Journal of Physics: Conference Series **664** (2015) 042044 doi:10.1088/1742-6596/664/4/042044

Multi-threaded Object Streaming

Salvatore Di Guida[1,3], Giacomo Govi[2], Miguel Ojeda[3], Andreas Pfeiffer[3], Roland Sipos[4,3] on behalf of the CMS collaboration

[1] Università degli Studi "G. Marconi", Via Plinio 44, Roma, Italy and INFN Sezione di Napoli
[2] Fermi National Accelerator Laboratory, Batavia, IL 60510-5011, USA
[3] CERN, CH-1211 Geneve 23, Switzerland
[4] Eötvös Loránd University, Faculty of Informatics, H-1117 Budapest, Hungary

E-mail: andreas.pfeiffer@cern.ch

Abstract. The CMS experiment at the Large Hadron Collider (LHC) at CERN, Geneva, Switzerland, is made of many detectors which in total sum up to more than 75 million channels. The detector monitoring information of all channels (temperatures, voltages, etc.), detector quality, beam conditions, and other data crucial for the reconstruction and analysis of the experiment's recorded collision events is stored in an online database. A subset of that information, the "conditions data", is copied out to another database from where it is used in the offline reconstruction and analysis processing, together with alignment data for the various detectors. Conditions data sets are accessed by a tag and an interval of validity through the offline reconstruction program CMSSW, written in C++. About 400 different types of calibration and alignment exist for the various CMS sub-detectors.

With the CMS software framework moving to a multi-threaded execution model, and profiting from the experience gained during the data taking in Run-1, a major re-design of the CMS conditions software was done. During this work, a study was done to look into possible gains by using multi-threaded handling of the conditions. In this paper, we present the results of that study.

1. Introduction

The primary goal of the Compact Muon Solenoid (CMS) experiment [1] is to explore physics at the TeV energy scale, exploiting the collisions delivered by the Large Hadron Collider (LHC) at the European Organisation for Nuclear Research (CERN) in Geneva, Switzerland. The central feature of the CMS apparatus is a superconducting solenoid, of 6 m internal diameter, delivering a field of 3.8 Tesla. Within the field volume are the silicon pixel and strip tracker, the crystal electromagnetic calorimeter (ECAL) and the brass-scintillator hadronic calorimeter (HCAL). Muons are measured in drift tube chambers (DT), resistive plate chambers (RPC), and cathode strip chambers (CSC) embedded in the steel return yoke. All detectors combined comprise more than 75 million channels, most of them to be found in the tracker system. A detailed description of the experimental apparatus can be found elsewhere [1].

Calibration and alignment data are fundamental to maintain the design performance of the experiment. Dedicated, very fast, workflows have been put in place to compute and validate the alignment and calibration sets and insert them in the conditions database before the reconstruction process starts. Some of these sets are produced by analysing and summarising the parameters stored in the online database. Others are computed using event data through a special express workflow.


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

2. Conditions Data in CMS experiments

During Run 1 of the Large Hadron Collider (LHC) at CERN (2009-2013), the CMS experiment recorded and analysed hundreds of petabytes of data, resulting in the discovery and publication of many new physics results including the discovery of the Higgs boson in July 2012.

Achieving these results required extensive processing and analysis of massive datasets in a largely distributed computing environment. In addition to event data stored in distributed file systems, other (non-event) data (so-called "conditions data") related to the detector state and data taking process are essential at nearly every stage of data processing and analysis. These conditions data are stored in an Oracle RDBMS, provided and administered by CERN IT. A REST based, hierarchical access layer (Frontier [2]) which, by caching the results of the queries, allows an efficient access to the conditions data from the thousands of jobs processing the data on the LCH Computing Grid [9].

The CMS conditions data are logically grouped in **Tags**, containing related conditions data from (parts of a) detector subsystem. A conditions **payload** is valid for a specific **IOV**, the "Intervals of Validity" (an interval in time, run-number or a combination of run- and luminosity-number). A **Tag** contains the relations between the **payloads** and the various **IOVs** for which they are valid. A coherent set of unique **Tags** is called a **Global Tag**, these are the central "entry-points" for the reconstruction and analysis code when accessing the conditions.

