CMS event processing multi-core efficiency status

CMS event processing multi-core efficiency status

C D Jones on behalf of the CMS Collaboration
Fermilab, P.O.Box 500, Batavia, IL 60510-5011, USA

E-mail: cdj@fnal.gov

Abstract. In 2015, CMS was the first LHC experiment to begin using a multi-threaded framework for doing event processing. This new framework utilizes Intel's Thread Building Block library to manage concurrency via a task based processing model. During the 2015 LHC run period, CMS only ran reconstruction jobs using multiple threads because only those jobs were sufficiently thread efficient. Recent work now allows simulation and digitization to be thread efficient. In addition, during 2015 the multi-threaded framework could run events in parallel but could only use one thread per event. Work done in 2016 now allows multiple threads to be used while processing one event. In this presentation we will show how these recent changes have improved CMS's overall threading and memory efficiency and we will discuss work to be done to further increase those efficiencies.

1. Introduction
Since the beginning of the LHC Run 2 era, CMS has been using multiple threads in the offline data processing framework [1] when doing production level work. The transition to a multi-threaded framework was to allow better utilization of the resource-constrained Grid site infrastructure, in particular to use sites with limit CPU memory/core as LHC Run 2 processing requires greater memory use per event. CMS has slowly expanded the use of threading into the standard data processing workflows. The first use was prompt reconstruction of data at the CERN Tier 0 site which used four threads [2]. The online high level trigger farm switched to also using multiple threads per job in September of 2015. Reconstruction of Monte Carlo events began using four threads for standard processing in the Summer of 2016. The simulation itself began using four threads during the Winter of 2016.

The implementation of the system up until Fall of 2016 only supported having one thread per concurrent event. In this paper this implementation will be referred to as ***original***.

The decision to only use four threads per job for offline processing was driven by the following concerns. Given the original implementation and the amount of code which had been validated as threads safe, four threads gave sufficient CPU efficiency at the time. In addition, using four threads was sufficient for staying within the memory limits of the worker nodes. Finally, additional threads were not needed to keep the number of simultaneous jobs controlled by the workflow management system to a workable limit.

In this paper we will explain the work of the CMS collaboration to increase the efficiency of the framework when running with more than four threads, in particular how to mitigate the effect of processing stalls. Additionally, we will present measurements of the efficiency using the full CMS reconstruction code.

2. Mitigating stalls

We use the term ***stall*** to refer to the case where the framework still has work to be done but at least one processing thread is idle or blocked by a lock. Such stalls are caused by sharing resources concurrently across running threads. For the original implementation, there was an event per thread so the shared resource was being used by more than one event. Several examples of activities that cause stalls are:

- reads from one ROOT [3] file must be serialized;
- writing to one ROOT file must be serialized (although it is possible to write to different ROOT files simultaneously);
- ***legacy*** modules are modules are ones which were developed during LHC Run 1 and are not designated as being thread-safe so the framework will only run one such module at a time.

One major reason for stalls in the original implementation was the modules running order for a given event was fixed. This meant that if another thread (i.e. another event) was using a shared resource, the thread would have to stall until that resource became available. The original implementation had no way to reschedule modules dynamically.

2.1. Stall demonstration

The following simple configuration can be used to demonstrate stalls:

- read event data from an input file;
- use two legacy modules which cannot run simultaneously;
- use five additional modules which are thread-safe.

Data dependencies between modules constrain the allowed concurrency since modules wait to run until all the data they will use has been made available from other modules or from the input file. This configuration is shown pictorially in figure 1.

