuminosityBlock* callback
- when a new Event is encountered, the framework issues an event callback
- when all Events within a LuminosityBlock have been processed, the framework issues an *endLuminosityBlock* callback
- when all LuminosityBlocks in a Run have been processed, the framework issues an *endRun* callback.

Originally the threaded framework only supported concurrent processing of Events within one LuminosityBlock. This meant that the framework would wait until the last Event in a LuminosityBlock had finished processing before it would move to the next LuminosityBlock and then allow concurrent processing of Events within the new LuminosityBlock. Such a synchronization point resulted in loss of efficiency. In this paper we will outline how we were able to extend the framework to allow concurrent processing of multiple LuminosityBlocks. This includes having multiple *beginLuminosityBlock* transitions occurring concurrently as well as concurrent processing of Events from different LuminosityBlocks. As of the writing of this paper, concurrent Runs are not yet supported as it is extremely rare for one production job to see more than one Run.

3 Shared Resources and Limited Task Queues

The primary motivation for CMS to move to a multi-threaded framework was to decrease the memory footprint of our production applications. By amortizing common data structures across threads, we were able to drop the average amount of memory needed per thread used by the application. One way the framework constrains memory is by being able to separate the number of threads used by the framework from the number of concurrent Events which are allowed to be processed. The memory needed to process an additional Event is greater than the memory used by adding an additional thread.

The number of threads as well as the number of allowed concurrent Events to process is controlled by the configuration passed to the application. In the most recent changes, the number of allowed concurrent LuminosityBlocks is also configurable. This allows complete control of the memory utilization needed per LuminosityBlock. The framework enforces the limitation on LuminosityBlocks via the use of a limited task queue.

The CMS framework uses Intel's Thread Building Block (TBB) library [2] to manage concurrency. All work done by the framework is broken into TBB tasks and then passed to

TBB to assign to available CPU cores. If two or more tasks need the same shared resource, e.g. need to write to the same output file or processing a LuminosityBlock, the tasks are placed into a queue rather than directly passed to TBB. Each unique resource is given its own queue within the framework. When a resource is available, the associated task queue will start to run one of its held tasks. The queue enforces that one and only one task using a given resource will be running at any given time.

If a chain of tasks all need the same resource (where a chain of tasks is the case were one task starts another group of tasks once the original task finishes) then the first task in the chain can pause the queue and then pass the remaining tasks directly to TBB. When the last task in a chain finishes, it will then resume the queue to allow the next task waiting in the queue to proceed.

For the LuminosityBlock case, we do not want to only allow one such resource to be used at a time, instead we want to specify a limited number and have that enforced. We accomplish this by using a limited task queue.

A limited task queue is implemented using multiple independent lanes where each lane is responsible for running its own task. All lanes pull tasks from the same queue of waiting tasks. The total number of allowed concurrent tasks from a limited task queue is equal to the number of lanes for the queue. Each lane can be paused and resumed independently. If all lanes are paused, no new tasks will be pulled from the shared queue. The CMS framework makes use of a limited task queue to constrain the number of LuminosityBlocks which can concurrently be processed.

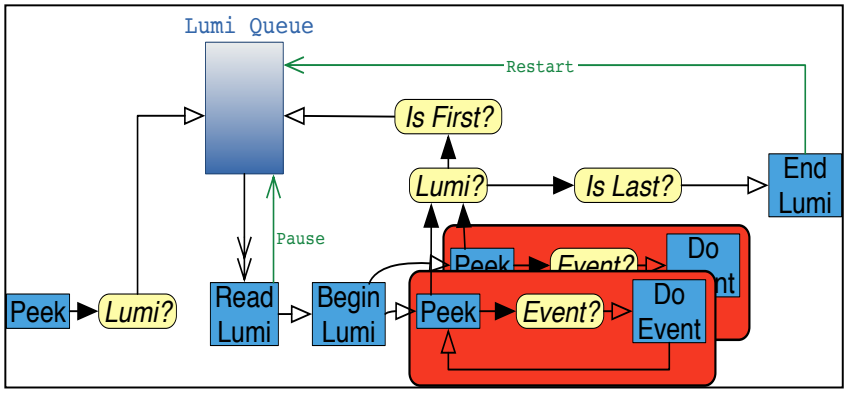


Fig. 1. Simplified diagram for how the framework processes data using tasks. Each solid blue square represents a task or chain of tasks. The yellow items are decision points made by the framework. The box labelled Lumi Queue is the limited task queue managing the processing of LuminosityBlocks and the contained Events. The black open arrow lines pointing to the Lumi Queue represent tasks being added to the queue. The double arrow out of the queue represents a task leaving the queue and being executed. The green arrows show tasks changing the state of the queue to either be paused or restarted.

