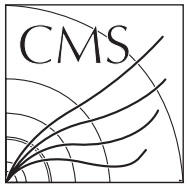
Available on CMS information server

CMS CR -2010/305



The Compact Muon Solenoid Experiment

Conference Report

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

21 December 2010 (v2, 23 December 2010)

Multicore-aware applications in CMS

Christopher Jones, Pete Elmer, Liz Sexton Kennedy, Chris Green, Andrew Baldooci

Abstract

One of the significant trends of recent years has been the move towards multicore CPU's with ever increasing numbers of cores. CMS has been preparing multicore aware applications that rely on "multi-processing", namely the sharing of memory between processes forked from a single parent process. First experience with deploying these applications for production will be presented, as well as results from detailed profiling done to understand the limits of scaling with increasing numbers of cores.

Presented at *CHEP2010: International Conference on Computing in High Energy and Nuclear Physics 2010*

Multi-core aware applications in CMS

C D Jones[1], P Elmer[2], L Sexton-Kennedy[1], C Green[1] and A Baldooci[1]
[1] Fermilab, P.O.Box 500, Batavia, IL 60510-5011, USA
[2] Physics Department Princeton University, Jadwin Hall, Princeton NJ 08544, USA

E-mail: cdj@fnal.gov

Abstract. One of the significant trends of recent years has been the move towards multicore CPU's with ever increasing numbers of cores. CMS has been preparing multicore aware applications that rely on "multi-processing", namely the sharing of memory between processes forked from a single parent process. First experience with deploying these applications for production will be presented, as well as results from detailed profiling done to understand the limits of scaling with increasing numbers of cores.

1. Introduction

HEP data processing is naturally parallelizable given that we have billions of statistically independent events. All experiments exploit this parallelization by processing events concurrently in separate machine processes. However, this style of data processing may not be possible in the future due to memory limitations in future architectures. Historically, memory per unit cost has increased at the same rate as the number of transistors in a CPU[1]. However, IT infrastructure funding levels for HEP experiments are not guaranteed to stay at our present levels so just because experiments can afford to put 2 GB per core now does not mean they will be able to afford to do that into the future. In addition, opportunistic use of grid sites improves if we lower our memory requirements since not all grid sites have 2GB per core. Finally, there are up coming technical limitations on connecting many cores to shared system memory[2]. Therefore future applications will likely need to find ways to decrease their memory footprints.

Multi-core aware applications can improve memory sharing. This can be achieved in two different ways: threading and forking. Either of these mechanisms would allow processing of multiple events simultaneously while sharing resources across events. In threading, all threads share the same address space but have to worry about concurrent usage. In forking, one parent process is started and then the parent 'clones' itself into independent child processes. Each child process gets its own address space but untouched memory setup by the parent process is shared between the child processes.

In this paper we will describe and provide measurements of CMS' use of forking and provide estimates for performance gains that might be possible by using threading on a sub-event level.

2. Forking

2.1. Copy on write

When forking, you start with a parent process and this parent then calls 'fork' to create child processes. Each child process starts by sharing the same memory pages as its parent. If a child asks for new memory it will be given a new page which it exclusively owns. If a child attempts to write to a memory page it shares with its parent or another sibling then the operating system will make a copy of that memory page and give sole ownership of that copy to the child. This is known as *copy on write*.

Therefore to maximize memory sharing between children the parent process needs to load into memory often used, nonvolatile data such as conditions, calibrations or geometry.

2.2. Use in CMS

CMS has one main application which is used for all offline data processing. This application was updated to accommodate forking and the steps used by the application during the forking process are described below.

When the parent process starts it first reads the configuration file, loads all the shared libraries containing the modules listed in the configuration and then creates instances of the modules. The configuration also says how many child processes should be forked and how the events should be distributed between children. The second step is to open the input file and find the first run which will be processed. The run is then used to prefetch all conditions, calibrations and geometry. During this step no event processing modules are called and no events are ever processed. The third step is to send a message to all modules telling them that forking is going to happen. This is done to allow them to release any resources, e.g. the source module closes the input file. The final step is to fork the children.

The child processes perform the following steps. First they redirect standard out and standard error to their own files whose names contain the parent PID and the child number (i.e. the first forked child uses the number 0). Second the child process sends messages to all modules saying that this process is child number *X*. The output modules use this information to append the child number to all file names. The source uses this information to calculate the event ranges to process (no interprocess communication is used) and then reopens the input file. Finally the child process processes events normally except for only processing a limited range of the events in the input files.

