Preparing the Gaudi framework and the DIRAC WMS for multicore job submission

N Rauschmayr 1, A Streit 2

1 CERN - European Organization for Nuclear Research, Geneva, Switzerland
2 Steinbuch Centre for Computing, Karlsruhe Institute of Technology, Karlsruhe, Germany
E-mail: nathalie.rauschmayr@cern.ch

Abstract. HEP applications need to adapt to the continuously increasing number of cores on modern CPUs. This must be done at different levels: the software must support parallelization, and the scheduling has to differ between multicore and singlecore jobs. The LHCb software framework (GAUDI) provides a parallel prototype (GaudiMP), based on the multiprocessing approach. It allows a reduction of the overall memory footprint and a coordinated access to data via separated reader and writer processes. A comparison between the parallel prototype and multiple independent Gaudi jobs in respect of CPU time and memory consumption will be shown. Furthermore, speedup must be predicted in order to find the limit beyond which the parallel prototype (GaudiMP) does not bring further scaling. This number must be known as it indicates the point, where new technologies must be introduced into the software framework. In order to reach further improvements in the overall throughput, scheduling strategies for mixing parallel jobs can be applied. It allows overcoming limitations in the speedup of the parallel prototype. Those changes require modifications at the level of the Workload Management System (DIRAC).

1. Introduction

The Worldwide LHC Computing Grid (WLCG) [1] offers a high variety of computing resources, which mostly consist of multicore CPUs. As the number of cores increases on future CPUs, concurrent accesses to system resources like network and filesystem will play a major role. Furthermore, total memory capacity must grow, which leads accordingly [2] to much larger power consumption and costs due to memory than due to processing units. This influences the memory balance negatively, thus memory ratio (memory per core) cannot remain the same. According to [2] memory ratio will decrease from 0.5 bytes of DRAM memory per peak flop to 0.1 bytes per peak flop on future many core systems. However, memory represents already a restrictive resource for many LHCb jobs nowadays, as WLCG limits it currently to 2 GB per process. Consequently, jobs start swapping and lose performance. Applications must be adapted in order to overcome those limitations. Parallelization can help to decrease the overall memory footprint of applications via sharing and to manage concurrent accesses to system resources. In this paper the parallel prototype for the Gaudi framework of the LHCb experiment is presented [3]. Section 2 will give an overview of the implementation and will compare it with multiple independent Gaudi jobs in respect of CPU time and memory. Speedup, the most important metric in parallel computing, will be evaluated in the context of the parallel prototype within section 3. Executing jobs with more than one process requires changes at the level of scheduling.
Optimization can also take place at this level by mixing jobs in an appropriate way. This will be evaluated in section 4.

2. Parallel prototype (GaudiMP)

The following subsections describe the parallel prototype GaudiMP in detail. Furthermore, a comparison with multiple independent Gaudi jobs will be given in respect of throughput and memory consumption. The speedup reached within GaudiMP will be shown as well.

2.1. Implementation overview

GaudiMP is the parallel prototype of the Gaudi framework, which is based on the multiprocessing approach [4]. As it is implemented in python, processes must be used instead of threads. This is due to the Global Interpreter Lock (GIL) which is implemented as a mutex and does not allow a parallel execution of python threads [5]. GaudiMP spawns several sub processes: one reader and writer process, which coordinate access to files, and one main worker which forks n subworkers. Via a shared queue the worker processes can receive events from the reader. Those events are stored as ROOT TBuffer objects and must be therefore serialized and deserialized. This is necessary as those objects cannot be transmitted via shared memory. This due to the fact that ROOT objects use virtual functions. If an object is created by process 1 in shared memory the virtual table points to the virtual address space of process 1, which is not shared. Consequently, process 2 cannot access the information to which the virtual table points. Instead, objects must be serialized and this is a time intensive task. One reconstruction job for example processes around 50k events, which must then be serialized and deserialized between reader and worker and between worker and writer. This represents the main limitation in GaudiMP.