3. Payloads in the Re-designed Conditions Code

After a review of the scalability and use cases and with a desire to simplify the architecture, CMS re-designed and re-implemented the handling of the conditions data [3]. One of the main changes in the new design was the move away from the structured way the payloads were stored (using object-relational mappings) to an opaque storage of payloads. These are now serialised into a "BLOB" object in the C++ code and the new conditions handling code treats them as "opaque" objects, easing the handling of the conditions significantly.

3.1. Automatic generation of serialisation code

Recent developments in compiler related tools, *libclang* and its Python binding *pyclang*, allowed the creation of a small (about 500 lines of code overall) script which automatically generates the (de-)serialisation code from the C++ header files of the user-defined payloads. This script is called by the CMS code configuration and build tool (SCRAM).

A simplified example of a user-defined payload is given here:

```
#ifndef CondFormats_PhysicsToolsObjects_Histogram_h
#define CondFormats_PhysicsToolsObjects_Histogram_h

#include "CondFormats/Serialization/interface/Serializable.h"

template<typename Value_t, typename Axis_t = Value_t> class Histogram {

   // persistent members
   std::vector<int> bins;

   // others ...

   // transient cache variables
   mutable Value_t total COND_TRANSIENT; //CMS-THREADING
   mutable bool totalValid COND_TRANSIENT;

   COND_SERIALIZABLE;

}; // end class<> Histogram
```

The user adds the inclusion of the "Serializable.h" header from the CondFormats/Serialization package and flags the class to be considered for serialisation by adding the "COND_SERIALIZABLE" macro to the end of the class(es) to be serialised. The script then parses all header files from all packages, extracts the information on which classes should be serialised and generates the actual code for the serialisation on a per-package basis, which in turn is compiles by the CMS code configuration and build tool (SCRAM) when processing the package. The user can declare any class-members which should not be serialised with the keyword "COND_TRANSIENT", so they will be ignored when parsing the file.

4. Performance study

An initial study to use Boost serialisation package [6], together with a third-party product for the "Archive" [7], showed promising results to store CMS Conditions payloads as "BLOB"s in the database. Single-threaded performance was found to be comparable to (de-)serialisation with ROOT (version 5) [8].

The performance study was done by comparing parallel, multi-threaded loading of the conditions from the DB, followed by parallel, multi-threaded deserialisation of the payload "BLOBs" into the corresponding C++ objects. The comparison was done against the same code in a serial access pattern, both for loading from the DB and deserialisation of the payloads. Write (serialisation) performance of the payloads was not measured as this is a one-time event and not time-critical.

4.1. Single-threaded Payload Deserialisation

In a first step, the performance of the loading of conditions from the Oracle database was studied. Fig. 1 shows the time to load the payloads for all tags of a given Global Tag as a function of the size of the payload. No significant difference is found between the two serialisation types, as expected. The constant time for payloads smaller than about 100 kB suggest that the loading is dominated by overheads in the database and/or network communication.

In a second step, after loading all payloads from the database into memory, the payload data was de-serialised in a single thread. The time to de-serialise each payload as a function of payload size is shown in Fig. 2, where the boost serialisation shows a smooth behaviour as a function of size, exhibiting an about linear correlation between time and size. Towards very small sizes (less than about a few hundred bytes) a flattening of the data is observed, indicating memory allocation overheads. The corresponding data for root deserialisation is less smooth; looking into some of the involved classes it seems as if some optimisation for specific data types are causing the large differences for payload sizes less than about 100 kB. As the overall time of the payload deserialisation is dominated by the larger payloads, the differences in the shape at smaller payloads can be neglected. A simple parameterisation of the boost data ($f = 1.856 \times 10^{-5} * x + 0.035$) is included, though given the locally big deviations, caused by more complex object types, it should be used with caution.

4.2. Multi-threaded Payload Deserialisation

The runs were then repeated with the deserialisation step running in multiple threads where each thread was handling only the deserialisation of one single payload at a time. On the same 24 core machine as for the single-threaded steps, the number of threads handling the deserialisation was successively increased and each run for a given number of threads was repeated three times. Before and after the runs, the environment on the machine was checked to make sure that no significant load was running in parallel on the machine and the database. The average time of the three runs for the deserialisation of all payloads is shown in Fig. 3 as a function of the number of threads used. The small variations are likely due to the limited statistics and fluctuations in network transfer speeds and load on the database.

