AIDA-2020-CONF-2020-019

AIDA-2020

Advanced European Infrastructures for Detectors at Accelerators

Conference/Workshop Paper

MarlinMT - parallelising the Marlin framework

Ete, R. (DESY) *et al*

26 May 2020

The AIDA-2020 Advanced European Infrastructures for Detectors at Accelerators project has received funding from the European Union's Horizon 2020 Research and Innovation programme under Grant Agreement no. 654168.

This work is part of AIDA-2020 Work Package **3: Advanced software**.

The electronic version of this AIDA-2020 Publication is available via the AIDA-2020 web site <http://aida2020.web.cern.ch> or on the CERN Document Server at the following URL: <http://cds.cern.ch/search?p=AIDA-2020-CONF-2020-019>

Copyright © CERN for the benefit of the AIDA-2020 Consortium

MarlinMT - parallelising the Marlin framework

Remi Ete[1,*], *Frank* Gaede[1], *Julian* Benda[1], and *Hadrian* Grasland[2]

[1]DESY, Notkestraße 85, 22607 Hamburg, Germany
[2]Université Paris-Saclay, CNRS/IN2P3, IJCLab, 91405 Orsay, France

Abstract. Marlin is the event processing framework of the iLCSoft [1] ecosystem. Originally developed for the ILC more than 15 years ago, it is now widely used also by other communities, such as CLICdp, CEPC and many test beam projects such as CALICE, LCTPC and EU-Telescope. While Marlin is lightweight and flexible it was originally designed for sequential processing only. With MarlinMT we now evolved Marlin for parallel processing of events on multi-core architectures based on multi-threading. We report on the necessary developments and issues encountered, within Marlin as well as with the underlying LCIO [4] event data model (EDM). A focus will be put on the new parallel event processing (PEP) scheduler. We conclude with first performance estimates, like the application speedup and a discussion on histogram handling in parallel applications.

1 Introduction

Event processing frameworks in High Energy Physics (HEP) have been used for several decades now, typically running a single task pipeline within a single thread. However, the evolution of hardware shows that we have a reached a limit in terms of single core frequency and that the number of cores in increasing with a power law [2]. This suggests that these frameworks should provide support for multi-threading to keep in sync with the hardware evolution and still provide the necessary scaling capabilities. Marlin [3] is the event processing framework of the linear collider community and was originally not designed for multi-threaded processing. The same is true for the LCIO [4] EDM solution that is used in Marlin. In this paper, we present the work to enable multi-threading in the Marlin framework, carried out in the MarlinMT project. Section 2 describes how the LCIO package has been refactored and improved to cope with the new parallelization scheme in MarlinMT. Section 3 presents the parallel event processing scheduler and the scaling performance obtained using a standard *CPU crunching*- benchmark. Finally, in section 4 we provide our perspective on histogram management in parallel event processing frameworks.

2 The multi-threaded implementation of LCIO

LCIO (*Linear Collider Input-Output*) defines the event data model for linear collider studies and provides the persistency implementation via the SIO (*Simple Input Output*) sub-package. The Marlin framework uses LCIO, both as input and output format as well as its transient

*e-mail: remi.ete@desy.de

event data model. As most of the software written by HEP scientist in the early 2000s, it has not been designed with multi-threading in mind. The first task to make MarlinMT work with LCIO was to refactor parts of the code in order to make multi-threading possible. Apart from removing some global states, it was also necessary to re-implement the SIO persistency layer of LCIO from scratch. The original user API was restricted to simple read and write functions that always handled one full event in memory in one pass. The re-write of the SIO layer now enables the following operations to be entirely decoupled and fully controlled by the user:

- pure I/O *read* or *write* operations from/to disk,
- buffer *uncompression* or *compression* using zlib,
- data *decoding* or *encoding*.

An LCIO event is written as a pair of two records. The first one provides a short summary of the event, including collection names and sizes, the second record then contains the full event data. There are three individual steps necessary to read one record from disk and therefore six steps in total to read one full event, as shown in Figure 1 . The first four of these operations are fast with the event summary record buffer being rather small compared to the full event data record size. The CPU intensive work happens in the last two steps: uncompressing of the record buffer and decoding of the full event data. The motivation for re-writing the

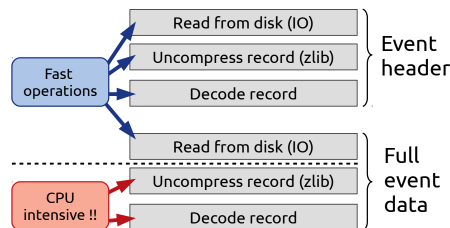

Figure 1. The six steps needed for reading an LCIO event: I/O read, uncompression and buffer decoding, executed for both, the event header and the full event record.