The main reason for using a parallel job model is certainly the reduction of the overall memory footprint. To provide a better memory sharing, automatic tools like KSM can be applied or forking of child processes must be postponed as late as possible. Both concepts are based on the copy-on-write principle (COW) of Linux, which keeps a page in shared memory as long as no process writes to it [6]. Within GaudiMP several approaches have been evaluated and combined [7]. The current version supports forking of child processes after the initialization but before the first event (figure 1). This allows the sub processes to share all configurations and datasets, which have been loaded during the initialisation period. If sub processes are forked after a few events, histograms and counters are filled and have to be reset after the fork. This is avoided in the current implementation, because it forks sub processes before the first event.

Furthermore the concept of reader and writer processes changes in the simulation software Gauss, which does not require any input files. Instead of reading files random seeds are generated on the reader process, which guarantees reproducibility and independent sets of random numbers. The generation of random seeds is based on run and event number. If random seeds are generated within the workers, it must be guaranteed that each worker generates different event numbers. Instead, this is done by the reader process which forwards the random seeds to the worker processes.
2.2. Comparison of GaudiMP with multiple Gaudi jobs

In the current job model one process represents an independent instance of the Gaudi framework and no sharing of datasets is applied. As soon as the jobs reach the memory limit, they start swapping. The following test determines, up to which point GaudiMP scales better than the current model. In the test each job has processed the same amount of events in total. The GaudiMP job has processed all events with 8 worker processes, while in the other case 8 separate Gaudi instances have been initialised and each of them has processed 1/8th of the input file. The memory limit has been continuously decreased, in order to evaluate the value, beyond which the parallel prototype provides an overall better throughput. As shown in figure 2, the difference in event throughput (events per second) does not differ significantly, as long as the limit is larger than 1 GB. This means that overhead due to serialization and deserialization does not have a significant impact. When memory is limited to 1 GB per process a large difference can be observed. As GaudiMP lowers the overall memory footprint via sharing, it can still provide the same throughput with 1.1 events per second in the stripping job and with 0.41 events per second in the reconstruction job. The value decreases to 0.35 events per second compared to 0.23 in the reconstruction job.

| Memory limit in GB | Throughput in evt/s |
|--------------------|---------------------|
| NoLimit            | 1.1                 |
| 2GB                | 1.0                 |
| 1.5GB              | 0.9                 |
| 1GB                | 0.8                 |
| 500MB              | 0.7                 |

![Figure 2. Throughput depending on different memory thresholds](image)

3. Evaluation of the parallel prototype

3.1. Improving the speedup of GaudiMP

Total time indicates the time from initialising and finalising the prototype. It takes into account all serial parts, like spawning the sub processes, synchronization, writing to disc and finalizing the processes. The run time measures only the processing part inside the event loop. It excludes synchronization and all serial parts. Run time should show a linear speedup as it represents the parallel part of GaudiMP. Nevertheless the memory footprint increases, which can lead to an increased number of page faults, cache misses and communication overhead increases. This influences speedup negatively. Enlarging the number of events diminishes the gap between speedup curves of total and run time, as the impact of serial parts becomes smaller (figure 3). However, the speedup curve of run time presents the theoretical upper limit, which is never reached in practice.

3.2. Prediction of speedup

Speedup can be predicted and very often Amdahl’s Law is used to define the speedup curve [8]. It splits a program in a parallel and serial part and indicates that speedup is limited due to the part, which cannot be parallelized at all. In most cases such a decomposition is not trivial and consequently a different model must be applied. The Downey speedup model is based
on parameters, which indicate the average and the variance of parallelism [9]. It distinguishes between a high and a low variance model, where high variance is larger than 1. This paper focuses on the low variance model, as the parallel prototype of Gaudi fits to this category. The low variance model defines speedup as:

\[
S(n) = \begin{cases} 
\frac{A^{n}}{A + \sigma(n-1)/2} & 1 \leq n \leq A \\
\frac{\sigma(A-1/2)+n(1-\sigma/2)}{A} & A \leq n \leq 2A - 1 \\
A & n \geq 2A - 1
\end{cases}
\]

where \(A\) is the average parallelism, \(\sigma\) the variance in parallelism and \(n\) the number of used cores. Parameter \(A\) indicates the point, after which an application will not allow further scaling. Knowing this value is important as it indicates the point, when cores cannot be used efficiently any longer and when other technologies must be introduced into the software framework.