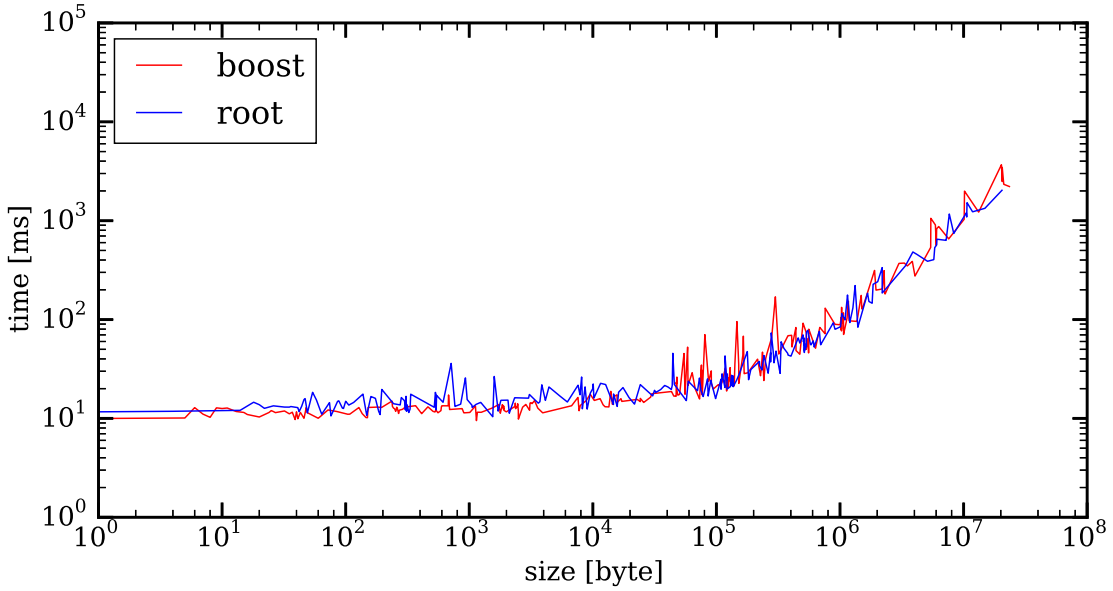

Figure 1. Time (msec) to load conditions from the database as a function of the size of the payload (bytes)

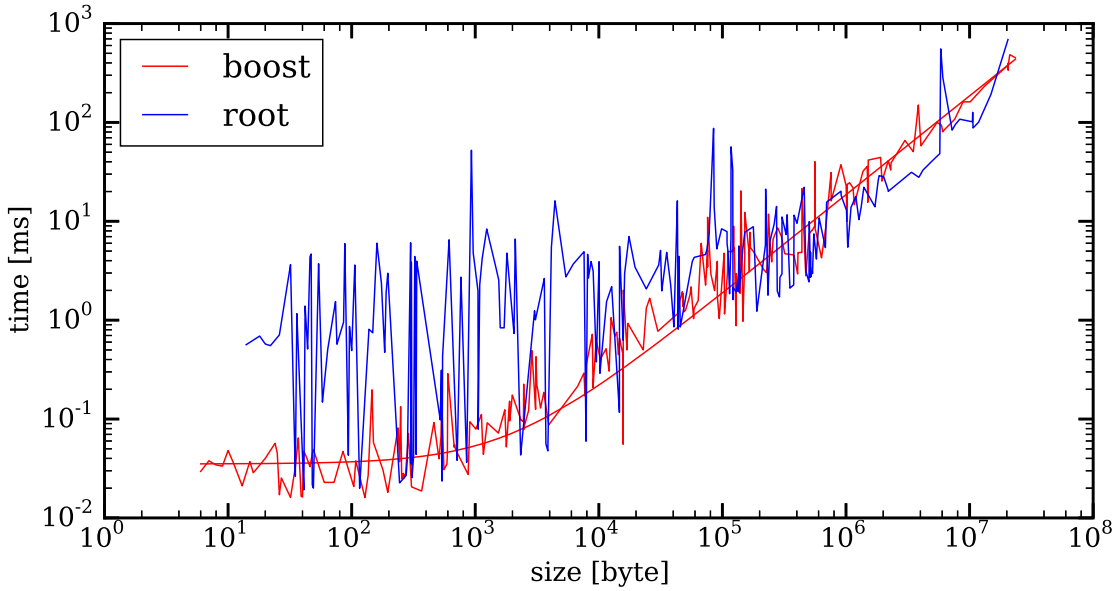

Figure 2. Time (msec) to de-serialise the payloads as a function of the payload size (bytes). A simple linear parameterisation ($f = 1.856 \times 10^{-5} * x + 0.035$) for boost is included.

