**14 May 2012 (v2, 28 May 2012)**

# Multi-core processing and scheduling performance in CMS

Jose Hernandez Calama for the CMS Collaboration

**Abstract**

Commodity hardware is going many-core. We might soon not be able to satisfy the job memory needs per core in the current single-core processing model in High Energy Physics. In addition, an ever increasing number of independent and incoherent jobs running on the same physical hardware not sharing resources might significantly affect processing performance. It will be essential to effectively utilize the multi-core architecture.

CMS has incorporated support for multi-core processing in the event processing framework and the workload management system. Multi-core processing jobs share common data in memory, such us the code libraries, detector geometry and conditions data, resulting in a much lower memory usage than standard single-core independent jobs.

Exploiting this new processing model requires a new model in computing resource allocation, departing from the standard single-core allocation for a job. The experiment job management system needs to have control over a larger quantum of resource since multi-core aware jobs require the scheduling of multiples cores simultaneously. CMS is exploring the approach of using whole nodes as unit in the workload management system where all cores of a node are allocated to a multi-core job. Whole-node scheduling allows for optimization of the data/workflow management (e.g. I/O caching, local merging) but efficient utilization of all scheduled cores is challenging. Dedicated whole-node queues have been setup at all Tier-1 centers for exploring multi-core processing workflows in CMS.

We will present the evaluation of the performance scheduling and executing multi-core workflows in whole-node queues compared to the standard single-core processing workflows.

Presented at *CHEP 2012: International Conference on Computing in High Energy and Nuclear Physics*

# Multi-core processing and scheduling performance in CMS

**J M Hernández[1], D Evans[2] and S Foulkes[2]**

[1] CIEMAT, Madrid, Spain

[2] Fermilab, Batavia, USA

E-mail: `jose.hernandez@ciemat.es`

**Abstract.** Commodity hardware is going many-core. We might soon not be able to satisfy the job memory needs per core in the current single-core processing model in High Energy Physics. In addition, an ever increasing number of independent and incoherent jobs running on the same physical hardware not sharing resources might significantly affect processing performance. It will be essential to effectively utilize the multi-core architecture.

CMS has incorporated support for multi-core processing in the event processing framework and the workload management system. Multi-core processing jobs share common data in memory, such us the code libraries, detector geometry and conditions data, resulting in a much lower memory usage than standard single-core independent jobs.

Exploiting this new processing model requires a new model in computing resource allocation, departing from the standard single-core allocation for a job. The experiment job management system needs to have control over a larger quantum of resource since multi-core aware jobs require the scheduling of multiples cores simultaneously. CMS is exploring the approach of using whole nodes as unit in the workload management system where all cores of a node are allocated to a multi-core job. Whole-node scheduling allows for optimization of the data/workflow management (e.g. I/O caching, local merging) but efficient utilization of all scheduled cores is challenging. Dedicated whole-node queues have been setup at all Tier-1 centers for exploring multi-core processing workflows in CMS.

We present the evaluation of the performance scheduling and executing multi-core workflows in whole-node queues compared to the standard single-core processing workflows.

## 1. Introduction

Commodity hardware industry advances in multi-core processors offer large improvements in performance while reducing power and space consumption. Virtually, all computing resources of the worldwide Computing Grid for the LHC (WLCG [1]) incorporate multi-core CPUs.

The current model by which High Energy Physics (HEP) utilize multi-core CPUs is very simplistic. Event level processing parallelism is exploited and one independent application process per core is launched, each processing independent sets of events. The local batch schedulers at the sites are configured correspondingly to schedule independently (and incoherently) individual "jobs" on each core.

This simple model has been very effective and has clear benefits. However, as the number of cores per CPU increases with each new CPU generation, including continual increases in the total amount of physical memory needed per box, and ever increasing number of readers and writers (to local disk and/or remote storage) will not scale sufficiently well in memory usage.