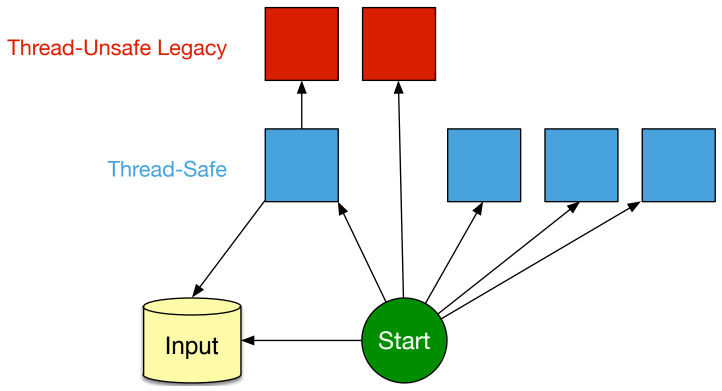

Figure 1. A pictorial representation of the configuration used to demonstrate stalls. The cylinder represents obtaining data from the input file (which must be done serially). The circle represents the module which is on a sequence and is what starts the processing chain. The squares represent modules which are started when their data is needed. The arrows between modules represent data dependencies.

Figure 2 graphically displays the time progression of an execution of the demonstration configuration. In this case, the framework is using four concurrent event loops. Each independent loop is referred to as a ***stream***. For the original implementation, each stream can only make use of one thread. The green bands on the plot are when a module is running in the stream. The white is when no module is running in that stream. The red band is when a 'stalled' module, one that stalled at some point in the job, is being run. A 'stalled' module is shown in red even if it was not part of a stall at that particular time in the stream. One can see that a white stall is followed by a red module. In addition, a white band on one stream corresponds to a time on another stream when a red module is running. This demonstrates that a module stalls because it cannot run concurrently while another stream is running the same module.

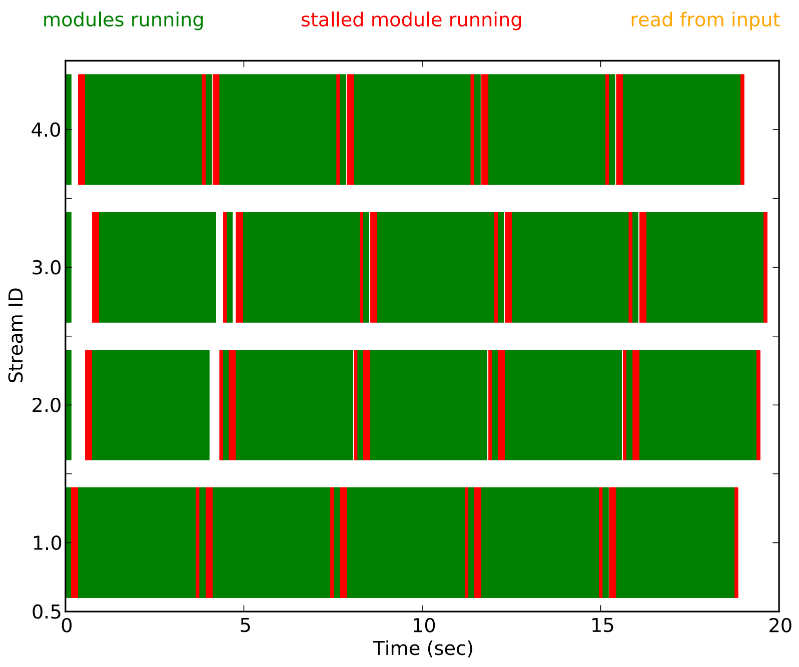

Figure 2. A visualization of the work being done by each stream for the original framework implementation running the stall demonstration configuration. Green shows when a module is running on the stream. Red is shown when a module that has stalled at least once in the job is running on the stream. White is used when no module is running.

2.2. Stall mitigation techniques

The new implementation of the CMS threaded framework uses three techniques to mitigate the efficiency loss from stalls: concurrent filter sequences, concurrent prefetching and serial task queues. All of these are meant to provide the framework with a list of modules which could be run at a particular moment (because their dependent data is available) which gives alternatives to modules which would otherwise stall. As CMS uses Intel's Thread Building Block [4] (TBB) library to handle concurrency, the list of modules is handled via spawning many TBB tasks, one task for each module.

One element of a CMS job configuration is sequences of modules which filter events. These filter modules inspect a given event and then decide if the event should be kept or rejected. If an event is rejected by a filter, all modules which appear later on the same sequence as that filter will not be run. A CMS configuration may contain multiple such filter sequences which are independent of each other. The new framework implementation will schedule to run the first filter module in all sequences concurrently when a new event is being processed. This allows sequence processing to happen concurrently.

