IOPscience iopscience.iop.org

Home Search Collections Journals About Contact us My IOPscience

Using the CMS Threaded Framework In A Production Environment

This content has been downloaded from IOPscience. Please scroll down to see the full text.

2015 J. Phys.: Conf. Ser. 664 072026

(http://iopscience.iop.org/1742-6596/664/7/072026)

View the table of contents for this issue, or go to the journal homepage for more

Download details:

IP Address: 188.184.3.56
This content was downloaded on 08/03/2016 at 22:14

Please note that terms and conditions apply.

Using the CMS Threaded Framework In A Production Environment

C D Jones[1]**, L Contreras, P Gartung, D Hufnagel and L Sexton-Kennedy on behalf of the CMS Collaboration**
Fermilab, P.O.Box 500, Batavia, IL 60510-5011, USA

E-mail: cdj@fnal.gov

Abstract. During 2014, the CMS Offline and Computing Organization completed the necessary changes to use the CMS threaded framework in the full production environment. We will briefly discuss the design of the CMS Threaded Framework, in particular how the design affects scaling performance. We will then cover the effort involved in getting both the CMSSW application software and the workflow management system ready for using multiple threads for production. Finally, we will present metrics on the performance of the application and workflow system as well as the difficulties which were uncovered. We will end with CMS' plans for using the threaded framework to do production for LHC Run 2.

1. Introduction

The CMS experiment has moved to using multi-threading in its application in order to achieve goals for both the data processing application and the workflow management system.

The primary motivation for using multiple threads within CMS' data processing application is to reduce the amount of memory used per CPU core. With the recent upgrades to the LHC accelerator, the complexity of the events recorded at CMS are expected to drive CMS' single-threaded application to need more than 2GB of resident memory when processing such events. However, by processing multiple events in parallel by using multi-threads the sharing of non-event related data reduces the per CPU core memory requirement to be well below 2GB.

The workflow management system coordinates the running of the CMS data processing application over grid sites spread throughout the world. The use of a multi-threaded application aids the workflow management system in several ways. First, by running fewer applications on a grid site it means fewer database requests and fewer open files thereby reducing pressure on that site. Second, using a multi-threaded application allows us to reduce the time it takes to process one event *block*. CMS groups events into *blocks* where one block is 23 seconds of data taking. We require that all events within one block must be successfully processed by only one job because CMS does luminosity accounting on the granularity of a block. Therefore the faster we can process one block the shorter the time it takes to finish a particular job. Faster jobs means less time spent waiting for failed jobs to be rerun and therefore less time waiting for the tails of a workflow to finish. Third, one job can now do more work in the same amount of time and therefore fewer jobs are needed to run a workflow. Fewer jobs put less scaling pressure on the workflow management system.

In this paper we will discuss the following topics: an overview of the design of the multi-threaded application and its implementation, plans for how CMS will use the application in the near term, and

[1] To whom any correspondence should be addressed.

Content from this work may be used under the terms of the Creative Commons Attribution 3.0 licence. Any further distribution of this work must maintain attribution to the author(s) and the title of the work, journal citation and DOI.
Published under licence by IOP Publishing Ltd

Journal of Physics: Conference Series **664** (2015) 072026 doi:10.1088/1742-6596/664/7/072026

finally results of performance measurements of the application by itself and when used for processing a large workflow.

2. Multi-threaded Application

2.1 Single-threaded application design

In order to understand the multi-threaded application, it is useful to first understand CMS' initial single-threaded application[1] since the multi-threaded application is an evolution from the initial application. Figure 1 shows a simplified representation of how data is processed in the single-threaded application. As described earlier, events are grouped into blocks. Data pertaining to a block or to an event is processed by running an Algorithm. In figure 1 there are two algorithms: A1 and A2. The framework which drives the application runs algorithms in a preset order.