RAM available at the worker nodes is a limitation. Most deployed worker nodes in WLCG have 2 GB RAM per core. The event processing algorithms of the LHC experiments require large amounts of RAM. The deployed hardware might soon not be able to satisfy the job memory needs per core in the current single-core processing model in HEP. Given the increased complexity of the events recorded by the LHC experiments in 2012 (a larger number of collisions per event due to the higher luminosity of the LHC collider), the event processing programs are straggling to keep below the 2 GB limit. Already in 2011, the CMS experiment [2] could not utilize all deployed slots in the Tier-0 center (where the data taken by the detector are promptly reconstructed) because of memory limitations.

Multi-core aware applications can improve memory sharing and processing performance. Multi-core processing jobs share common data in memory, such us the code libraries, detector geometry and conditions data, resulting in a much lower memory usage than standard single-core independent jobs. The High Energy Physics processing is naturally parallelizable. Recorded events can be processed in parallel within one single instance of the processing application.

A further motivation to evolve the processing model into running multi-processing jobs in multi-core nodes is the large decrease of the number of jobs the experiment job management system has to handle. CMS runs about 200k jobs/day whose management requires significant hardware and human resources. The deployed nodes in WLCG typically have 8 cores, therefore the number of jobs could be reduced about one order of magnitude by running in multi-core mode.

The experiment will be billed by all the cores allocated to the jobs, so it will be important to effectively utilize the multi-core architecture, keeping busy all the cores used to avoid inefficiencies.

Multi-processing 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 set up by the parent process is shared between the child processes.

CMS has incorporated support for multi-core processing in its event processing framework [3] by means of the forking mechanism [4]. In this paper we will evaluate the performance of scheduling and executing multi-core workflows compared to the standard single-core processing workflows. CMS is also investigating the threading approach (see [5]).

## 2. Implementation of Multi-core processing in CMS

The CMS offline data processing application has been updated to accommodate the multi-processing forking approach. The parent process forks a number of child processes which share the parent's memory pages. If a child attempts to write to a memory page it shares with its parent the operating system will make a copy of that memory page and give sole ownership of that copy to the child. Therefore to maximize memory sharing between children the parent process needs to load into memory often used, nonvolatile data such as geometry, calibrations or other conditions data.

When the parent process starts it first reads the configuration, loads the shared libraries of the processing modules, pre-fetches geometry, calibrations and other conditions data and forks the child processes. Either a fixed number of children can be specified in the configuration or the parent process can discover in real time the number of cores available in the node and use all of them. Each child processes a portion of the events in the input files. The child processes share the parent's memory pages as long as they remain unmodified. When the child processes finish the job wrapper merges the output files into bigger files and stages them out into mass storage.

This simple approach, where child processes read and write events independently, incurs in some small overheads with respect to single-core processing. The number of events to process by each child is distributed evenly and set up up-front. Given that the time to process an event is not fixed but it is subject to fluctuations depending on the complexity of the event, not all child processes will finish simultaneously and all cores but one will be idle for some time until all child processes are done. We call this effect child processing dispersion. As it will be shown later, we have measured this inefficiency to be quite small.

When all child processes finish, their output files are merged by the job wrapper into larger files and staged out into mass storage. The merge and stage-out processes keep the cores idle for some (small) time. It should however be noted that all CMS data processing workflows incorporate an asynchronous merging step through special merge jobs that merge output files from the processing step into larger files. This is done to avoid having small files that cause overheads in the data bookkeeping and transfer systems. These merge jobs are submitted in the same way as processing jobs, but its only task is reading the unmerged files from the mass storage (stored by the processing jobs) merging them in the local disk of the node and stage out the merged file into mass storage. The local merging done in the multi-core processing operation mode largely minimizes the asynchronous merging step saving time in the overall workflow execution. Running local merging in the multi-core jobs is an implementation choice. If the overhead introduced by local merging were significant, it could be safely skipped or parallelized.

Processing dispersion and merging overheads could be overcome by using a more complex multi-processing scheme. For example, child processes could read input events from a common buffer fed by a reader process and could write processed events into a buffer managed by a writer process. That would involve more complex changes to the CMS framework which are not worth the effort given that the overheads are small as will be shown in the next sections.