2.3. Measurements

All measurements were performed using CMS' 64bit reconstruction code on a 4 CPU, 8 core/CPU 2GHz AMD Opteron™ Processor 6128. In addition, all input files were read from a local disk and all output files were written to the same local disk.

Below is shown the shared and private memory for the parent and child processes. Figures 1 and 2 show how the private and shared memory for the parent process, in blue, and four child processes change over time. Initially, all memory is held privately by the parent since no children were yet forked. After initialization and prefetching have finished, around the two minute mark, all four children are forked and the private memory temporarily goes to 0 since all memory becomes shared. However, right after that mark we see that each child gains each own private memory used for event processing. In addition we see that some of the previously shared memory is returned to the parent process. This 'returned' memory is probably due to memory buffers still held by the parent for the input file or to the condition objects containing caches which get reset during event processing. Once in steady state event processing we see that we get approximately 700MB of memory shared between all the children and approximately 375MB of private memory for each child. For a 32 core machine we find that forking takes a total of 13GB while running 32 independent processes takes 34 GB. Therefore forking gives us a memory sharing of 62%.

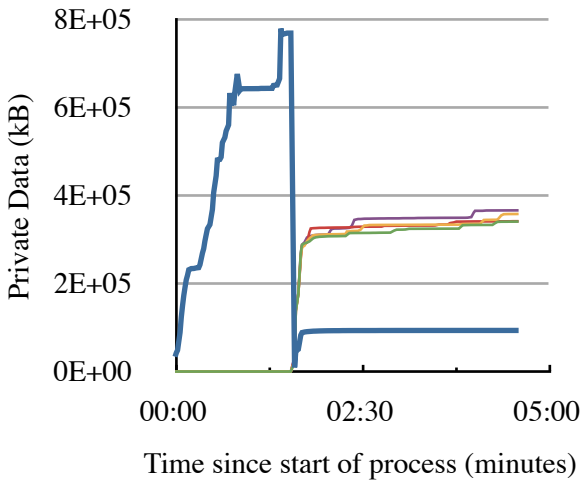


Figure 1. Amount of private data owned by parent (blue) and child processes

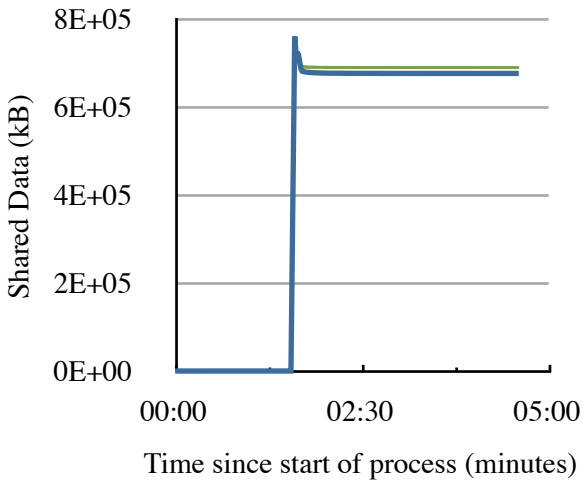


Figure 2. Amount of shared data between parent and child processes

Another measurement of interest is throughput, i.e., how many events per unit time can be processed. Figure 3 shows the throughput as a function of the number of forked children. From the figure we see that up to 32 children we get a consistent throughput for each child but once we attempt to use more

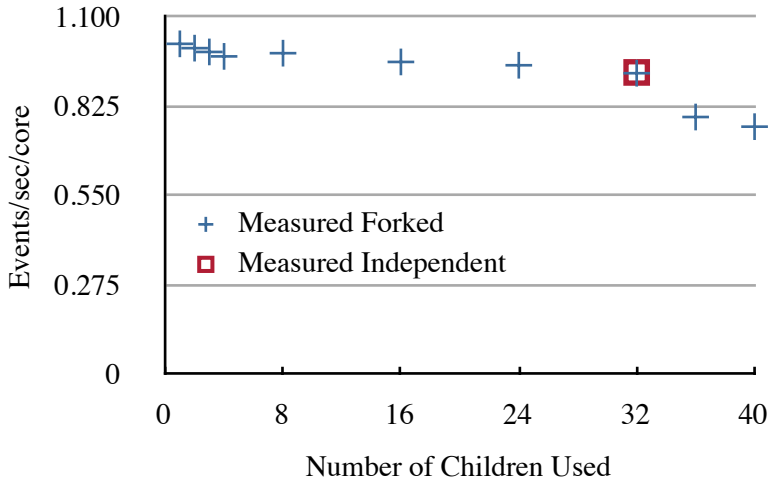


Figure 3. Events per second per core vs the number of forked child processes for a data file ignoring startup and shutdown effects.

children than there are cores on the machine we do not gain in throughput. We also see that we get the same throughput for 32 independent processes, shown by the red box, as we do for a job with 32 forked children, the blue cross. The values shown in figure 3 were calculated by taking the total number of events processed by all children divided by the sum of the time taken by each child to just process events. Therefore the values ignore edge effects created by startup and shutting down of the processes.