As shown in figure 1, when the framework sees a new event block (e.g. while reading data from a file) it passes the data associated with that block to each algorithm in a set order. For the application represented by Figure 1 this means first calling Algorithm 1 and then calling Algorithm 2. Once all Algorithms have been informed about a new block, the framework starts processing each event in the block in order. The first event is passed to Algorithm 1 and when it finishes the event is then passed to Algorithm 2. Once all Algorithms have finished with one event the framework proceeds to the next event. Once all events in a block have finished processing, the framework calls each Algorithm in turn to tell them that the block has finished.

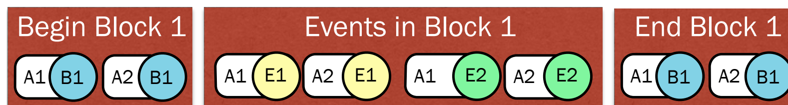

Figure 1. Simplified representation of data processing with a single-threaded application where time is represented along the x axis. Algorithms processing data are represented by the boxes labeled A1 and A2. Data in an event block are represented by the blue circles labeled B1. Data in events are represented by the yellow and green circles labeled E1 and E2.

2.2 Multi-threaded application design

In order to easily reuse the code written for CMS' single-threaded application in a multi-threaded application, the multi-threaded application maintained the concepts of the initial application: blocks, events and Algorithms. The main difference is instead of processing events one at a time, the multi-threaded framework can process multiple events concurrently as shown in figure 2.

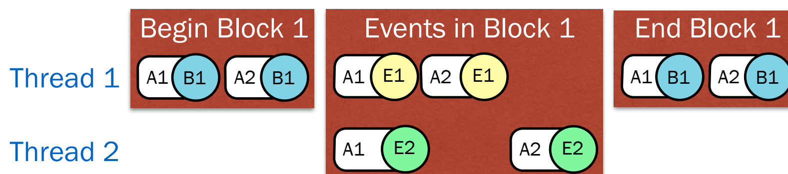

Figure 2. Simplified representation of data processing with a multi-threaded application. Two threads are used to process events in parallel.

C Jones I Using the CMS Threaded Framework in a Production Environment 14/04/2015

As shown in figure 2, when a new event block is seen, the Algorithms are called one after another on just one thread. This is identical to the single-threaded application. Once all Algorithms have been informed of the new block, the framework begins processing a different event on each thread it is controlling. In figure 2 the application is using two threads so it therefore begins to process two events. In figure 2 we see that Algorithm 1 can process event 1 and event 2 concurrently. This is because Algorithm 1 was updated to be thread-safe. Part of that update is to inform the framework that it is safe to run the Algorithm concurrently. In contrast, we see that Algorithm 2 begins processing event 1 on thread 1 but can not begin processing event 2 until it finishes with event 1. This is because Algorithm 2 has not been updated to be thread safe (such Algorithms are referred to as *legacy* Algorithms) and the framework guarantees that one and only one legacy Algorithm can run at a time. Once all events in the block have finished processing, the framework once again uses just one thread and handles the end of block in the same way as the single-threaded application.

Figure 2 illustrates the two areas of potential CPU inefficiency in this design: block processing and legacy Algorithms. If the time it takes to process the begin and end block transitions is significant compared to the time spent processing events then the overall CPU efficiency drops. Similarly, if the time spent running legacy Algorithms is sufficiently large compared to the total time it takes to process one event we can also impact CPU efficiency. The performance measurements discussed later in this paper attempt to quantify the extend of such inefficiencies.

2.3 Changes for multi-threading

Although the multi-threaded application is capable of running Algorithms which have not been updated to be thread-safe, i.e. legacy Algorithms, doing so causes a significant CPU efficiency penalty because some threads are forced to remain idle. Therefore over the last year CMS has embarked on a campaign to update its Algorithms. As of the time of the writing of this paper, 1100 Algorithms have been covered to be thread-safe while leaving 2800 as legacy. The remaining legacy Algorithms have not been converted because either they are not used in standard 2015 multi-threaded workflows (e.g. are only used for testing) or have not been shown to be a significant efficiency problem.