Finally, another source of small overhead with respect to the traditional single-core processing is the memory consumed by the parent process, typically of the order of 1 GB. The more cores are utilized the less important this overhead is.

## 3. Multi-core scheduling in CMS

Exploiting the new multi-core processing model requires a new model in computing resource allocation, departing from the standard single-core allocation for a job. The experiment job management system needs to have control over a larger quantum of resource since multi-core aware jobs require the scheduling of multiples cores simultaneously.

The CMS workload management system [6] has been upgraded to incorporate the forking multi-core processing mode. In the workflow definition there is a parameter to specify if the processing jobs should be run in single-core or multi-core mode. In the latter case the job wrapper is prepared to execute the data processing application in multi-core mode, to merge the output files produced by the child processes and to stage out the merged files and register them in the CMS data management system. Multi-core processing jobs are currently submitted to dedicated multi-core queues at the sites that give access to multi-core resources.

CMS has explored two approaches for multi-core allocation: i) Dedicated whole-node queues with dedicated whole-nodes behind where all cores of a node are allocated to a multi-core job; ii) Dedicated queues that allocate to the multi-core job a fixed number of cores (not necessarily the whole node) from a shared farm where single and multi-core jobs can be run. The use of dedicated whole-node resources was the initial approach in CMS. It was started in the context of the WLCG whole-node task force. It allows the experiment to control the whole node. Dedicated queues with few multi-core nodes behind were setup at all 7 CMS Tier-1 centers for testing. The idea was to share with the other LHC experiments these dedicated resources. Sites felt that partitioning resources was not efficient. Thus, the next approach was to provide access to multi-core slots from the shared farms via dedicated queues. These special queues provide slots with a fixed number of cores allocated to the jobs. A more dynamic resource allocation, where jobs get allocated a dynamic number of cores, requires changes both in the experiment job management system and the local resource management system at the sites. This approach will be pursued in the near future. It is in line with the recommendations of the WLCG Workload Management Technology Evolution Group.

## 4. Multi-core processing and scheduling performance

In this section we evaluate the performance of running multi-core data processing jobs measuring memory gains and processing inefficiencies. The results are compared with the case of running the same data processing workflow with standard single-core jobs. The performance change as a function of the number of cores utilized in parallel is also studied.

Figure 1 shows CPU, network and memory utilization of a typical CMS 8-core data reconstruction job, immediately followed by another job. At steady event processing, the 8 cores are fully utilized (upper graph). When the child processes finish processing the input events, the CPU utilization falls down sharply indicating that the processing dispersion is small (in this example all child processes finished within 1 minute). The merging step of the child processes output files can be seen as the small peak in the CPU graph. In this example is takes about 5 minutes. When the merging is done the merged files are staged out into mass storage (blue peak of 60 MB/s in the middle graph). The stage-out step in this example takes around 2 minutes. The green line of the network graph corresponds to the incoming traffic generated reading the input events from the mass storage at a rate of about 5 MB/s. The multi-core job memory utilization is shown in the lower graph. The memory used by the 8-core job (in blue) is about 9 GB.

In order to qualitatively compare the performance between multi-core and single-core jobs, in figure 2 CPU, network and memory utilization of the node are plotted for the case of filling up the very same 8-core machine with 8 simultaneous single-core jobs processing the same set of events. As can be seen in the memory graph (right-most), the memory utilized by the 8 independent single-core processes is about 12 GB, about 25% higher than the case of running one single 8-core job. The machine, equipped with 16 GB RAM, even uses some swap disk space (in purple in the graph) which could degrade processing performance. No CPU inefficiency due to local merging happens for single-core jobs. The small dip in the CPU utilization corresponds to a job finishing and staging out its output files.

Data processing jobs can be configured either to stage in the input files into the node local disk before starting the processing or to directly read the input events from mass storage. In multi-core mode, the former case is not optimal, as can be seen in figure 3. The child processes try to stage in at the same time the output files hammering the local disk, causing large I/O wait (in orange in the CPU plot) which results in a self-inflicted start-up overhead. Thanks to the significant improvements in the I/O layer of the CMS data processing framework, reading files directly from storage does not cause any inefficiency anyway, so there is no problem in running multi-core jobs reading input files directly from the storage.