Figure 1 outlines how a limited task queue is used to restricted the number of concurrent LuminosityBlocks being processed. In the example shown in the figure, the job has been configured to allow two concurrent Events but only one concurrent LuminosityBlock. Driving the processing loop is the Source which for this example has a LuminosityBlock containing two Events followed by a second LuminosityBlock containing no Events.

The job starts by peeking into the Source and seeing that the first transition is a LuminosityBlock. The framework then creates a LuminosityBlock processing task and inserts that task into the limited task queue for LuminosityBlocks (i.e. the Lumi queue). Because the Lumi queue is presently running and has no outstanding tasks, the LuminosityBlock processing task is immediately executed. This processing task pauses the Lumi queue and creates a new task to read the LuminosityBlock information from the Source. Once the reading task finishes, it creates an additional task which will run all the *beginLuminosityBlock* callbacks. Once all the callbacks have been completed, two Event processing tasks will be started.

Each Event processing tasks starts by peeking into the Source and seeing that the next entry is an Event. The necessary task chains are then started to process each Event independently. Once the last task for an Event has finished, it will start a new task that will again peek into the Source. In this case, the peeking task will see that the next Source entry is a LuminosityBlock. This task then sees it is the first task to peek at that entry in the Source so it creates a new LuminosityBlock processing task and puts it into the limited task queue. Given that the queue is presently paused, the LuminosityBlock processing task is not immediately started.

Once the final Event in the first LuminosityBlock finishes with all of its tasks, a new task is created to peek at the Source. It also sees that the next entry is a LuminosityBlock and see it is not the first task to peek. The task does determine it is the last task associated with the first LuminosityBlock so it creates the chain of tasks to call the *endLuminosityBlock* callbacks. Once all the callbacks finish, the final task resumes the Lumi queue which causes the waiting LuminosityBlock processing task to be executed, thereby continuing the cycle until the Source is empty.

4 Performance Measurements

We have made performance measurements to study the impact of allowing concurrent LuminosityBlock processing. The measurements record the event throughput (events per second) of a standard CMS reconstruction job. The job read an input file consisting of one Run containing eight LuminosityBlocks with each LuminosityBlock holding two hundred Events. The measurements were done using an Intel Knights Landing (KNL) machine with each job using sixty-four threads. Two different job variants were tried. The first only allowed one concurrent LuminosityBlock and the second allowed eight LuminosityBlocks. The upper limit of eight was chosen since that was the maximum number of LuminosityBlocks in the file being processed so was the maximum possible for the job.

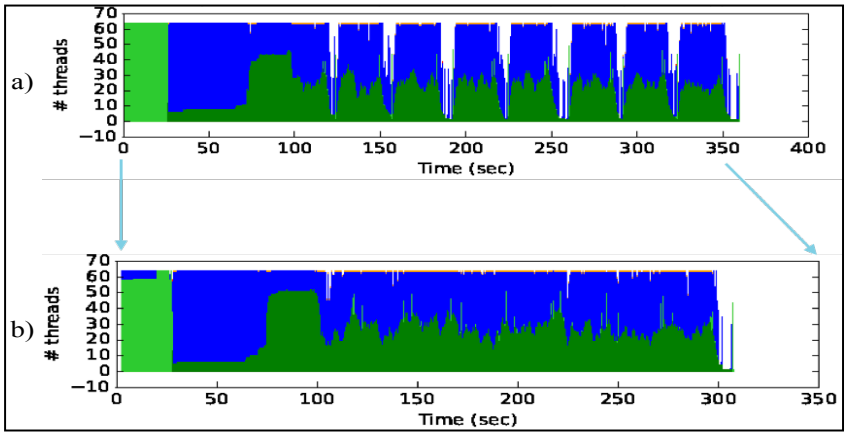


Fig. 2. Thread utilization over time comparison between a standard CMS reconstruction processing job allowing only a single concurrent LuminosityBlock (a) and one allowing eight concurrent LuminosityBlocks (b). The eight concurrent LuminosityBlock job shows much higher continuous thread utilization which results in a substantially shorter job processing time.