Overall we estimate that it took 6.3 person-years to make all the changes needed to reach our present level of performance for the multi-thread application. Three person-years of that effort were for converting the single-threaded framework into the multi-threaded framework. An additional 3 person-years were required to convert the Algorithms to be thread-safe. The final .3 person-years was spent in modifying external libraries used by CMS. One of the major efforts was in changing ROOT[2] so the parts used by CMS were thread-safe. CMS gratefully acknowledges the effort the ROOT team spent in answering our questions and in incorporating our changes to ROOT back into the official ROOT code base.

3. Use of Multi-threading in 2015

CMS' ultimate goal for multi-threading is to be able to efficiently run all our major processing workflows using multiple threads. However, we do not plan on reaching that goal in 2015. Instead, for 2015 we will only run the reconstruction application using multiple threads while all other applications will use only one thread. The reconstruction is the highest priority in order to guarantee that event blocks containing the highly complex data from the detector can be processed within the time limits imposed by the grid sites. CMS does have a working version of our Geant 4[3] simulation application running with multi-threads. However, the necessary thread-safe version of Geant 4 was not available at the time CMS began creating simulation samples for LHC run 2 during the summer of 2014. We do, however, plan to use multi-threads when we begin our next large simulation campaign.

4. Multi-threaded Application Performance

Given that only the reconstruction workflow will be run with multiple threads in 2015, in this paper we only report the multi-threaded performance of the reconstruction application.

4.1 Performance measurement technique

The application performance measurements were all made using an AMD 64-core Opteron 6376 machine with 126GB of RAM. From previous performance measurements made on different hardware we expect these results to be fairly representative to all hardware presently available on the grid sites accessible to CMS.

Two different data samples were used for the performance measurements. Both samples used the same underlying signal event but differed based on the amount of *pileup* events (events from collisions that happen within the same beam crossing as the signal event and which contribute to the data read out by the detector). The underlying signal events were $t\bar{t}$ Monte Carlo created to mimic the expected LHC conditions for 2015. One data sample used a low pileup scenario (50ns bunch spacing with average of 4 events per crossing) consistent with the beginning of data taking in 2015 while the other data sample used high pileup (25ns bunch spacing with average of 40 events per crossing) consistent with the LHC at the end of 2015. Using two different pileup samples allows us to project how the application will perform as a function of the event complexity.

In addition, two different application configurations were tested. One configuration only used Algorithms needed for reconstruction while the second configuration included Algorithms for both reconstruction and monitoring of the results of reconstruction. Our production workflows always run the reconstruction and the monitoring together. However, given the monitoring has some built in synchronization points as well as higher memory use we need to determine how big of effect that is when using multiple threads.

Finally, we ran both single threaded and multi-threaded jobs. In order to measure the single-threaded performance, we ran 64 concurrent jobs on the machine. To measure the multi-threaded performance we ran 8 concurrent jobs where each job used 8 threads thereby potentially saturating all 64 cores of the machine. We chose 8 threads because 8 is the number of cores our grid sites allocate for each of the jobs in their multi-threaded queues. All of our measurements are given in terms of the ratio of the multi-threaded to the single threaded performance. Both the single and multi-threaded jobs just processed one event block in order to determine the maximum achievable performance of the multi-threaded application.

4.2 Performance measurements

Table 1 and Table 2 summarize the processing time and memory usage performance measurements for the pileup sample and application configuration.

In table 1 we see that when using only the reconstruction configuration the 8 multi-threaded jobs run slightly faster than 64 single-threaded jobs. However, when monitoring is added to the configuration we see that the multi-threaded speed drops. The drop is larger for low pileup than for high pileup. This is expected since as the complexity of event rises the time it takes to do reconstruction (particularly particle trajectory determination, i.e. tracking) increases much faster than the increase in time to do monitoring. Therefore more time is spent in reconstruction at high pileup and reconstruction is already very parallel efficient.