Given that large portions of physical memory are typically shared by processes, the standard measure of memory, the resident set size (RSS), significantly overestimates memory usage. The Proportional set size (PSS) instead measures each application's "fair share" of each shared area to give a realistic measure. The PSS of a given process (or sub-process of a multi-core process) depends on the other processes running. The CMS framework measures the PSS value of each sub-process at the peak RSS value. This is a good indicator of memory consumption by the application.

Figure 4 shows the RSS and PSS consumption per core as a function of the number of cores utilized by the multi-core application. The data point corresponding to number of cores equal to 1 has been measured filling the multi-core node with standard single-core data processing

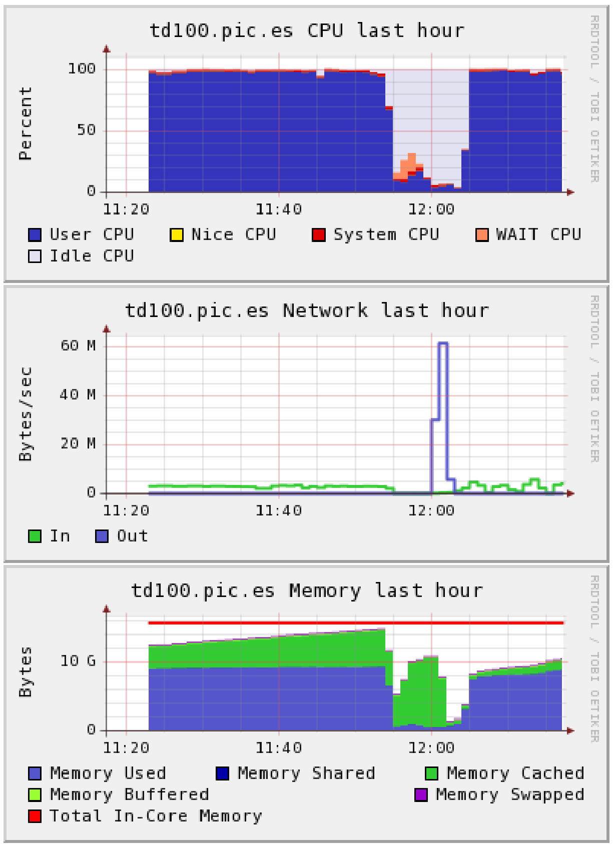

**Figure 1.** CPU, network and memory utilization of a typical 8-core job.

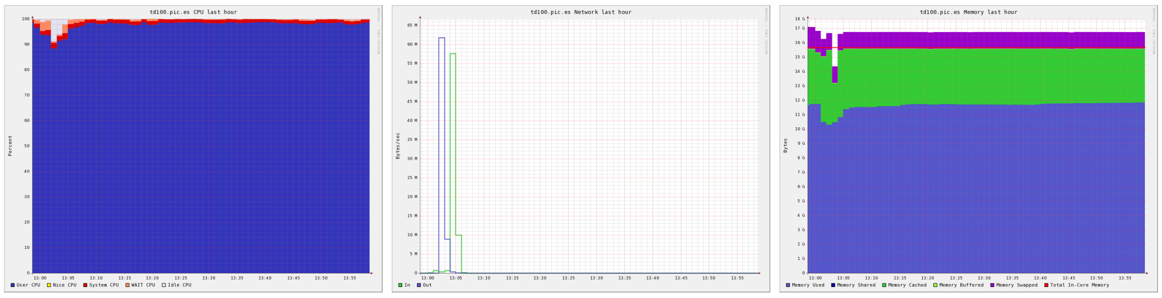

**Figure 2.** CPU, network and memory utilization of single-core jobs filling up a multicore node.

applications. As can be seen in the plot, even if the single-core applications are independent, some small fraction of the memory used is shared between them (e.g. shared libraries).