In addition to filtering modules on a filter sequence, modules can also be run 'on demand' when the event data products they create are needed. Given that a module may need multiple data products from the event, the new framework implementation schedules all the prefetching of those data products concurrently. That is it creates and spawns a TBB task per prefetching of data products. This can create a very large number of tasks for TBB to schedule which greatly aids concurrency.

The final mitigating technique is to use a serial task queue to protect a shared resource. A module which is using a shared resource, such as a module which writes out all events to a file, is run via a TBB task. However, instead of having that TBB task spawned directly, the task is placed in a serial task queue which is specific for the shared resource. The serial task queue guarantees that there is one and only one task from the serial task queue being spawned into TBB's running task system. When a task originating from a serial task queue is done executing, its last step is to signal back to the serial task queue and cause the next task in the queue to be spawned.

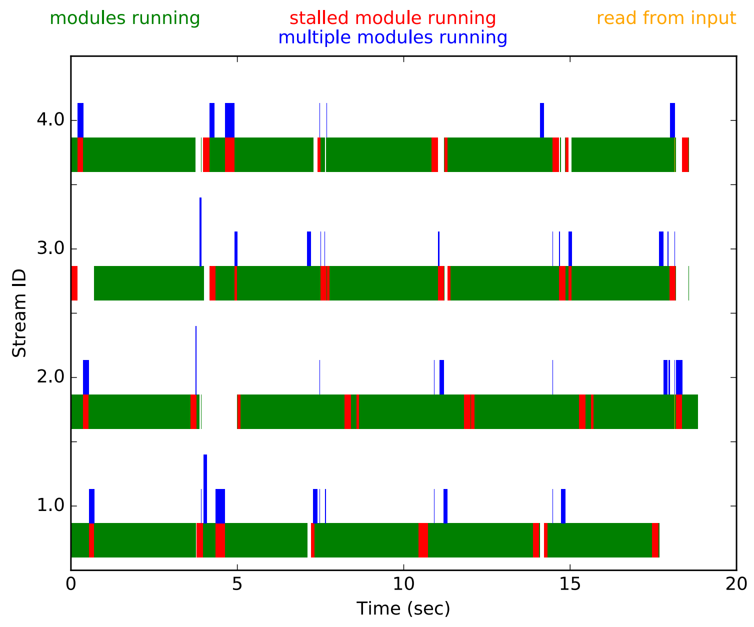

Figure 3. Stall measurements for new framework implementation using the stall demonstration configuration. This run uses only four threads with four streams.

Figure 3 shows the time progression of an execution of the demonstration configuration using the new framework implementation and four threads. The blue bars on this graph represent the time a stream is running more than one module, where the height of the bar is proportional to the number of extra modules running. Given that there are four streams and four threads, we see that a blue bar in

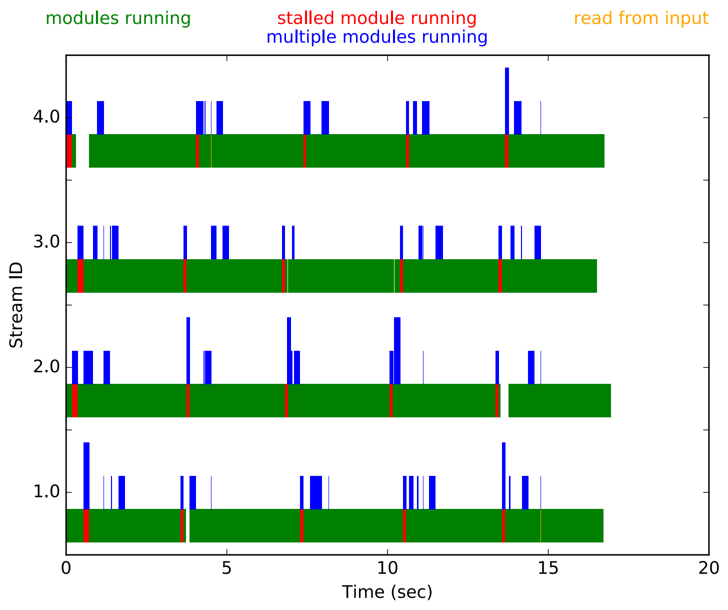

Figure 4. Stall measurements for new framework implementation using the stall demonstration configuration. This run uses five threads with four streams.

one stream corresponds to a white bar (that is no running modules) on another stream. Looking at the times when a red module has stalled (that is a red bar is preceded by a white bar) we can see they correspond to times where another stream is using multiple threads. This shows that the framework has successfully scheduled work during the times that the original implementation would have stalled.