SIO layer was to be able to postpone the two CPU intensive reading steps and forward their execution to separate threads, run in parallel. This feature has been implemented in the LCIO file reader in order to perform a *lazy unpacking* of the event. After performing the four first steps in the main thread, the extracted buffer is moved into the user event structure. The uncompression and full event decoding happens only when the user tries to access event data for the first time. As will be discussed in section 3, this lazy triggering of the event unpacking dramatically improves the scaling performance of MarlinMT. The new implementation of SIO was also used as an alternative I/O for the EDM toolkit PODIO [6] and shows better reading performance than the ROOT based default implementation.

3 The MarlinMT framework

The Marlin framework has been used by the ILC community and related detector R&D groups such CALICE, FCAL, LCTPC and EUTEL for two decades. It is a C++ application framework with sequential processing of tasks, originally designed for single core architectures. The MarlinMT[1] framework is a fork of Marlin that allows for parallel event processing

[1] https://github.com/iLCSoft/MarlinMT

on modern multi-core architectures. While some other HEP frameworks have adopted a task-level parallelism model, we focus with the scheduler presented here on the parallelization at the event level only.

3.1 The scheduler implementation

The scheduler is implemented as a thread-pool that starts N workers at program start-up, as shown in figure 2. Each worker is mapping a logical thread to a hardware thread, provided the hardware allows for it. It runs an ordered list of modules (`marlin::Processors`), called a *sequence* (`marlin::Sequence`). The framework allows for either cloning or sharing individual modules between sequences. If the event processing function of the module is entirely *thread-safe*, sharing the module avoids duplication of data members. In case the module is cloned, thread-safety is ensured by construction, assuming that there is no *non-const static* state used. Each worker is pulling out a single event from the concurrent queue of the scheduler that in turn is populated from the LCIO event reader. While this scheme is rather straight forward to implement it also might have potential issues, compared to a task-oriented scheduler:

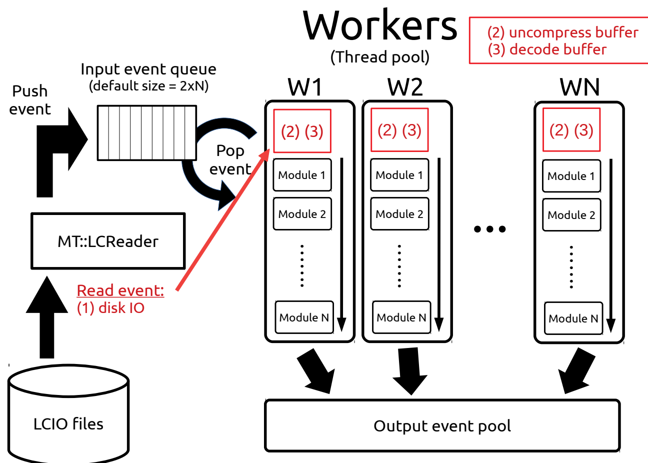

Figure 2. Schematic view of the MarlinMT parallel event processing (PEP) scheduler.

- Single queue access: the concurrent queue is constantly polled in a locking mode by all workers which could impact the scheduling performance if it becomes too frequent. This could happen for a very large number of workers and/or for very short event processing times. The situation could be improved by using a task-stealing implementation to reduce the queue access frequency.
- Memory usage: the number of events loaded in memory is given by the number of workers plus the queue size. Depending on the available memory and average event size there might be an issue. Fortunately, the lazy unpacking of data offered by the new LCIO implementation minimizes this effect as the memory is expanded only on demand during the processing.

On the other hand, there are clear advantages of this scheduler implementation:

- Thread locality: events are, by construction, thread-local (one event per worker) and therefore their processing is thread-safe.
- Minimized scheduling overhead: after popping out an event from the shared queue, no additional scheduling is necessary in contrast to a task-oriented scheduling for which the task granularity will be much higher.
- Cache locality: an event, being thread local by construction, will see its data loaded into a single CPU cache throughout its processing. This will potentially reduce cache misses compared to a task-oriented parallelism for which each task may be run on a different CPU, using a different cache.