As expected, the RSS consumption is constant with the number of cores used by the application since it does not reflect the memory sharing between the child processes. However, the gain in PSS is significant. Already with 8-core multiprocessing jobs, the memory gain is about 25%. Note that the parent process also consumes some RAM (about 1 GB) which has to be taken into account when calculating the gain in memory usage. For 24-core jobs the gain is about 40%. As the number of cores increases further, the memory gain stabilizes since only a fraction of the child processes memory can be shared (geometry, calibrations and conditions

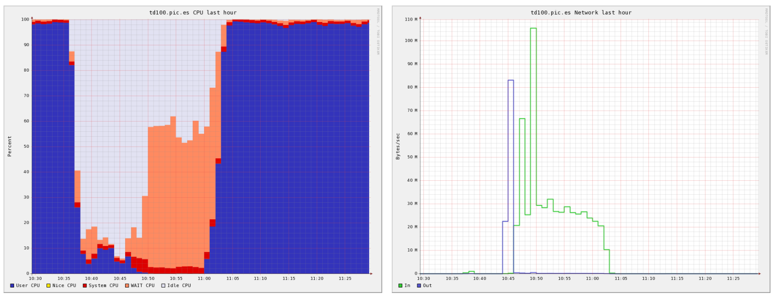

**Figure 3.** CPU and network utilization of a multi-core job when the input files are staged-in into the local disk at the beginning of the job.

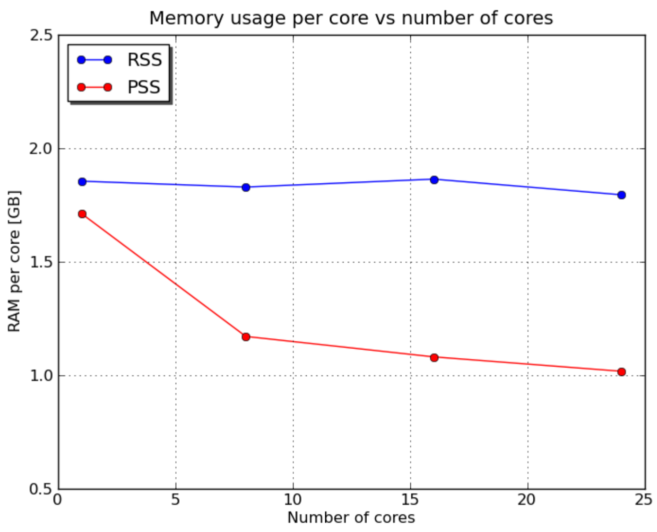

**Figure 4.** RSS and PSS memory utilization per core as a function of the number of cores used by the application.

data).

As mentioned before, the multi-core processing mode incurs in some small CPU inefficiencies since due to data processing dispersion, file merging and stage-out the cores of the node stay idle for some time. We show those inefficiencies in figure 5 for the case of 8-core jobs as a function of the job duration.

The processing dispersion, calculated as 1 - (sum of children processing time)/[(max child processing time)*(number of children)], is quite small (below 1%) and its relative importance decreases with the duration of the job and becomes negligible.

The merging inefficiency is the largest contribution to the overall overhead, about 5%. While output files are being merged, all cores but 1 stay idle. The merging overhead stays constant with the job duration since the merging time is roughly proportional to the size of the files to be merged and the size is proportional to the job duration. The overhead caused by the stage-out of the output files behaves in the same way. It contributes below 2% to the overall overhead. The time to stage out the merged files into mass storage is roughly proportional to their size (assuming a constant stage-out throughput).