In Fig. 4 the calculated gain (or speedup) factor is given as a function of the number of threads used. While an almost linear gain (as indicated by the green line) is visible for a very small number of threads, a clear levelling out of performance can be observed from about five

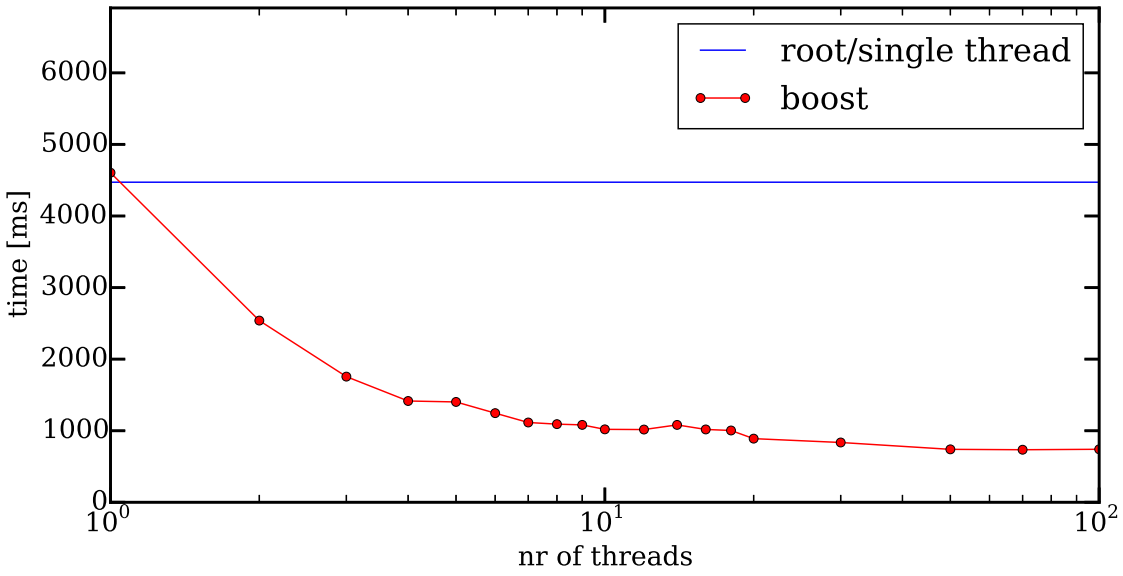

Figure 3. Time (msec) for the (parallel) deserialisation of all payloads as a function of the number of threads used.

Nr. of threads	1	8	16	100	gain 8	gain 16	gain 100
loading from DB	25409	10737	7173	5862	**2.4**	**3.5**	**4.3**
deserialisation	4233	1190	1134	648	**3.6**	**3.7**	**6.5**
overall elapsed	32366	14485	9122	9122	**2.2**	**3.0**	**3.5**

Table 1. Time (msec) and gain/speedup factor of the loading and deserialisation steps for various number of threads. The elapsed time also contains about 2.5-3 sec overhead from other activities (like loading the IOVs).

parallel threads. Still, even at a number of threads larger than the actual number of cores on the machine, a small increase can be observed. The less than linear behaviour is likely due to code and/or memory limitations, as all payloads reside already in memory before the deserialisation is started, excluding any I/O limitations.

Similar to the deserialisation, another run was performed to measure the gain for parallel loading of the payloads from the database. Table 1 summarises the times for loading. deserialising the payloads and the sum of these times for different numbers of threads used. An overall speedup factor of about 3-4 can be achieved for about 8-10 parallel threads.

5. Summary

A performance study done in the context of the new CMS conditions software shows that multi-threaded loading and deserialisation of payloads can achieve a speedup of about a factor of 3-4 for about 8-10 threads, both in loading payloads from the database and in deserialisation.

