Available on CMS information server **CMS CR -2020/060**

The Compact Muon Solenoid Experiment

Conference Report

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

10 February 2020 (v3, 27 February 2020)

Using OpenMP for HEP Framework Algorithm Scheduling

Christopher Jones for the CMS Collaboration

Abstract

The OpenMP standard is the primary mechanism used at high performance computing facilities to allow intra-process parallelization. In contrast, many HEP specific software packages (such as CMSSW, GaudiHive, and ROOT) make use of Intels Threading Building Blocks (TBB) library to accomplish the same goal. In this talk we will discuss our work to compare TBB and OpenMP when used for scheduling algorithms to be run by a HEP style data processing framework (i.e. running hundreds of interdependent algorithms at most once for each event read from the detector). This includes both scheduling of different algorithms to be run concurrently as well as scheduling concurrent work within one algorithm. As part of the discussion we present an overview of the OpenMP threading model. We also explain how we used OpenMP when creating a simplified HEP-like processing framework. Using that simplified framework, and a similar one written using TBB, we will present performance comparisons between TBB and different compiler versions of OpenMP.

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

Using OpenMP for HEP framework algorithm scheduling

Christopher Jones[1],[*], *Patrick* Gartung[1]

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

Abstract. The OpenMP standard is the primary mechanism used at high performance computing facilities to allow intra-process parallelization. In contrast, many HEP specific software packages (such as CMSSW, GaudiHive, and ROOT) make use of Intel's Threading Building Blocks (TBB) library to accomplish the same goal. In this talk we will discuss our work to compare TBB and OpenMP when used for scheduling algorithms to be run by a HEP style data processing framework (i.e. running hundreds of interdependent algorithms at most once for each event read from the detector). This includes both scheduling of different algorithms to be run concurrently as well as scheduling concurrent work within one algorithm. As part of the discussion we present an overview of the OpenMP threading model. We also explain how we used OpenMP when creating a simplified HEP-like processing framework. Using that simplified framework, and a similar one written using TBB, we will present performance comparisons between TBB and different compiler versions of OpenMP..

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. This framework makes use of Intel’s Threading Building Blocks (TBB) library [2] to handle scheduling of processing tasks across the limited number of threads available to the process. The framework supports concurrency on three different levels via TBB. The first is concurrently processing multiple transitions, e.g. multiple *Event* or *begin of LuminosityBlock* transitions. The second is allowing different algorithms to run concurrently during each transition. The third level is within a given algorithm concurrent tasks can be scheduled and run. On the whole, CMS has found this system to be very successful.

Given the success of the system, why did we bother with exploring the use of OpenMP [3] to do the same processing? The reason is the growing need for CMS to exploit resources from High Performance Computing (HPC) facilities in the coming years. These facilities typically support only OpenMP as the intra-process concurrency mechanism. We have found when we communicate with HPC specialists, they often ask why we are not using OpenMP for concurrency. As CMS’s utilization of HPC facilities increases we should either have a strong case for why the software does not use OpenMP or we should convert to using OpenMP.

In this paper we will present our findings of a comparison between TBB and OpenMP via the use of demonstrator frameworks. We begin by presenting a review of the relevant OpenMP commands used to create a demonstrator framework capable of the three levels of

[*] Corresponding author: cdj@fnal.gov

concurrency already supported by CMS's framework. We then go on to briefly describe the abilities of the demonstrator frameworks. This is then followed by the experimental setup used to do the measurements as well as the results of the measurements.

2 Review of OpenMP Commands

OpenMP is implemented as an extension to a C++ compiler. That is in stark contrast with TBB which is a standard C++ style third party library. OpenMP C++ syntax is implemented as pragma statements dictated by the OpenMP standard. How the OpenMP features are implemented by a given compiler can vary greatly from compiler to compiler as the OpenMP standard gives a great deal of freedom for the implementations. These large variations can be true across versions of a compiler as well as across compiler vendors.

In the rest of this section we will describe four OpenMP 4.5 constructs which we used to construct the demonstrator framework: *omp parallel*, *omp for*, *omp task*, and *omp taskloop*.

2.1 Construct: *omp parallel*

The *#pragma omp parallel* statement starts threads which are then used to process the C++ block directly following the statement. Once assigned, those threads can only be used by that parallel construct. (This is relevant for the case of nested parallel blocks we will discuss in subsection 2.3.) The thread which first encountered the pragma statement, OpenMP refers to this thread as *master*, will also join in processing the block. The master thread will not continue past the end of the block until all other threads used by the *omp parallel* statement have finished with the block. What happens with the other threads used for processing is implementation defined, not dictated by the OpenMP standard.