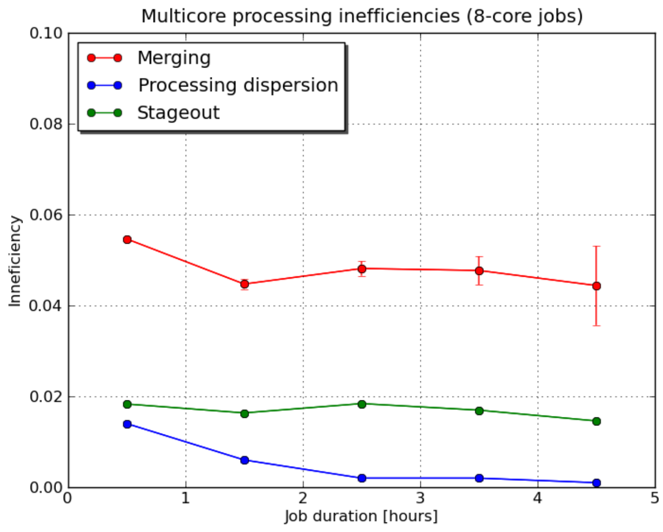

**Figure 5.** Overhead caused in a multi-core job by event processing dispersion, merging and stage-out of output files, as a function of the job duration for the case of 8-core jobs.

The overall multi-processing overhead is plotted in figure 6 adding all contributions for 8 and 16-core jobs separately. For sufficiently long jobs, longer than 2 hours, the overall overhead is about 6% roughly independent of the job duration and the number of cores used. When the application uses a larger number of cores, the overall overhead is somewhat larger due to the higher processing dispersion. This difference is only noticeable for short jobs, since for longer jobs the processing dispersion becomes negligible.

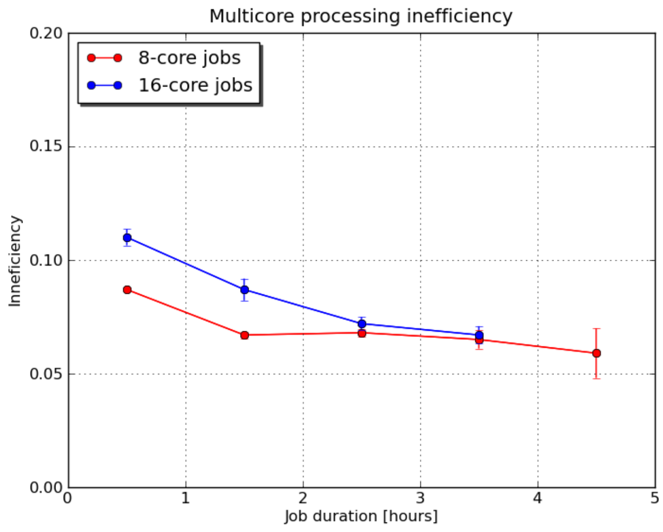

**Figure 6.** Global overhead (considering together the effects of event processing dispersion, merging and stage-out) for 8-core and 16-core jobs as a function of job duration.

At steady processing, the event processing throughput (number of events processed per second and per core) is not degraded in multi-core processing mode, as can be seen in figure 7.

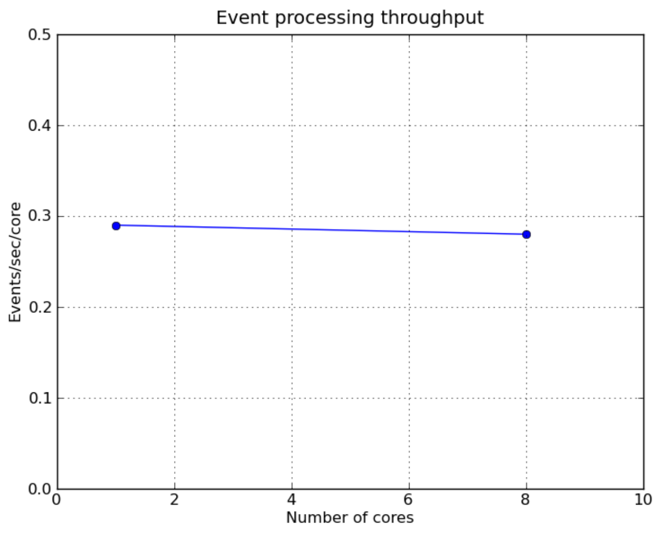

**Figure 7.** Event processing throughput (events/sec/core) as a function of the number of cores used by the multi-core application.

The application CPU efficiency, defined as the CPU time over the wall-clock time employed by the application, is similar for the single and multi-core cases, and very close to 100% as can be seen in figure 8.