Figure 2 shows the thread occupancy of each job over the running time of the job. The superimposed light green histogram shows the number of concurrently running modules which are processing non-Event (i.e. Run or LuminosityBlock) transitions. The superimposed dark green histogram in the plots shows the number of concurrent events with at least one module actually running at that instance in time. As can be seen from the plots, although the allowed number of concurrent events is equal to the number of threads, cross event synchronization (i.e. modules which can only process one event at a time) do not allow all events to be processing data at the same time. Looking at the two plots, for the single LuminosityBlock job, we can see the effect of synchronizing on the LuminosityBlock boundaries which causes the thread utilization to drop. For the eight concurrent LuminosityBlock job one sees that the synchronization points are gone giving excellent thread utilization and an approximate 15% faster job.

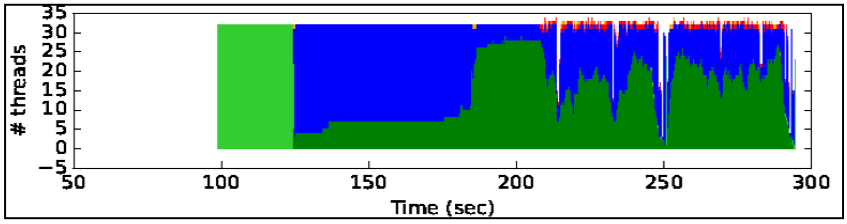


Fig. 3. Thread utilization over time for a standard CMS reconstruction processing job using four concurrent LuminosityBlocks and containing a single serial module. Notice the drop in thread utilization at each LuminosityBlock boundary.

As mentioned in the previous paragraph, the CMS framework supports the ability to run modules which can only be run by one thread at a time. To allow this, the framework serializes access to such *serial* modules. An additional complication arises if a serial module opts-in to the LuminosityBlock transition callbacks. When this happens, the serial module is expecting to see the *endLuminosityBlock* transition callback for the previous LuminosityBlock before the *beginLuminosityBlock* transition of the next one. Figure 3 shows the case where just one serial module opting-in to the LuminosityBlock transitions is

in the configuration, where the configuration allows concurrent LuminosityBlocks (the input file contains four LuminosityBlocks). Here we see that even with just one serial module, the framework is again forced to synchronize on the LuminosityBlock boundaries. The Events for the following LuminosityBlock are not processed until all the *beginLuminosityBlock* callbacks have finished and the *beginLuminosityBlock* callback for the serial module cannot happen until the previous LuminosityBlock's *endLuminosityBlock* callback for that serial module has finished. Some amount of concurrency is possible as non-serial modules can begin to run their *beginLuminosityBlock* callbacks before the serial module processes the previous LuminosityBlock's *endLuminosityBlock*. In general, the amount of work done at *beginLuminosityBlock* time is small and therefore provides limited concurrency potential.

3 Conclusion

CMS has successfully extended its multi-thread enabled framework to allow concurrent processing of Events across LuminosityBlock boundaries. This allows processing jobs to have higher average event throughput and allows more efficient processing of input data files containing few Events per LuminosityBlock. The number of allowed concurrent LuminosityBlocks is controlled via the use of a limited task queue. This was an extension of how CMS uses task queues to limit access to shared resources of any kind. Unfortunately, the practical benefits of the change are hampered by the existence of serial modules in CMS which opt-in to the LuminosityBlock transition callbacks. These modules effectively force the framework to synchronize on the LuminosityBlock boundaries again. These serial modules are being reviewed by CMS in order to minimize their negative consequences.

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

References

1. C.D. Jones and E. Sexton-Kennedy J. Phys.: Conf. Ser. **513** 022034 (2014)
2. https://www.threadingbuildingblocks.org