Table 1. Speed of Multi-threaded task relative to Single-threaded

Application Configuration	Pileup	
	Low	High
Reconstruction	1.05	1.04
Reconstruction & Monitoring	0.83	0.94

Table 2 shows how much more memory is used by 64 single-threaded jobs compared to 8 multi-threaded jobs. We see that one gets more memory savings at low pileup than at high pileup. This is because at high pileup the amount of memory needed to process an individual event increases while the memory shared by all events (e.g. geometry) remains constant. We can also see that the addition of monitoring to the application decreases the memory savings. This is also expected since the way monitoring was changed to be thread-safe was to allow each thread to have its own copy of the histograms being filled. Once all events have been processed those separate copies are merged together to get the final histogram. Therefore the memory needed for monitoring increases linearly with the number of concurrent events used in the application.

Table 2. Resident Memory (RSS) Savings for Multi-threaded task

Application	Pileup	
Configuration	Low	High
Reconstruction	4.6	3.7
Reconstruction & Monitoring	3.3	2.7

5. Full Scale Testing

Measuring the CPU efficiency of the application alone is useful to set an upper bound on the CPU efficiency which could be achieved when running a large scale workflow on the grid. The actually achievable efficiency is dependent upon many factors such as event complexity, number of events per block, network congestion at sites etc. In order to better understand the efficiency factors we did a large scale workflow test using the multi-threaded application to process a representative data sample.

5.1 Performance measurement technique

The full scale test was done by reprocessing a sample of data taken during the LHC Run 1 period. The complexity of the events in this data sample are closer to the low pileup events than the high pileup events used in the application performance test. Also this data sample had a wide variation in the number of events in each block. This variation was because of the way CMS clusters events in the same file based on preselection criteria. Unlike the application performance test, the jobs in the full

scale test processed more than one event block and therefore were affected by the synchronization at the block boundaries.

The reprocessing jobs used our standard reconstruction and monitoring workflow. The test used an older CMS software release than the application performance test where the older release had a lower CPU efficiency for threading and included a few known thread-safety problems. Because of these factors the jobs only used four threads instead of eight threads as done in the application performance testing.

The scale test ran jobs on all CMS Tier 1 computing sites and was able to make use on average of 50% of the available CPUs at the sites.

5.2 Performance measurements
One purpose for doing a full scale test of the multi-threaded application was to attempt to find thread-safety issues which happen rarely and which were not found during the development and validation stages. As such, of the 45,000 jobs run during the full scale test, 72 of the jobs failed due to a problem which may have been caused by the application as apposed to failures due to transient problems at a given grid site. If we attribute all application failures to be caused by thread-safety issues, which is an over estimate, then our threading failure rate was less than 0.2% per job. Normally the CMS workflow system will retry a failing job four times in order to be less sensitive to transient problems. From our experience, failures due to thread-safety problems tend to appear randomly in a job, i.e. if a job fails once then further runs of the same job usually succeed. Therefore the chance that a job with a greater than 99.8% success rate failing four times in row is 1 in $160x10^9$. That overall failure rate is much less than failures due to site related problems. Therefore the present upper bound of expected thread-safety related problems is small enough to not impede the running of large workflows.[2]

Figure 3 shows the CPU efficiency achieved for all the full scale test jobs running the application both with four threads and with one thread. The four threaded job achieved an average CPU efficiency of