Depending on how events are distributed to the different child processes, some children may finish processing their allotted events before other children thereby causing some cores to be idle towards the

end of a processing job. We refer to the variation in end time as *dispersion*. In the present implementation of forking in the CMS application, the framework pre-assigns which events will be processed by each child. The algorithm is as follows. In the configuration, the user specifies how many contiguous events, `N`, each child will process, e.g. `N`=100. The starting point for each child is just determined by its child number and the number of contiguous events to process. So if we had three children, the first child would start at event 1, the second child would start at event 101 and the third would start at 201. Once a child is finished with its contiguous section, e.g. child one finishes event 1 to 100, it skips `(number of children-1)*(N)` events and then begins processing `N` contiguous events. So for child 1 it would processes 300 to 301. This continues until the child reaches the end of the number of available events. We measure dispersion by calculating *utilization*:

(sum of child processing time)/[(max child processing time)*(number of children)]

If all children finish their processing at the same time the utilization will be equal to one, else it will be less than one.

We found reasonable utilization values when running reconstruction jobs on different monte carlo event types using the largest value for the sequential number of events to process, i.e. total events divided by the number of children. High Pt QCD samples gave utilization of 0.92, TTbar gave 0.92 and minimum bias gave 0.85. These samples were chosen since high Pt QCD samples take the longest time to processes one event, minimum bias takes the least time and TTbar falls between those two extremes. However, when processing real data files we have found utilizations as low as 0.38. After investigating these files we found that the events in the beginning of the file generated output events which were twice the size on disk as the events at the end of the file. However, if we changed from using the largest possible sequential events to process to the smallest, i.e. just process one event and then skip, we were able to get a utilization of 0.95 for that file.

From the previous paragraph we see that a large value of `N` can lead to poor utilization if event characteristics change over the length of the input file (i.e. change with time). However, large values of `N` are very good for I/O. ROOT has a read-ahead cache so large `N` means more reads from one cache fill. When using the maximum value for N on a 32 core machine we see an average total read operations per second after startup of 6.2 operations/s and an average read size after startup of 600kB. However, when using `N=1` we get 156 operations/s and an average read size of only 25kB. Clearly a large `N` gives better I/O performance.

Besides better I/O performance for the reconstruction job, using the maximum value for `N` also improves the I/O characteristics of the resulting file created by merging the output files of all child processes. The first reason for this is a constraint in the CMS framework which requires all events from the same luminosity section be processed in one block. In CMS, a new luminosity section is created every 23 seconds during data taking. As its name implies, a luminosity section is the smallest granularity of timing for which CMS measures integrated luminosity. For proper luminosity accounting CMS requires all events for a luminosity section must be processed by a job or none of them should be. The second reason for `N` affecting read performance of the merged file is CMS performs a merge by simply concatenating the output files using a fast copy mechanism. Therefore all the events processed by the first child are in the beginning of the file, all the events from the second child are added right after that and so forth until the last child's events are placed at the end of the merged file. Therefore if N is small enough such that each child jumps from events in one luminosity section to events in another luminosity section it causes events from one luminosity section to be distributed throughout the resulting merged file. Reading back such a merged file leads to many random accesses throughout the merged file. For a merged file created from a job with `N` set to its maximum size we saw 5.9GB read from ROOT's read-ahead cache. For a merged file created from a job with `N=1` we saw only 750MB read from the cache.

The optimum solution is to keep events from the same luminosity section together in the resulting merge file. This happens automatically for N set to its maximum value. However, since large N can lead to poor CPU utilization we are working on an alternative. Instead of having each child process write out one temporary file we will have each child write one temporary file per luminosity section. The merge job will then read the temporary files in luminosity section order and therefore group all events of a luminosity section together in the resultant merged file.

3. Threading

CMS has found that forking has been very useful for event level parallelization of our legacy framework. However, if event processing latency matters (e.g. in online processing) or if in the future the memory requirement for processing an event is too large to allow further event level parallelization then CMS may need to be able to do sub-event level parallelization. In that case threading seems like a viable mechanism. In this section we will provide performance estimates for one simple parallelization technique: simultaneous running of different modules.

