Available on CMS information server

CMS CR -2020/034

The Compact Muon Solenoid Experiment

Conference Report

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

04 February 2020 (v4, 06 August 2021)

Concurrent Conditions Access across Validity Intervals in CMSSW

Christopher Jones for the CMS Collaboration

Abstract

The CMS software system, known as CMSSW, has a generalized conditions, calibration, and geometry data products system called the EventSetup. The EventSetup caches results of reading or calculating data products based on the interval of validity, IOV, which is based on the time period for which that data product is appropriate. With the original single threaded CMSSW framework, updating only on an IOV boundary meant we only required memory for a single data product of a given type at any time during the program execution. In 2016 CMS transitioned to using a multi-threaded framework as a way to save on memory during processing. This was accomplished by amortizing the memory cost of EventSetup data products across multiple concurrent events. To initially accomplish that goal required synchronizing event processing across IOV boundaries, thereby decreasing the scalability of the system. In this presentation we will explain how we used limited concurrent task queues to allow concurrent IOVs while still being able to limit the memory utilized. In addition, we will present performance measurements for both threading scalability of event throughtput and memory utilization.

Presented at *CHEP2019 24th International Conference on Computing in High Energy and Nuclear Physics*

Concurrent conditions access across validity intervals in CMSSW

David Dagenhart[1,*], and *Christopher* Jones[1]

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

Abstract. The CMS software system, known as CMSSW, has a generalized conditions, calibration, and geometry data products system called the EventSetup. The EventSetup caches results of reading or calculating data products based on the 'interval of validity', IOV, which is based on the time period for which that data product is appropriate. With the original single threaded CMSSW framework, updating only on an IOV boundary meant we only required memory for a single data product of a given type at any time during the program execution. In 2016 CMS transitioned to using a multi-threaded framework as a way to save on memory during processing. This was accomplished by amortizing the memory cost of EventSetup data products across multiple concurrent events. To initially accomplish that goal required synchronizing event processing across IOV boundaries, thereby decreasing the scalability of the system. In this presentation we will explain how we used 'limited concurrent task queues' to allow concurrent IOVs while still being able to limit the memory utilized.

1 Introduction

In 2015, the CMS Experiment became the first LHC experiment to begin using a multi-threaded framework for event processing. This came after multiple years of preparation. Over the succeeding years CMS has gradually improved both the level of concurrency in its multi-threaded processes and the efficiency of those processes. The software enabling this capability is concentrated in the framework, but significant changes were needed throughout the entire CMS software system (CMSSW). Much of the required work has been outside the framework in algorithmic modules. Concurrency is implemented using Intel's Threading Building Blocks library (TBB) [1]. Concurrent processing of events is supported; concurrent processing of modules within an event is supported; and modules can internally start tasks that are processed concurrently [2, 3]. The use of tools to help find issues related to multi-threading in CMSSW has been critical [2, 3]. CMS performed static code checks using the Clang [4] tool suite, which were extended beyond the default set of checks provided by the tool. Helgrind [5] was used to look for possible race conditions.

In 2015 CMS performed tests of its multi-threaded capability in a production environment and reported the results [6]. The primary benefit of using a multi-threaded process over using many independent single threaded processes is reduced per core memory requirements. Threads share the memory needed to store data not associated with specific events. This was critical for CMS to be able to make good use of CPU resources

[*] Corresponding author: wdd@fnal.gov

currently available and affects the costs associated with future CPU purchases. There were other benefits. CMS uses a workflow management system to coordinate running a large number of applications on many grid sites spread throughout the world. Because a multi-threaded job can do more, fewer jobs are needed. This reduces database requests and file opens among other things. Fewer jobs reduce the pressure on the workflow management system and the many sites processing data.

CMS has been making continual improvements to its multi-threading capability. In 2017 CMS reported about its progress, which included changes that allowed CMS to increase the number of threads above 4, that allowed using multiple threads in digitization and simulation processes in addition to reconstruction processes, and that improved efficiency in multi-threaded processes [7]. CMS reported on efforts to identify algorithmic modules impairing multi-threading efficiency. There were also improvements to the framework. CMSSW now schedules modules that produce data objects before the modules that need that data are spawned to the TBB scheduling system. Serial task queues aid in the scheduling of tasks that required resources that cannot be executed concurrently. CMSSW now schedules concurrent execution of filter sequences.