The number of threads used by each parallel construct is controlled by the environment variable, *OMP_NUM_THREADS* or by calling the function *omp_set_num_threads*. The maximum number of concurrently running threads which OpenMP is allowed to use for one job can only be set via the environment variable *OMP_THREAD_LIMIT*.

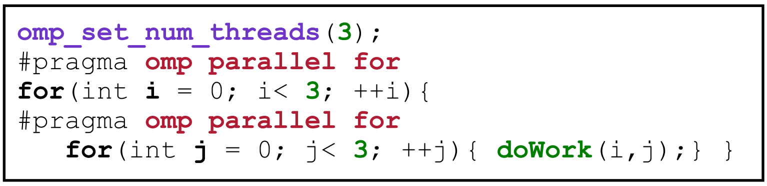
2.2 Construct: *omp for*

The OpenMP *for* construct, *#pragma amp for*, must directly precede a *for* loop and is used to distribute the iterations of the *for* loop to threads associated with the inner most *parallel* statement. By default, the master thread waits until all iterators have completed before moving onto any C++ statements following the *for* loop.

The OpenMP *for* construct and OpenMP *parallel* construct can be combined into one statement for ease of use.

2.3 Nested parallel blocks

In OpenMP, support for concurrent nested parallel blocks is implementation defined. If supported, the feature is controlled via the environment variable *OMP_NESTED* or by calling the function *omp_set_nested*. In addition, the number of threads assigned to the inner nested parallel blocks is the same as the number of threads assigned to the outer most block. How threads are assigned, in the case where nested parallelism is supported, is explained below using an example.



```
omp_set_num_threads(3);
#pragma omp parallel for
for(int i = 0; i< 3; ++i){
#pragma omp parallel for
   for(int j = 0; j< 3; ++j){ doWork(i,j);} }
```

Fig. 1. Example illustrating the use of nested *parallel* blocks with each block using OpenMP *for* constructs. In the example, the function *doWork* will be called 9 times which means there is the possibility for 9 concurrently running calls to the function.

Figure 1 shows code for a parallel nested *for* loop using OpenMP. The call to *omp_set_num_threads* restricts each parallel *for* construct to use 3 threads. The loop over *i* can use three threads (one per iteration) and each of those threads each see the inner loop over *j*. In turn each of the *j* loops can also use up to 3 threads. Therefore the maximum number of concurrent threads for the doubly nested loop is 9. Since the total number of calls to *doWork* is also 9, it is theoretically possible to have all 9 calls running concurrently. As explained earlier, the master thread for each inner loop must wait for each of the iterations to finish before proceeding. Similarly, the master thread for the outer loop must wait for all inner loop master threads to finish before proceeding. Figure 2 shows two different examples where different numbers of maximum threads per job are used to run the code from Figure 1. In the left sub figure, the maximum number of threads is 9 and we see the main thread only has to wait until the longest running inner iteration finishes. For the right sub-figure the total number of threads is only 6. In this case, the *i = 0* loop gets three threads, the *i = 1* gets 1 thread and the *i = 2* loop gets 2 threads. Once threads are assigned to the inner loops, they can not be re-assigned. Therefore the main thread must wait for the *i = 1* iteration to process all three inner iterations on a single thread before it can proceed. The other threads are not allowed to do any other work once they have finished their iterations. Clearly this behavior does not make the most efficient use of the available threads.

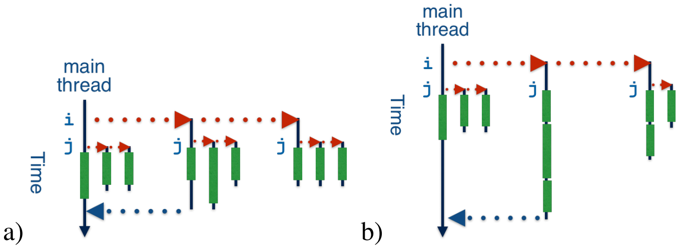


Fig. 2. Possible distributions of work for nested *for* loops shown in Figure 1. Sub-figure a) shows the optimal distribution when a total of 9 threads are used. Sub-figure b) shows one potential sub-optimal distribution where only a maximum of 6 threads are allowed.

2.4 Construct: *omp task*

The *omp task* construct, *#pragma amp task*, is used to place all code in the block following the construct into a task object. The task object is then scheduled to run on a thread. If the *untied* keyword is also used when declaring the task, the resultant task can be run by any thread being controlled by the inner most *parallel* construct. When a task completes, another task can be scheduled on that thread. The only restriction on the following task is it must be from the same *parallel* construct.