In order to evaluate the parallel prototype of Gaudi, it is worth determining the parameters \(A\) and \(\sigma\) for the case that an infinite number of events is provided. Due to synchronization and serialization of objects, which is a function of the number of events, the speedup curve of the total time can never be the same like the one for run time. In order to obtain the values \(A\) and \(\sigma\), the run time of jobs running with different number of worker processes is measured. Curve fitting is then applied on those values. Figure 4 shows the predicted speedup curves. The average parallelism is 43.0 for reconstruction, 21.93 for simulation, and 29.52 for stripping. It means that simulation jobs can scale up to 21 cores. The results indicate, that new parallelization concepts must be applied, as soon as a simulation job shall run with more than 20 cores.

4. Scheduling of moldable jobs
In order to use modern CPUs more efficiently, the scheduling of parallel jobs must be also taken into account. As shown in the previous sections, the parallel prototype does not scale linearly with the number of cores. Beyond a certain degree of parallelism it is more reasonable to add an additional job instead of assigning more cores to one single job. In this context different methodologies can be applied, in order to define the degree of parallelism for each job. Therefore the current prototype of parallel Gaudi can be assigned to a moldable job model. It means, that a scheduler has to assign an appropriate number of cores and this number will not vary during the runtime of a job [10]. The basic idea of a scheduler doing this is to increase the overall job throughput (number of events processed within a given time period). A scheduler must take
Figure 4. Average parallelism and variance in parallelism for reconstruction, simulation and stripping jobs

characteristics of the worker node into account, like current workload, hardware configuration and scaling behaviour. On top of that, runtime and speedup of jobs must be estimated by the scheduler. The minimal degree of parallelism of each job is determined by its memory demand and the memory capacity provided by the system. The larger the degree the better is the sharing and the lower is the required memory per process.

5. Outlook

As presented in this paper, parallelization can help to significantly reduce the memory footprint of LHCb applications. On the other hand scheduling becomes more complex because one must take into account the additional parameter of number of cores requested for a job. As a result, the Workload Management System must provide mechanisms to schedule multicore jobs in an efficient way. The Workload Management System must also be able to obtain knowledge about the scaling behaviour of different software versions and about required run time of different job types. Such statistics can be automatically gathered, since a large amount of jobs is processed every day. Furthermore, the speedup of an application is always influenced by the scaling behaviour of a machine due to different hardware configurations. This must be also respected in the decision making.

Finally, optimization of LHCb jobs can be applied at several levels: different parallelization concepts can be applied at the level of software. Scheduling can help to overcome limitations in speedup by mixing jobs in an appropriate way. On top of that tools provided by the operating system (like KSM [7]) allow further improvement in memory reduction.

References

[1] 2012 WLCG Worldwide LHC Computing Grid URL http://wlcg.web.cern.ch/
[2] Shalf J, Asanovic K, Patterson D, Keutzer K, Mattson T and Yelick K 2009 SciDAC Review URL http://www.scidacreview.org/0904/html/multicore.html
[3] Corti G, Cattaneo M, Charpentier P, Frank M, Koppenburg P, Mato P, Ranjard F, Roiser S, Belyaev I and Barrand G 2006 Nuclear Science, IEEE Transactions on 53 1323–1328 ISSN 0018-9499
[4] Mato P and Smith E 2010 Journal of Physics: Conference Series 219 042015 URL http://stacks.iop.org/1742-6596/219/i=4/a=042015
[5] David Beazley 2010 Understanding the Python GIL (PyCon 2010)
[6] Bovet D and Cesati M 2008 Understanding the Linux Kernel (O’Reilly Media) ISBN 9780596554910
[7] Rauschmayr N and Streit A 2012 Reducing the memory footprint of parallel applications with ksm Facing the Multicore-Challenge III (Lecture Notes in Computer Science vol 7686)
[8] Krishnaprasad S 2001 J. Comput. Sci. Coll. 17 288–293 ISSN 1937-4771 URL http://dl.acm.org/citation.cfm?id=775339.775386
[9] Downey A B 1997 A model for speedup of parallel programs Tech. rep. Berkeley, CA, USA
[10] Cirne W and Berman F 2001 A model for moldable supercomputer jobs Parallel and Distributed Processing Symposium., Proceedings 15th International pp 8 pp.– ISSN 1530-2075