In CMS, an event is data associated with a particular interesting collision in the detector. Events are grouped in luminosity blocks. A luminosity block corresponds to the events recorded in a contiguous 23-second time period. In the original multi-threaded framework, events from different luminosity blocks could not be processed concurrently. In 2018, CMS implemented changes that allowed processing multiple luminosity blocks concurrently [8].

In this paper, CMS reports further improvements that increase the concurrency of CMS multi-threaded processes.

2 Intervals of validity

The CMS Experiment uses different kinds of data as it reconstructs the particles and collisions from the data recorded in its detectors and simulates these particles and collisions. Some of the data are associated with particular events or luminosity blocks, but there are other data that CMS calls **conditions** data. Conditions data include calibration data, alignment data, and geometry data. Conditions data are associated with a time interval over which it is valid and can be used. CMS calls these **intervals of validity** or **IOV**s.

Typically conditions data are valid for many events. It might be valid for one or many luminosity blocks. It is often read from a database or calculated from data read from a database. Then it is stored in a cache in memory. This cache takes a significant amount of time to fill and uses a significant amount of memory, so it is shared and used by multiple threads and multiple events in a multi-threaded process.

There are many different types of conditions data. For example, one type might specify the values related to the average noise to be subtracted from data for each detector element in the hadronic calorimeter. Another type might contain the location of the particle beams in the detector. Another type might contain geometric constants. The differing types are valid for different time periods. Geometric constants might be valid for years. The beam position might be updated in every luminosity block. Different calibrations get updated after different intervals of time. Types with the same IOVs are grouped together in what CMS calls a **record**. CMS has defined hundreds of different record types. As data are processed there will be transitions where the IOV of each record type changes.

CMS calls the software system it uses to manage conditions data and IOVs the **EventSetup** system.

Prior to 2019, CMS could not concurrently process data from different IOVs. So this meant that when a process encountered an event associated with a different IOV for any record, the process had to wait until all processing was complete for all events associated with all IOVs. The framework had to wait to synchronize processing of IOVs. During this wait, there is often not enough work to keep all threads busy. Threads end up sitting idle and this reduces the CPU efficiency of CMS processes.

In 2019, CMS implemented improvements in its framework that add support for processing multiple IOVs concurrently. If configured, this will eliminate the wait at IOV boundaries and increase CPU efficiency for CMS processes.

3 Configurable by record

The number of IOVs allowed to run at the same time can be independently configured for each record. This is important for three reasons.

First, the amount of memory associated with a record varies widely. Some records use an amount of memory that is negligible compared to the total memory used by a process. Other records use a very significant amount of memory. When IOVs are run concurrently, the data for every active IOV must be cached in memory at the same time. CMS can reduce memory requirements by configuring records using large amounts of memory to allow one IOV at a time.

Second, as already mentioned, the length of an IOV varies from one record to the next. Performance improvements related to concurrent IOVs are greater for those records whose IOVs change more often.

Third, it is possible some software associated with some records does not support concurrent IOVs. Being able to configure records independently allows records that do not support concurrent IOVs to be used in jobs processing IOVs concurrently for other records.

4 Inefficiency at IOV boundaries

CMS tested the new ability to process multiple IOVs concurrently by running reconstruction on data and plotting the number of threads running at each moment of time. We chose data from 2018 that consisted of 151 events with a known IOV transition near the middle of the input. We also included special code that monitors the number of threads executing modules as a function of time. The job was configured to run 16 threads concurrently, 16 events concurrently, and 2 luminosity blocks concurrently. The job was run twice. The first time only 1 IOV was allowed to run at a time. The second time the job was configured to allow 2 IOVs run concurrently. The results are shown in the figure below.