2.5 Construct: *omp taskloop*

The *omp taskloop* construct, *#pragma omp taskloop*, is very similar to the *omp for* construct except each iteration is encapsulated into an OpenMP task. In addition, the master thread may run other non-iteration tasks while waiting for all the tasks created by the *taskloop* to end. This is known as ***task stealing*** which some OpenMP implementations use.

3 Demonstrator Frameworks

Three separate, but related, simplified demonstrator data processing frameworks were used for the experiments. One framework used OpenMP to schedule work, the second uses TBB, and the third is a single threaded framework. All three frameworks use the same configuration file format thereby making it easier to run the same configuration using different technologies. In addition, all the frameworks process an input sequence of Events.

All three frameworks bundle the work needed to be done into *Modules*. A Module generates data and puts it into an Event. A Module can depend on data from another module and the frameworks guarantee proper ordering of Modules based on that dependency. For the multi-thread capable frameworks, the execution of a Module is wrapped in either an OpenMP or a TBB task. A Module's task only starts once the data needed by the Module is available in the Event. When implementing a Module, the use of either OpenMP's or TBB's *parallel for* construct is allowed. This allows testing of nested parallelism for the different technologies.

The code for all three frameworks and all the Modules can be found at GitHub [4].

4 Experimental Setup and Results

We used the demonstrator frameworks to allow fair performance comparisons for the three different cases: OpenMP and TBB using multi-threading, and running N concurrent single-threaded processes. Given that the implementations of OpenMP differ across compilers, we made all the performance measurements using both the gcc 8 [5] and the clang 7 [6] compiler. For the single-threaded and TBB cases, the performance differences of the gcc and clang build executables were indistinguishable within measurement error. Therefore for TBB and the single-threaded performance numbers we only report a single number rather than one for each of gcc and clang.

The configuration used to run the tests was built to emulate the behavior of the actual CMS reconstruction application. This was achieved by using the same data dependency between Modules as the actual CMS reconstruction process. In addition, the time each Module ran for each Event matched the time spent by the equivalent Module in the reconstruction process. Timings for 100 different Events were used in the test in order to simulate the effect of varying Event processing times.

In each experiment the number of cores used was varied and the total Event throughput (in Events per second) was measured. For OpenMP and TBB this meant varying the number of threads used. For the single-threaded framework, the total number of simultaneously running jobs was varied. For the OpenMP and TBB measurements, the number of concurrently processing Events was set to be equal to the total number of threads used in the job. In addition, the total number of Events processed per job was equal to 100 times the number of threads used in the job. This guaranteed that each Event time was reused exactly the same number of times for all jobs. Across experiments, the amount of internal parallelism within a Module was changed to see the effect of nested parallelism. All the experiments were done using an Intel Xeon Phi (a.k.a Knights Landing) CPU [7] with 64 physical cores with each core supporting 4 hardware threads.

Figure 3 shows the Event throughput versus core utilization for the case where all Modules used in the threaded frameworks are configured to be concurrent capable. That is each Module can handle concurrent processing of different Events. For this case of perfect parallelism, both implementations of OpenMP and TBB show equivalent performance. The slope changes at 64 and 128 threads is caused by the use of additional hardware threads per core of the Xeon Phi device.

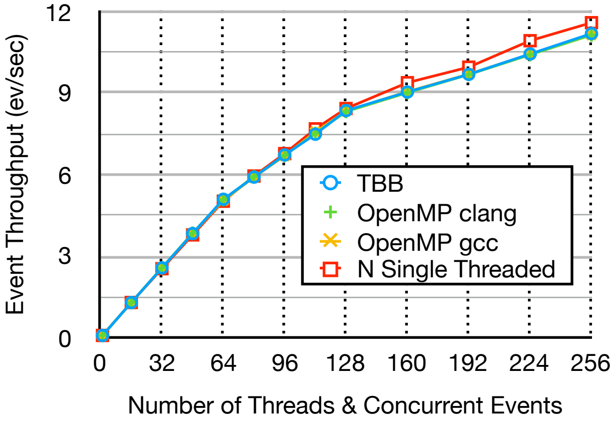


Fig. 3. Event throughput as a function of utilized cores where all Modules are able to run concurrently.

A more realistic configuration is to change the Module used to simulate the behavior of the OutputModule, which writes the resultant processed Events out to a file, such that the simulated OutputModule only be able to process one Event at a time. This serialization was accomplished in a non-blocking manner so that other Modules do run while an Event waits for the OutputModule to become available. Even with the ability to schedule around the OutputModule, all the threaded frameworks still hit the serialization limit of 0.9 Events/second at around 16 threads.

