Preparing HEP Software for Concurrency

Marco Clemencic, Benedikt Hegner, Pere Mato, Danilo Piparo
CERN CH-1211, Switzerland

E-mail: `marco.clemencic@cern.ch`, `benedikt.hegner@cern.ch`, `pere.mato@cern.ch`, `danilo.piparo@cern.ch`

Abstract. The necessity for thread-safe experiments' software has recently become very evident, largely driven by the evolution of CPU architectures towards exploiting increasing levels of parallelism. For high-energy physics this represents a real paradigm shift, as concurrent programming was previously only limited to special, well-defined domains like control software or software framework internals. This paradigm shift, however, falls into the middle of the successful LHC programme and many million lines of code have already been written without the need for parallel execution in mind. In this paper we have a closer look at the offline processing applications of the LHC experiments and their readiness for the many-core era. We review how previous design choices impact the move to concurrent programming. We present our findings on transforming parts of the LHC experiments' reconstruction software to thread-safe code, and the main design patterns that have emerged during the process. A plethora of parallel-programming patterns are well known outside the HEP community, but only a few have turned out to be straight forward enough to be suited for non-expert physics programmers. Finally, we propose a potential strategy for the migration of existing HEP experiment software to the many-core era.

The evolution of microprocessor technologies of the past few years stressed a rather clear trend. Newer architectures are not characterised by ever rising clock frequencies but rather by an increasing number of cores per node. The nature of these cores has also evolved: not only general purpose, highly performant multicore processors appeared on the market, but also, driven by the need of power efficient solutions, products featuring a high number of thin and specialised cores, not necessarily x86 compliant, like accelerators or RISC processors. HEP software needs to leverage this new hardware scenarios to cope with the challenges posed by future data processing, especially the one of LHC experiments. Pushed by the ambitious CERN physics' program, the expected performance of the accelerator in terms of centre of mass energy and luminosity in the next decade will make our present software solutions simply not adequate. A primary role in the evolution of particle physics software is played by data processing frameworks, the orchestrators of the algorithms that transform the readout signals of the detectors and Monte Carlo generator programs into discoveries.

1. Concurrent Gaudi

Gaudi [1] is the data processing framework at the core of the software stacks of the Atlas [2], LHCb [3] and other present and past experiments. Its original design was sequential and an effort is ongoing [4] to make it able to fully support parallelism:

- at event level, i.e. being able to process multiple events simultaneously

- at algorithm level[1], i.e. running several algorithms concurrently
- within algorithms.

The attitude adopted to achieve the aforementioned goal is pragmatic: it was decided to start and parallelise a real life use case, i.e. a 20 algorithms slice of the full LHCb reconstruction. The philosophy chosen is highly innovative but evolutionary: a large (millions of lines) existing codebase has to be preserved and only minimal changes to it are affordable. The new components developed to support concurrency did not change the overall architecture of the framework, but can just be considered as a simple extension of the pre-existing infrastructure.

In order not to deal with threads' management explicitly, a task oriented approach was chosen. The unit of work packaged in a task is the closure of an algorithm and its input data. To seamlessly support our design choice, the Intel Threading Building Blocks (TBB) library was utilised [5].

Figure 1 is a sketch of the design of Concurrent Gaudi and stresses the newly added components created to support concurrency.

