Gaudi Evolution for Future Challenges

To cite this article: M Clemencic et al 2017 J. Phys.: Conf. Ser. 898 042044

View the article online for updates and enhancements.

Related content
- The LHCb Turbo Stream
  Sean Benson, Vladimir Gligorov, Mika Antton Vesterinen et al.
- Latest progress from the Daya Bay reactor neutrino experiment
  Zhe Wang and Daya Bay Collaboration
- Recent Results from the Daya Bay Reactor Neutrino Experiment
  En-Chuan Huang
Gaudi Evolution for Future Challenges

M Clemencic¹, B Hegner¹ and C Leggett²

¹ CERN, CH-1211 Geneva 23, Switzerland
² Lawrence Berkeley National Laboratory, 1 Cyclotron Road Mailstop 50F-1650, Berkeley, CA 94720, USA

E-mail: marco.clemencic@cern.ch

Abstract. The LHCb Software Framework Gaudi was initially designed and developed almost twenty years ago, when computing was very different from today. It has also been used by a variety of other experiments, including ATLAS, Daya Bay, GLAST, HARP, LZ, and MINERVA. Although it has been always actively developed all these years, stability and backward compatibility have been favoured, reducing the possibilities of adopting new techniques, like multithreaded processing. R&D efforts like GaudiHive have however shown its potential to cope with the new challenges.

In view of the LHC second Long Shutdown approaching and to prepare for the computing challenges for the Upgrade of the collider and the detectors, now is a perfect moment to review the design of Gaudi and plan future developments of the project. To do this LHCb, ATLAS and the Future Circular Collider community joined efforts to bring Gaudi forward and prepare it for the upcoming needs of the experiments.

We present here how Gaudi will evolve in the next years and the long term development plans.

1. Introduction
The LHCb Software Framework Gaudi[1] was designed around a few core principles

• separation between data an algorithms
• well defined interfaces
• reusable components

Although these principles are still valid, several factors make the current code base not ready for the computing challenges posed by future experiments.

For example, at the time of the conception of Gaudi, computing was mostly relying on single process applications, so no particular attention was posed on multi-threading and thread-safety. With the shift of paradigm of CPU manufacturing from single fast cores towards many slower cores, multi-threading applications are performing better than single process applications, so Gaudi needs to evolve in that direction.

It’s also important to note that Gaudi code complied to the 1998 version of the C++ standard (C++98), but the new versions of the C++ standard (C++11, C++14 etc.) are a great step towards a better language, easier to use, with better performance and that can make developers more productive.

After almost 20 years of backward compatible improvements, using the opportunity provided by LHC second Long Shutdown, we want to move forward, leaving behind legacy code, to take...
advantage of new standards and change design choices that are not considered anymore as best practices.

2. Multi-threading
Leveraging on the experience gained in the first developments aiming at a multi-threaded version of Gaudi\cite{2, 3, 4, 5}, we will consolidate the task-based approach to multi-threading in order to provide scalable inter- and intra-event parallelism. The newly developed task scheduler will be improved and extended where needed, to support the use cases that surfaced during the latest Gaudi Workshop\cite{6}.

3. Reentrant Algorithms
The original implementation of Gaudi assumed that algorithms were to operate one by one on one event at a time. This lead to code that relies on event specific states hosted in the algorithms classes, making it impossible to execute the same instance of an algorithm on two events at the same time.

Although it has been shown that one can work around the limitation\cite{3}, to get the best performance from a multi-threaded application (in terms of throughput and memory use) we need to change the way algorithms interact with event data, changing the base classes to make them reentrant.

This change is intrinsically backward incompatible, because it requires a change in the interfaces and the removal of internal states, but we can allow for a smooth transition using backward compatible base classes implemented as special cases of the new reentrant algorithms.

4. Gaudi::Functional Framework
An algorithm is essentially a piece of code that extracts some data from the transient event store, transforms it, and store the results of the transformation back in the transient event store, to be later consumed and transformed by other algorithms. In this respect, algorithms can be seen as functions.

Taking advantage of modern C++ features like variadic templates and template meta-programming techniques, it is possible to hide in a base class most of the event data interaction boilerplate code common to many algorithms. In this way, developers of algorithms only have to implement a method that takes as arguments the data they need from the transient event store, and that returns the data it needs to push back, as if they were writing a simple function. The Gaudi::Functional framework provides such base classes.

User algorithms implemented using the Gaudi::Functional framework have some limitations with respect to normal reentrant algorithms, but these limitations, like the elimination of internal states, are meant to force developers to write code that is more likely to be thread safe.

5. Modernization
Native support for multi-threading and reentrant, thread-safe building blocks are fundamental to efficiently use modern computing resources, but they are not enough to help developers in their day to day work, so we need to reshape Gaudi code to make it easier to use.

Gaudi has plenty of legacy code kept for backward compatibility or needed to support use cases that have become obsolete. We will review the use cases on the basis of the experience of the experiments using Gaudi as their software framework, to finally drop obsolete code. This will not only reduce the amount of code to be maintained, but will help new adopters to focus on best practices.

Thanks to the code generation possibilities given by modern C++, we plan to reduce as much as possible hand written boilerplate code, with the advantage of reducing the possibility of mistakes introduced by careless copying of such code from existing examples.
To improve robustness of the Gaudi code base we will replace old C++98 constructs with their modern counterparts. In particular we plan to adhere as much as possible to the C++ Core Guidelines[7].

6. Conclusions
With the Gaudi Workshop help in September 2016, we made the first steps to prepare Gaudi for future challenges. The work is far from being completed, but we have a clear the vision of what we want to obtain.

To help us keeping up with the task, we meet biweekly at the Gaudi Developers’ Meetings (https://indico.cern.ch/category/1790/).

References