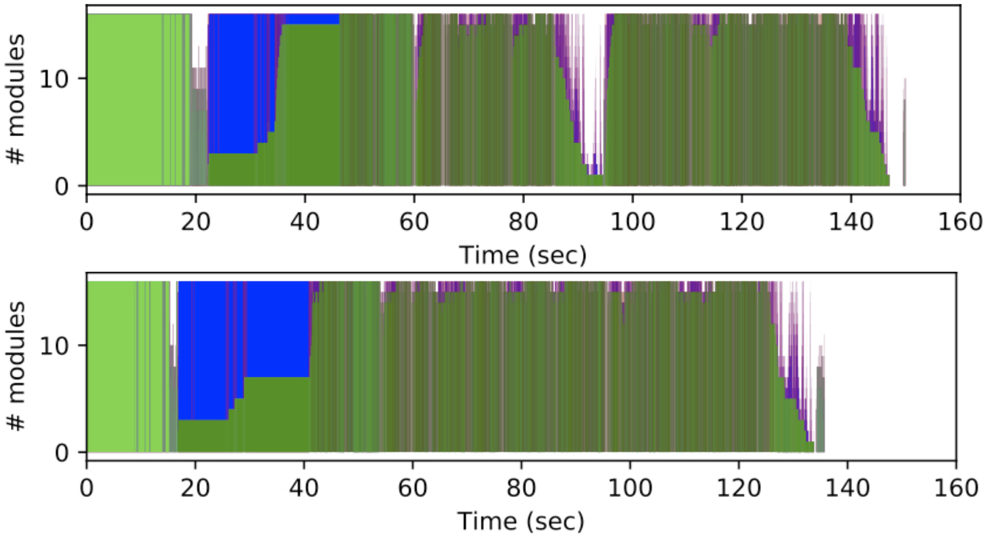

Fig. 1. Number of modules executing concurrently as a function of time in a reconstruction process using data from 2018 as input. In the top plot, the process is configured to allow only 1 IOV to be processed at a time. The only difference in the lower plot is that the process was configured to allow 2 IOVs to be processed concurrently. Section 4 is devoted to discussing this figure and includes more details describing it.

In the top plot, a gap with white space is visible between 80 and 100 seconds. This white space indicates threads that are idle and not performing work. In the bottom plot, all

16 threads are busy executing modules during this time. This graphically indicates the CPU inefficiency related to IOV boundaries. The main point of the figure is to show this inefficiency.

One might want to reduce this to a single number indicating the overall improvement in CPU efficiency that results from this new capability in the framework, but this is difficult to generate. The overall efficiency gain becomes more significant as the number of threads increases. It becomes less significant as the number of threads decreases. It also significantly depends on how frequently IOV boundaries are crossed within a processing job. As the frequency of IOV transitions increases, the degradation of efficiency increases. If one studies production jobs prior to 2019, then one would find the effect to be small. In the past, many production jobs run by CMS included only 1 IOV. The improvement is targeted more to the future. There are development efforts in CMS in progress that will result in IOV boundaries occurring more often in CMS production jobs. We also expect the number of concurrent threads to continue to increase. The CMS framework is now ready to handle these expected environments. This efficiency improvement will be more important in the future than it would have been in the past.

The type of plot shown in Figure 1 has been useful to CMS and is described in more detail in [8]. For this paper, the most important feature of the plot is the total area that is colored (or shaded). The work that needs to be done to process an event is divided into logical pieces and the software related to a particular piece is called a module. For example, one module might execute one algorithm to reconstruct track segments in one part of the CMS detector. Hundreds of modules might execute on each event in a typical CMS reconstruction process. In Figure 1, the number of modules is almost the same as the number of threads that are busy doing work. The darker green counts the first module working on each event. Blue counts modules beyond the first and appears when multiple threads are working on the same event. Purple and the varying darkness of the green is really an artefact related to blue and green being plotted too close together to distinguish. The initialization period before approximately 40 seconds is irrelevant to this paper. It looks overly significant in the plot because of the large number of threads and the small number of events. CMS is currently working on improving concurrency in initialization.

5 Limited concurrent task queues

The ability to process IOVs concurrently is implemented with a class defined in CMSSW called LimitedTaskQueue that uses TBB tasks. This section starts by discussing how the CMSSW framework uses TBB in general. Then it continues by explaining how the limited task queue uses TBB and how the framework uses the limited task queue to implement concurrent IOVs.