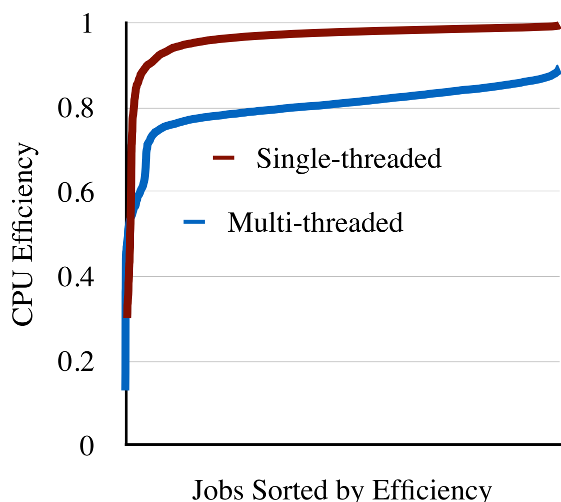

Figure 3. CPU efficiency for full scale test comparing multi to single-threaded jobs.

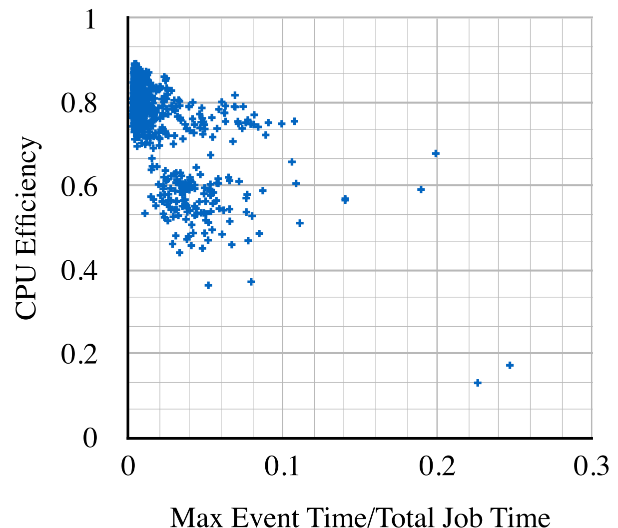

Figure 4. CPU efficiency for full scale multi-threaded jobs versus the fraction of time taken by the longest running event.

[2] We will, of course, fix any thread-safety problems we encounter.

0.8 while the single-threaded job reached 0.97. This is in contrast to the lowest CPU efficiency measured during the application performance testing of 0.83 efficiency for 8 threads. The main reason the CPU efficiency for the multi-threaded jobs are much less than for the application performance measurements is the use of the older version of the CMS software for the full scale test.

From figure 3 we can also see that some of the jobs had much lower CPU efficiencies than others. These lower efficiencies were related to jobs which had one event take an appreciable amount of the total job time. This correlation can be seen in figure 4. These long running events are primarily the first event being processed and are usually caused by an unusually long I/O related latency from either reading conditions data from the database or from the first read from a file which resides at a remote site. In addition, no strong correlation was found between low CPU efficient jobs and the number of events within the blocks processed by those jobs.

6. Conclusion and Outlook

CMS has successfully transitioned from its original single-threaded application to a multi-threaded application capable of processing multiple events concurrently. From the full scale workflow test we find that the failure rate caused by potential thread-safety problems is small enough to not impede large scale workflows during LHC Run 2. From the application performance testing we also find that running reconstruction and monitoring on the types of events expected for Run 2 will give an acceptable CPU efficiency and significant memory savings.

Although CMS has achieved an acceptable level of efficiency for multi-threaded jobs we still plan on improving our efficiency. We will convert more of our Algorithms to be thread-safe, specifically those Algorithms used in other CMS workflows in order to allow all workflows to exploit multiple threads. We also plan to exploit additional levels of parallelism such as running multiple Algorithms simultaneously on the same event and being able to parallelize across event block boundaries.

7. References

[1] Jones C D, Paterno M, Kowalkowski J, Sexton-Kennedy L and Tanenbaum W 2006 *Proc. CHEP 2006* vol 1, ed S Banerjee (India: Macmillan) pp 248-251
[2] http://root.cern.ch/root/
[3] The Geant4 Collaboration (Agostinelli S et al.) 2003 *Nucl. Instr. Meth. A* **506** 250-303

Acknowledgements

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