3.2 Scaling performances with CPU crunching

A standard way of stressing a scheduler implementation is to run a "*CPU crunching*" module that performs a simple computation (no memory allocation, no data access) for a fixed amount of time. By varying the crunching time and/or the amount of workers, the access to the task queue can be stressed and the resulting scaling performance estimated. Unfortunately, the cache locality cannot be benchmarked with this method as this requires a more realistic implementation of data analysis modules.

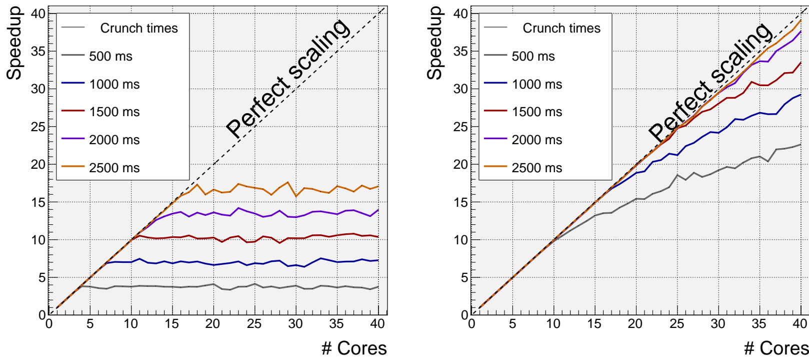

Figure 3. Application speed-up without using the LCIO lazy unpacking (left) and with LCIO lazy unpacking (right).

The scaling of a multi-threaded application is defined as the relative speed-up of the running time as function of the number of cores or threads used. This scaling is obtained by running the application in sequential mode and then in multi-threading mode by varying the number of workers and computing the ratio of the processing times. Figure 3 shows the application speed-up measured without (left) and with (right) the lazy unpacking of the LCIO event for different crunch times. This lazy triggering of data unpacking helps a lot in reducing the amount of sequential code, run in the main thread, by moving it to a worker thread. As an example, the average processing time of the full ILD event reconstruction for $t\bar{t}$ events at $\sqrt{s}$ = 500 GeV is about 5 seconds. As a first estimate, the ILD reconstruction would scale up to 40 cores but this doesn't take into account cache misses, necessary data locking and memory allocations. A re-implementation of the complete ILD reconstruction chain in the MarlinMT framework has started, but it will take some time until a full validation of the realistic scaling behavior of the implemented approach can be measured.

4 Histogramming in a multi-threaded environment

One of the core functionality needed in every HEP-framework is the booking and filling of histograms. In the Marlin framework both tasks were implemented independently in every *Processor* individually, using ROOT histograms. This implementation is inherently not thread-safe and therefore a solution for MarlinMT had to be be found. Other HEP-frameworks typically implement one of the following two approaches:

- Split and merge method: create one *thread-local* copy of the histograms per module, fill these copies independently in each thread and eventually merge them into one before writing to disk. While thread-safety is guaranteed by construction, the memory for the histograms is duplicated for every thread. This can turn out be a problem for applications with very large numbers of histograms, such as online data quality monitoring.
- Resource synchronisation: create only a single histogram in memory, use a synchronisation mechanism, such as a mutex or an atomic array for bins, when filling the histogram. This method could have an effect on the application scaling, as every fill operation might suspend others threads in the synchronisation phase. As no histogram is duplicated this works nicely for applications where memory is crucial, such as online monitoring.

For the MarlinMT framework, we decided to allow for both methods to be used transparently in the framework by implementing the dedicated component *BookStore* for this. The *BookStore* holds the actual instantiations of the histograms that are created in the initialization phase, executed at program start in sequential mode. Depending on a runtime parameter for every histogram, either a single shared memory layout with a mutex synchronisation or a multi-copy memory layout with histogram copies but no synchronisation mechanism is chosen. The *Processors* then fill the histograms via dedicated thread-local *Handle* classes that they receive from an *Entry* member object that had been created when booking the histogram. This is shown in Figure 4-left. The intermediate *Entry* object is needed in order to not have