5.1 Threading Building Blocks (TBB)

CMSSW relies on the TBB library from Intel as the base on which it implements its concurrency capabilities. The CMSSW framework makes heavy use of the tbb::task class. CMSSW classes derived from this contain functions that perform almost all the concurrent work done in CMSSW. TBB has a scheduler that executes the tasks. The TBB scheduler has configurable limits on how many tasks will execute concurrently and algorithms that determine the order of execution of tasks. CMSSW takes advantage of the TBB scheduler. Although not described here (see [1]), TBB's scheduler is important to CMSSW.

A task is **spawned** when it is passed to the TBB scheduler using a TBB function named spawn. From this point forward, TBB controls when the task will be executed. It will be executed as soon as possible given the limits on the number of concurrent tasks and TBB's scheduling algorithms. Much less often, CMSSW will use the enqueue function of TBB, which also submits tasks to TBB, but these are scheduled differently.

The framework also makes use of TBB functions to create tasks, destroy tasks, attach tasks, isolate tasks, manage reference counts, and wait for tasks to complete. Often the framework uses the reference count to determine when a task should stop waiting and be spawned. The framework uses task_arena in limited circumstances. CMSSW uses TBB's concurrent containers in many contexts.

The CMSSW framework does not use the higher level functionality available in TBB. For example, TBB's parallel_for loop is not used within the framework. Although it is used in CMSSW algorithmic code and works well with the framework in that context.

The CMSSW framework does a lot of work outside of TBB to understand the dependencies between tasks and to understand which tasks cannot be run concurrently. The framework does not spawn tasks to TBB when the task cannot run because it needs data that is not available yet or when other tasks have been spawned and have not yet finished that cannot run concurrently. This paper is focused on one particular case where the framework uses TBB to control how many IOVs can be processed concurrently. This is done through a CMSSW class called LimitedTaskQueue.

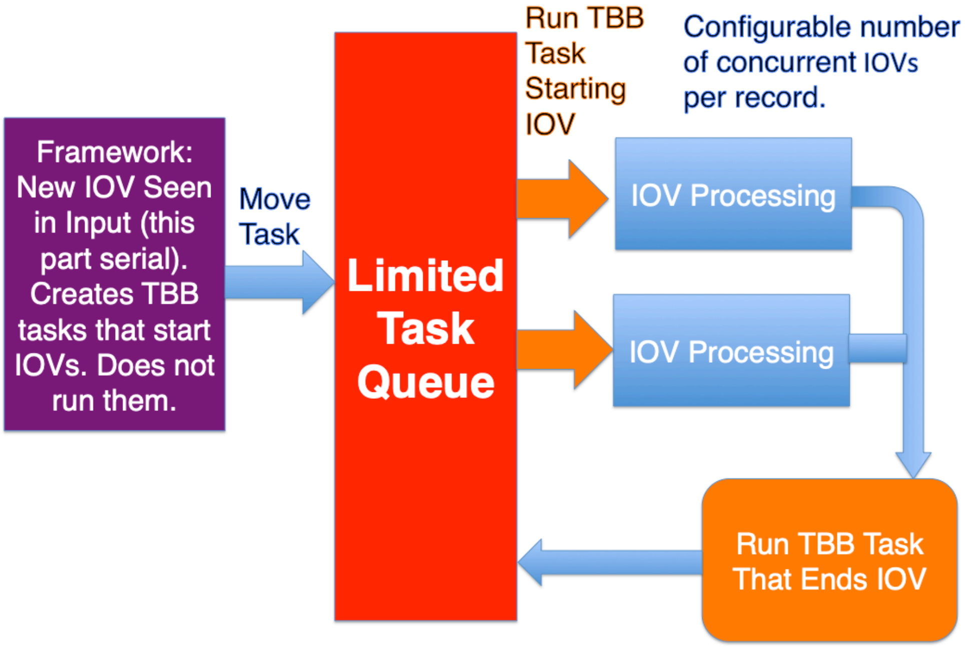
Framework: New IOV Seen in Input (this part serial). Creates TBB tasks that start IOVs. Does not run them.
Move Task
Limited Task Queue
Run TBB Task Starting IOV
Configurable number of concurrent IOVs per record.
IOV Processing
IOV Processing
Run TBB Task That Ends IOV

Fig. 2. A limited task queue is used to implement concurrent IOV processing.