Figure 4 is another execution of the demonstration configuration using the new implementation but this time using five threads with four streams. The extent of the stream stall times (shown by white bars) is much less than in Figure 3 and the job finishes in less time which shows that the new implementation is able to exploit the extra thread, although it is still limited by the available intra-event concurrency of the configuration.

3. Realistic measurements

We also made measurements of running CMS reconstruction jobs on both the original and the new implementation. The measurements were made on a machine with an Intel Westmere-EP [5] L5650 CPU with six cores and two hyper-threads per core. The reconstruction configuration is made up of three 'output modules' (modules which write event data to files), 1,780 other modules and 21 filter sequences.

Three different groups of measurements were made:

- using the original implementation with one-thread-per-stream;
- using the new implementation and keeping the number of threads equal to the number of streams;
- using the new implementation and setting the number of threads to 12.

Figure 5 shows a comparison of event throughput as a function of number of streams for the three different cases. From this we see that the new implementation always offers greater throughput than the original implementation. When the new implementation uses the same number of threads as the original implementation it is able to have between 5% and 10% greater event throughput. The new implementation is able to increase a jobs event throughput by using more threads than streams as demonstrated by the fact that the new implementation using twelve threads always has the greatest throughput of all three cases. Investigations of stalls in the new implementation showed that the stalls were only caused by one output module when the output module was writing data to the disk and the time it took was greater than the average time to process one event. The new implementation was able to process all other tasks for an event until the only task left was to run the stalled output module.

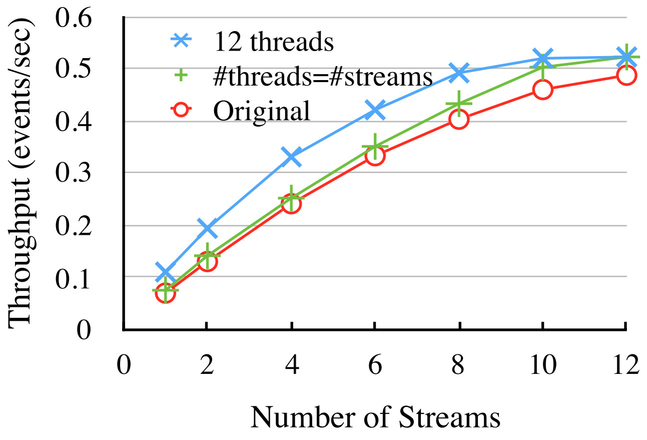

Figure 5. Event throughput versus number of streams for the three realistic measurements.

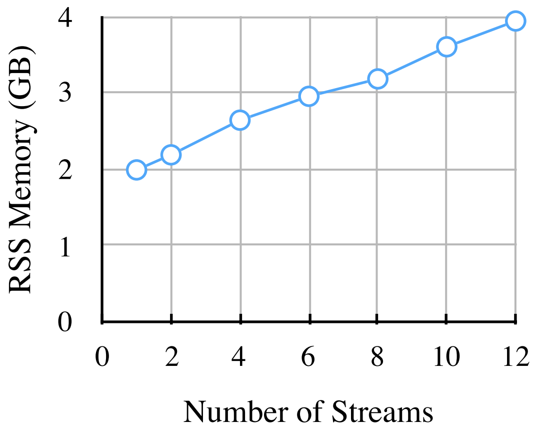

Figure 6. Resident memory versus number of streams for the realistic measurements.

Figure 6 shows the memory usage as a function of the number of streams. With just one stream, the initial memory needed is fairly high at about 2GBs. However, the memory requirement grows slowly as the number of streams increased: about 150 MB/stream. If the number of threads is increased instead of the number of streams, there is no appreciable memory change.