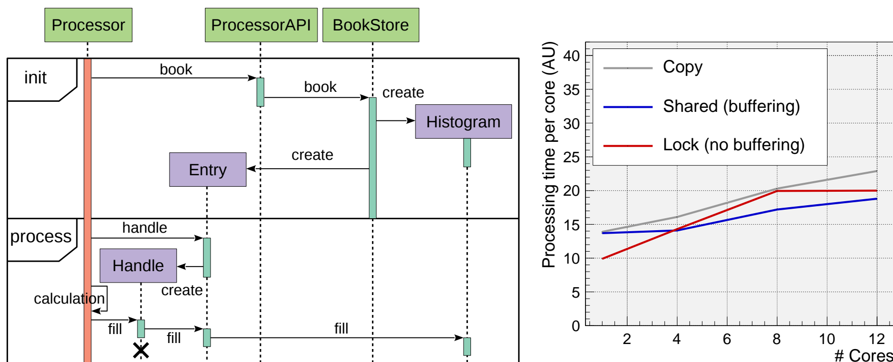

Figure 4. Left: UML sequence diagram of the BookStore component. Right: Performance comparison for the three types of handling the histograms for 1000 histograms with 1000 bins each as a function of the number of cores (threads).

blocking access to the *BookStore* itself. Depending on the mechanism chosen when booking the histogram, the *Entry* object then either fills the local histogram copy or the shared histogram instance via mutex synchronisation. The actual implementation is based on ROOT:

- the histogram implementation is using the new ROOT 7 histogram library (previews of which are bundled in ROOT 6). This new library is better in terms of internal thread-safety and provides built-in tools for histogram filling synchronization and buffering.
- the histograms are converted to ROOT 6 histograms before they are written to disk in order to allow for using ROOT 6 for further processing and visualization of the histograms.
- to reduce the synchronisation phases when using the single shared memory layout, fill calls can be buffered in a thread-local buffer that is flushed into the histogram using the mutex only when it is filled up.

The *BookStore* component itself does not depend on MarlinMT and can be made available for other multi-threaded applications.

In order to measure the performance of the different approaches, we have created a *Processor* that books 1000 histograms with 1000 bins and fills them from an LCIO data file using the copy approach and the shared approach with (shared) and without buffering (lock). The resulting total processing times per core are shown in Figure 4-right as a function of the number of cores/threads used. All three approaches show very similar behavior, where the copy turns out to be the slowest, presumably due to more frequent cache misses. Bulk filling performs better than locked single filling for five cores or more. Other applications might reveal different behavior and users can adjust their choice of the memory layout for the histograms accordingly.

5 Conclusion

MarlintMT is a multi-threading re-implementation of the Marlin framework which is used in the linear collider community and beyond. The scheduler allows for processing events in parallel in different threads on modern multi-core architectures. The performance of MarlinMT has been benchmarked, using a CPU crunching module. It shows good scaling of the total processing time with the number of cores/treads used. The development of the MarlinMT framework also triggered new developments in the LCIO EDM and persistency library. Besides making the underlying I/O implementation SIO thread-safe, a new feature for lazy unpacking was introduced. This allows to postpone the uncompressing and unpacking of the event data to the worker threads after the data has been read from the file. This reduces the amount of sequential code, that is run in MarlinMT and improved the scaling performance considerably. In order to address the problem of histogramming in a multi-threaded environment a new component the *BookStore* has been developed. This component allows to choose at initialization time between three different ways of handling the access to the histograms when filling them in the parallel modules: a local copy per *Processor* or filling a shared histogram instance, either in single fill mode or via buffered bulk-filling. The implementation is based on the, yet experimental, ROOT 7 histogram library with a conversion to the commonly used ROOT 6 histograms at the end of the program. The *BookStore* has been in implemented as an independent, reusable solution which can also work in other applications. The next important step will be to make the large number of Marlin processors that are intesively used in various future collider studies thread-safe and adapt them to the new MarlinMT API.

6 Acknowledgements

This project has received funding from the European Union's Horizon 2020 Research and Innovation programme under Grant Agreement no. 654168.

References

[1] The iLCSoft project: *https://github.com/iLCSoft*

[2] K. Rupp, “42 Years of Microprocessor Trend Data”, https://www.karlrupp.net/2018/02/42-years-of-microprocessor-trend-data

[3] F. Gaede, “Marlin and LCCD: Software tools for the ILC”, NIM **559**, 177-180 (2006)

[4] F. Gaede, T. Behnke, N. Graf and T. Johnson, “LCIO: A Persistency framework for linear collider simulation studies”, eConf C **0303241** (2003) TUKT001

[5] R. Brun and F. Rademakers, “ROOT: An object oriented data analysis framework,” Nucl. Instrum. Meth. A **389** (1997) 81.

[6] F. Gaede, G.A. Stewart, B. Hegner, “PODIO: recent developments in the Plain Old Data EDM toolkit”, These proceedings.