One way to minimize the serialization limit is to allow internal parallelism within the OutputModule. For this paper we had the OutputModule execute a loop for 100 iterations and then used OpenMP and TBB constructs to allow the iterations to be run concurrently. Figure 4 shows the results using different techniques.

Figure 4a shows the case where the TBB based framework used *tbb::parallel_for* and the OpenMP framework used the *taskloop* construct. In this case, both TBB and the clang version of OpenMP employ *task stealing* while the gcc implementation just does a wait. In task stealing, if the master thread that is running the concurrent *for* loop finishes its allotted work before all the other threads working on the *for* loop finish, that thread can run another scheduled task which is unrelated to the *for* loop. Only once the unrelated task finishes can the master thread proceed with work following the completion of all iterations of the *for* loop. In this experiment, task stealing is shown to be detrimental as the extra work done while in the OutputModule keeps the OutputModule from finishing its work as soon as possible and therefore delaying the time before another Event can use the OutputModule.

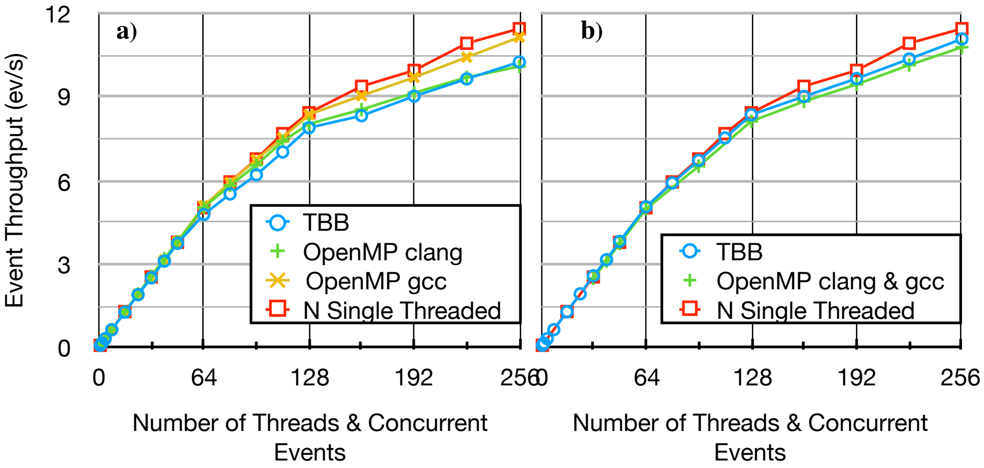


Fig. 4. Event throughput as a function of utilized cores where a) the OutputModule is serialized and uses internal parallelism with the possibility of task stealing and b) it is serialized and uses internal parallelism with task stealing prohibited. Both plots share the same Y axis.

Figure 4b shows the case where task stealing is prevented. For TBB it was a simple case of putting the *tbb::parallel_for* within a TBB task arena. The only way the OpenMP standard guarantees no task stealing is with the use of the omp *for* construct. As explained earlier, when using *omp for* one must specify the number of threads to use for the loop via the call to omp_set_num_threads. In order to make Figure 4b, for each point on the x axis we ran 8 to 10 jobs where the total number of threads was fixed while each job used a different value in the *omp_set_num_threads* call. The throughput of the jobs varied routinely by over a factor of 2. Only the result giving the highest Event throughput was added to Figure 4b. Even doing our best to hand tune each OpenMP point, the automatic behavior of TBB gives the best throughput.

5 Conclusion

In this paper we have shown that it is possible to create a multi-threaded HEP data processing framework using OpenMP. However, it is also shown that using TBB's automatic scheduling provides a better throughput than a hand tuned OpenMP. Such automatic scheduling is extremely important as HEP processing vary widely in time per

Module as well as the mixture of Module types (re-entrant and non-reentrant) as well as the number of threads used to run a job.

We have also seen that compiler variations in the implementation of OpenMP make portable performance hard. In particular gcc *taskloop* does not do task stealing while the clang implementation of *taskloop* does do task stealing with no way to disable that feature.

The major takeaway from the paper is OpenMP 4.5 has composability difficulties. In particular, OpenMP *parallel* blocks do not share threads which leads to nested parallelism using fixed allocation of threads. The fixed allocation makes it very hard to tune how many threads to use at each nested *parallel* level, particularly if the optimal number can vary during the execution of the program.

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
3. https://www.openmp.org
4. http://github.com/Dr15Jones/toy-mt-framework
5. https://gcc.gnu.org
6. https://clang.llvm.org
7. https://ark.intel.com/content/www/us/en/ark/products/codename/48999/knights-landing.html