With appropriate choices on the number of streams and threads used by one job as well as the number of jobs to run on one computing node, it is possible to maximize the total event throughput while staying within a set memory per core limit. This is shown in figure 7, where seven different configurations are used but all configurations are constrained to have (number of processes) * (number of threads/process) = 12. From the figure we see that we can choose configurations using 2 GB/core, 1 GB/core or even 0.5 GB/core and still maintain reasonable event throughput.

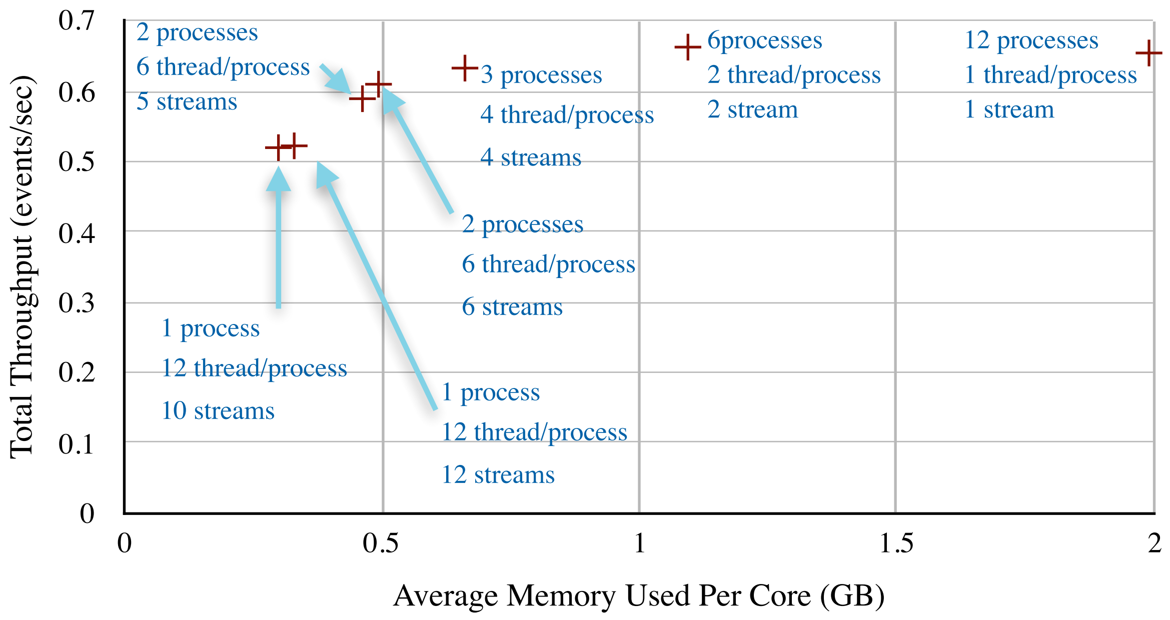

Figure 7. Total event throughput versus average memory used per core, running the realistic configuration for different numbers of processes, threads and streams per process (the values used are given by the blue labels).

4. Conclusion

CMS has successfully utilized multi-threaded processing jobs starting with the LHC Run 2 era. All of prompt reconstruction run by CMS for Run 2 were run using four threads per process. The transition to a multi-threaded framework was to allow better utilization of the resource-constrained Grid site infrastructure, in particular CPU memory. This change allows CMS to make full use of the CPU cores while processing more memory intensive jobs and to utilize machines with lower memory per core limits.

Future threading efficiency is important to CMS as we move towards utilizing more threads per job. To that end, the new framework implementation makes better use of available cores by scheduling lots of tasks while avoiding co-scheduling tasks which share the same non-concurrent resource.

Acknowledgments

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

References

[1] Jones C D and Sexton-Kennedy E 2014 *J. Phys.: Conf. Ser.* **513** 022034
[2] Jones C D *et al* 2015 *J. Phys.: Conf. Ser.* **664** 072026

[3] http://root.cern.ch/root/
[4] https://www.threadingbuildingblocks.org
[5] http://ark.intel.com/products/codename/33174/Westmere-EP