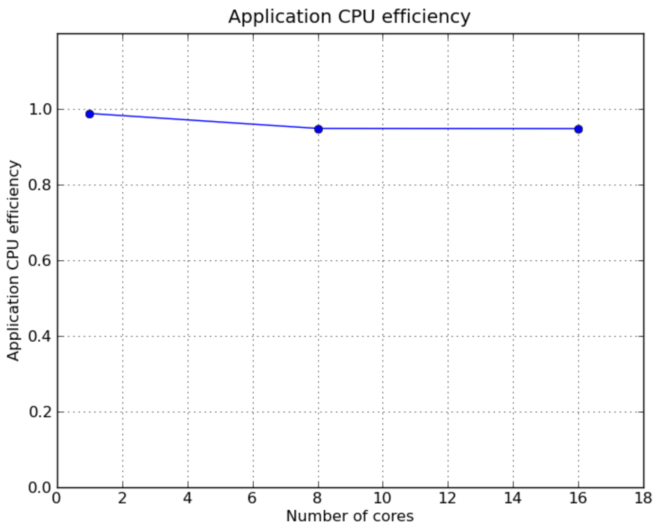

**Figure 8.** Application CPU efficiency (cpu time over wall-clock time) as a function of the number of cores used by the application.

## 5. Outlook and conclusions

The CMS data processing framework and workload management system fully support multi-core scheduling and execution in forking mode. This processing mode has been extensively tested but only at a modest scale. There exist dedicated whole-node queues at the 7 CMS Tier-1 centers with few multi-core nodes behind, and a couple of dedicated multi-core queues (at Purdue University and at the Karlsruhe Institute of Technology -KIT-) which give access to a larger number of multi-core nodes shared with other production activities. During the testing phase we have run of the order of 10k jobs, up to 100 running concurrently. CMS plans to significantly increase the amount of resources available for multi-core processing at the Tier-1s. Those resources will be shared with standard single-core workflows and the local batch system will allocate a number of cores to the multi-core jobs.

There is also a potential gain by moving the processing at the Tier-0 center at CERN to multi-core mode. As mentioned before, in 2011 only 6 out of 8 cores of the Tier-0 nodes could be used due to memory limitations. The multi-core mode will be pursued at the Tier-0.

Multi-core processing results in a memory utilization reduction of about 25% for 8-core child processes (40% for 24 children). This gain is very important in order to keep the reconstruction application memory footprint below 2 GB/core.

The current implementation of the the multi-core forking processing mode incurs in a small CPU inefficiency of about 6%, independent of the job duration and number of child processes for long enough jobs (longer than about 2 hours). The overhead is essentially due to the merging of the output files from each sub-process. The merging step could be skipped or parallelize in order to remove this overhead. On the other hand, the local merging done in the multi-core processing operation mode largely minimizes the asynchronous merging step (present in all CMS production workflows) saving time in the overall workflow execution.

CMS is ready to go multi-core for data processing workflows.

**Acknowledgments**

We thank the CMSSW developers for implementing the multi-core processing mode in the data processing framework and CMS site administrators for putting in place the multi-core queues.

## References

[1] Worldwide LHC Computing Grid. Technical Design Report. E-ref: http://lcg.web.cern.ch/LCG/TDR/LCG TDR v1 04.pdf
[2] S Chatrchyan et al, The CMS collaboration, The CMS experiment at the CERN LHC. JINST 3 S08004, 2008.
[3] C D Jones et al., The New CMS Event Data Model and Framework, Proceedings of the International Conference on Computing in High-Energy Physics and Nuclear Physics, Mumbai,India (2006).
P Elmer et al., 2010 J. Phys.: Conf. Ser. 219 032022.
[4] C D Jones et al., Multi-core aware applications in CMS, 2011 J. Phys.: Conf. Ser. 331 042012
[5] C D Jones et al., Study of a Fine Grained Threaded Framework Design. These proceedings.
[6] S Wakefield e al., The CMS Workload Management System. These proceedings.