5.2 Concurrent IOVs and limited task queue

The CMSSW framework sequentially reads through the luminosity blocks in an input file. When it encounters a new luminosity block, it creates a task that will start processing for that luminosity block. The task is not immediately executed nor is it immediately spawned to TBB. It is constructed, ready, and waiting. Note that the framework will not read the next luminosity block from the input file until this task has started. The recursion in the sequence of tasks, which includes this task and the many tasks involved in processing a luminosity block, drives the iteration over luminosity blocks.

After the task that starts a luminosity block has been created and while it is waiting, the framework loops over all the records (different types of conditions data). For each record, it determines if the new luminosity block is within the current interval of validity (IOV). If it is not in the current IOV or the first IOV needs to be started for a record, then a task is created that will start an IOV for that record. Again, this task is not immediately executed nor is it immediately spawned to TBB. It is constructed, ready, and waiting. The task that starts the IOV is given a pointer to the task that starts the luminosity block. When the task that starts the IOV finishes the work needed to start the IOV and the IOV has begun, it notifies the task that starts the luminosity block. The task that starts the luminosity block maintains a reference count of IOVs that need to start before it can be run. When that reference count indicates all required IOVs have started, the task starting the luminosity block is spawned to TBB.

Figure 2 depicts the creation of the task that starts the IOV and the actions described below.

For each record there is a limited task queue. The task to start the new IOV is passed to the limited task queue associated with its record. Each limited task queue is configured with a positive integer limit restricting the number of IOVs that can be active at the same time. If the number of active IOVs for that record is less than the limit, then the task that starts the IOV is immediately spawned to TBB. Otherwise, the task is saved in the queue and waits. Note that this task simply does the work to start the IOV. When the task completes execution, the IOV is active and might be in use for a long time while luminosity blocks within that IOV are processed. The IOV does not end when the task that starts it finishes.

The task that starts an IOV creates yet another task that will end the IOV. Again this task is constructed, ready, and waiting, but it is not immediately executed or spawned. It also holds a reference count that is incremented and decremented to determine when it should be spawned. The most recent IOV for each record is held open by incrementing the counter and it is decremented when the next IOV for that record is opened. The counter is also incremented for every luminosity block that is using it. Each luminosity block is passed a pointer to the tasks that end all of the IOVs that they are using. When each luminosity block ends, the pointers are used to decrement the reference counts. The task that ends an IOV will be spawned when all luminosity blocks using it have completed and it is no longer the most recent IOV for its record or at the end of the job.

One action that the task that ends an IOV performs is to notify the limited task queue that the IOV is finished and then the limited task queue is able to spawn the task that starts the next IOV if there is a task waiting in its queue.

6 Conclusion

CMS has improved its software framework so that it is now capable of running multiple intervals of validity concurrently. This will help CMS to maintain and improve the CPU efficiency of its production software as the CMS computing environment changes to require more threads and shorter intervals of validity.

This document was prepared by the CMS Collaboration using the resources of the Fermi National Accelerator Laboratory (Fermilab), a U.S. Department of Energy, Office of Science, HEP User Facility. Fermilab is managed by Fermi Research Alliance, LLC (FRA), acting under Contract No. DE-AC02-07CH11359.

References

1. https://www.threadingbuildingblocks.org
2. C.D. Jones, E. Sexton-Kennedy, J. Phys.: Conf. Ser. **513** 022034 (2014)
3. E. Sexton-Kennedy, P. Gartung, C.D. Jones, D. Lange, J.Phys.: Conf. Ser. **608** 012034 (2015)
4. http://clang.llvm.org
5. http://valgrind.org
6. C.D. Jones, L. Contreras, P. Gartung, D. Hufnagel, E. Sexton-Kennedy, J. Phys.: Conf. Ser. **664** 072026 (2015)
7. C.D. Jones, J. Phys.: Conf. Ser. **898** 042008 (2017)
8. C.D. Jones, CERN Document Server: CR-2018/277 (2018)