In the CMS framework, jobs are composed of different algorithms, called modules, which are run in sequence. The event is passed from one module to the next where each module reads from and then writes to the event. Once all modules have finished processing the event is written out. However, if two modules are not dependent upon each others results in principal those two modules could be run simultaneously in different threads.

3.1. Methodology

One can estimate the performance benefits from module level parallelization if one knows the average time each module takes to process an event and which modules create data used by another module. CMS' framework records both pieces of information. The steps of the calculation are as follows.

- Calculate the start and end time of each module. The start time of a module is the end time of the last to stop dependent module. The end time of a module is just the start time plus the average event processing time for that module.
- The end time of the last to finish module is the estimated parallel processing time for one event.
- The sum of all the average event processing time for all modules is the estimated serial processing time for one event.
- The estimate of the number of concurrently running modules in a given time period is obtained by counting how many module's start/end times overlap with each other.

3.2. Estimates

Figure 4 shows the estimated number of concurrently running modules as a function of time for processing one TTbar event. The figure shows that we get short periods of high parallelism and extended periods of only one or two modules running concurrently. At the beginning of processing one event we see large parallelism from all the different calibration modules all doing work for their individual sub-detectors. The first highly sequential section, around 0.6 to 1.2 seconds, is when the various tracking modules are working. Once tracking finishes we see another short burst of parallelism which ends in another fairly sequential section, around 1.3 to 1.6 seconds, in which the electron and muon modules are running.

Table 1 summarizes the module parallelization estimates done for three different event types: minimum bias, TTbar and high P_t QCD. We see that to get the maximum performance improvement of 2.2 to 2.6 times speed up requires the use of 16 to 26 independent threads. However, instead of assuming the availability of an infinite number of threads we can instead impose a maximum number

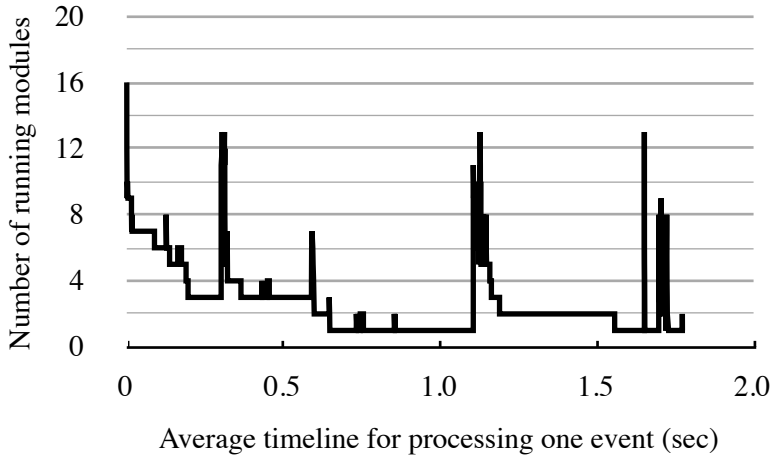


Figure 4. Number of concurrently running modules versus the average timeline for processing one TTbar event.

of threads. If the maximum number of concurrently runnable modules exceeds the maximum number of threads we assume equal sharing of the time available and scale the running time of the modules accordingly. E.g., if we have a maximum number of 4 threads but for the time span of interest we could run 8 modules concurrently then we scale the time of that span by a factor of 2 (=8/4). Applying this scale factor for a range of thread counts from 1 to the maximum number of threads we find that we could reach 90% of the maximum speedup by only using 4 threads.

Table 1. Comparison of estimates of module parallelism speedup for different event types

Event Type	**Max number of threads**	**Average number of threads**
Minimum bias	26	2.64
T Tbar	16	2.62
High Pt QCD	20	2.19

4. Conclusion

We have presented two separate approaches for multicore applications: forking and threading. CMS has found that forking provides good event level parallelization while allowing substantial savings of memory because of memory sharing between child processes. However, further work needs to be done to guarantee good I/O performance. On the other hand, threading may be needed in the future to use multiple cores to speed up the processing of a single event. However, the present decomposition of algorithms into modules is not conducive to high parallelization. The work that would be needed to make present code thread safe is beyond the potential gains. Therefore we find that for now and the near future, forking provides the best benefits.

5. References

[1] http://www.jcmit.com/memoryprice.htm
[2] http://www.intel.com/technology/itj/2007/v11i3/3-bandwidth/7-conclusion.htm