With the help of compiler related tools which became recently available, a simple script can be used to effectively automate the creation of the (de-)serialisation code, reaching a level of "user-friendliness" at least comparable to the "streamer code" available in ROOT (version 5).

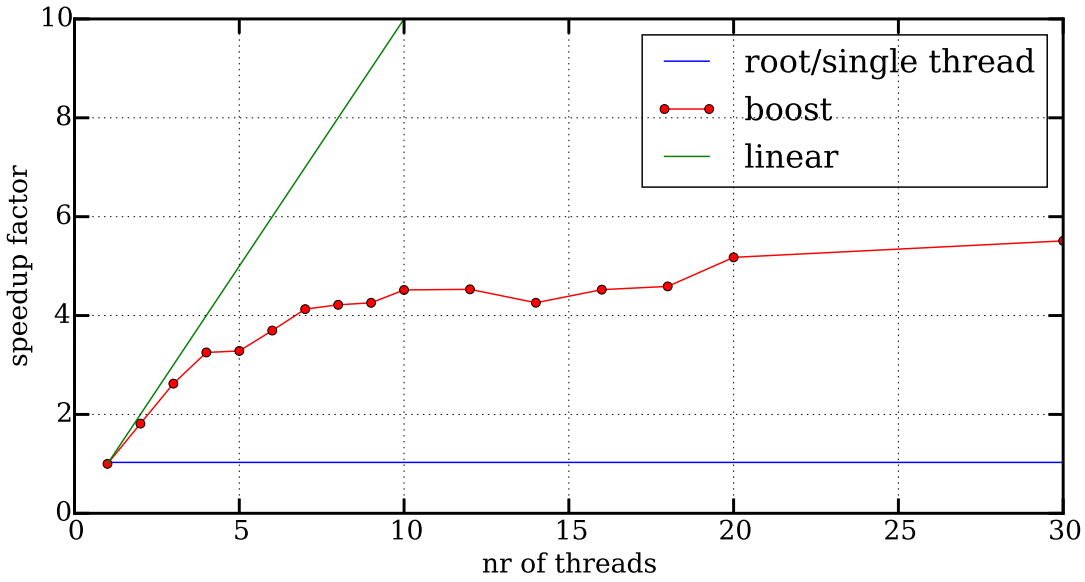

Figure 4. Gain/speedup factor for the (parallel) deserialisation of all payloads as a function of the number of threads used.

6. Acknowledgments

We would like to thank the team of database administrators in the CERN IT department for their excellent work in keeping the performance of the CMS databases at a very high level and for their continued support for the developers of the CMS applications.

References

[1] CMS Collaboration "The CMS experiment at the CERN LHC" (2008) JINST 3 S08004, doi:10.1088/1748-0221/3/08/S08004

[2] Dykstra D "Scaling HEP to Web size with RESTful protocols: The frontier example" (2011) *J. Phys.: Conf. Series* **331** 042008.
`http://iopscience.iop.org/1742-6596/331/4/042008/pdf/1742-6596_331_4_042008.pdf`.

[3] Govi G, Di Guida S, Pfeiffer A, Ojeda M "The CMS Condition Database system" (2015) this conference.
`https://indico.cern.ch/event/304944/session/10/contribution/130`.

[4] clang: a C language family frontend for LLVM
`http://clang.llvm.org`.

[5] Python bindings for Clang
`https://github.com/llvm-mirror/clang/tree/master/bindings/python`

[6] Abrahams D and Gurtovoy A "C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond." (2004) *Addison-Wesley*, November, 2004. ISBN: 0-321-22725-5.
`http://boost.org`

[7] "EOS Portable Archive" (2006)
`https://epa.codeplex.com`.

[8] Brun R and Rademakers F "ROOT - An Object Oriented Data Analysis Framework" (1997) Proceedings AIHENP'96 Workshop, Lausanne, Sep. 1996, *Nucl. Inst. and Meth. in Phys. Res.* **A 389** (1997) 81-86.
`http://root.cern.ch/`

[9] The LCG TDR Editorial Board "LHC Computing Grid: Technical Design Report" (2011) *LCG-TDR-001*, CERN-LHCC-2005-024 ISBN 92-9083-253-3
`http://wlcg.web.cern.ch/`.