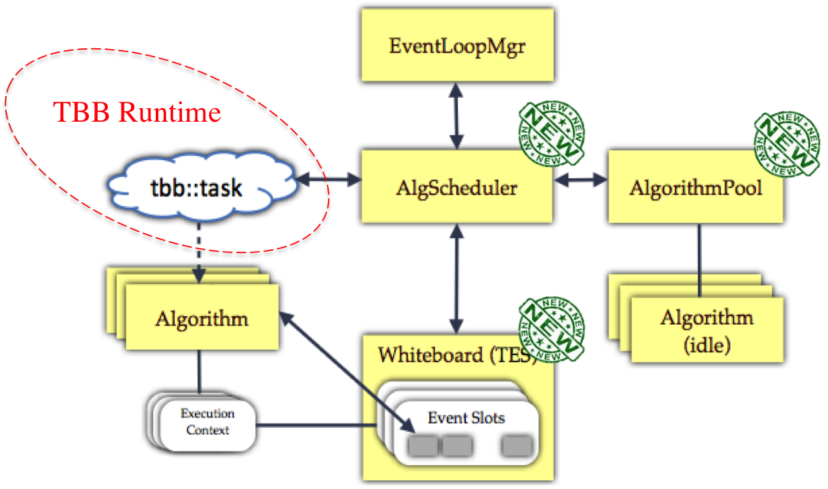

Figure 1. Representation of the design of Concurrent Gaudi. The "new" stamp identifies the components that were added to the original design to support parallelism. The event loop manager loops over the events and hands them over to the algorithm scheduler, which requests algorithms' instances from the AlgorithmPool, packages them into a task and submits them to the TBB runtime. The WhiteBoard is the component responsible to hold the event data stores for all the events being treated simultaneously.

2. Resource protection and correctness

In presence of concurrently executed algorithms, protection of resources, e.g. memory, is an issue. Different traditional techniques can be adopted, such as transactional memory, atomic operations or, as a last resort, locks. A prime example of such issues is the printing on screen of messages coming from algorithms running in parallel.

This problem was elegantly solved adopting a different strategy, i.e. a thread safe multiple producer single consumer queue to serialise the operations to be performed. Basically, a message passing approach is mimicked: the queue contains messages sent by the single algorithms where a

[1] The fundamental data processing unit in Gaudi is called algorithm.

single message is composed by the text to be printed and the printing function. This is achieved with an implementation based on a *tbb::concurrent_bounded_queue* of C++11 lambda functions, called *queue of actions* in the following.

On the other hand, the best way protect resources is a design that excludes or at least minimises contention. In the context of a software evolution which does not foresee intrusive and pervasive changes in the existing code bases, it can happen that multiple algorithms running in parallel need to access a thread-unsafe resource, for example a particular library. The solution identified for this case is the *AlgorithmPool*, the entity containing and coordinating algorithms' instances. It provides instances for their execution and recollects them after their processing finished. In case a thread-unsafe resource is declared to be used by a group of algorithms, the AlgorithmPool delivers only one of such algorithms at the time to avoid a priori the need for resources' protection in a lock-free fashion.

3. The treatment of multiple events simultaneously

The treatment of multiple events simultaneously represents a simple way to increase the problem's size and therefore the opportunity for parallelism, i.e. the probability of being able to schedule an algorithm. The first element necessary to achieve event level parallelism is an entity able to accomodate several *Event Stores*. This is achieved with the *Whiteboard* (see figure 2), which can contain multiple Event Stores, implements the Event Store's interface in a thread-safe manner and provides some additional functionalities, for example the bookkeeping of the newly added data objects (useful for algorithms' scheduling, see section 4).

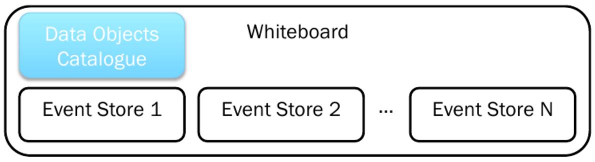

Figure 2. The Whiteboard can be thought as a group of Event Stores. The calls to the Whiteboard are dispatched transparently to the single Event Stores using the information in the Execution Context.

Once multiple events can be treated simultaneously, it is crucial for the entities orchestrated by the framework to know on which event they should act on. This is achieved with the *Execution Context*, which carries the information of the event being processed. The problem of the access to the Event Store can be taken as example: the execution context is necessary to know from (to) which event the data should be read (written). A requirement for the implementation was to minimise the number of changes in the user code and to transparently switch between an ordinary Event Store and a Whiteboard. This goal was achieved through the communication of the execution context between the algorithms and the Whiteboard via the thread local storage. This is the only corner of the design where the pure task oriented model is violated.

3.1. Cloning of components

Even after increasing the size of the problem by processing multiple events simultaneously, it was noted that the runtime performance of the application was still not optimal [6]. This was due to the presence of long running algorithms, which blocked the smooth scaling imposing a coarser granularity to the work carried out by the application.

One possibility to solve this issue is to parallelise such long running algorithms to achieve a finer granularity which allows a more efficient usage of the resources. Unfortunately, this operation is often hard: the parallelisation within algorithms may require major re-engineering efforts.

The solution implemented was the *cloning* of algorithms, i.e. the creation of identical instances of the same algorithm which could run simultaneously on several events. The relevant property of the cloning procedure is that it is not only limited to algorithms, but can be easily extended to all entities participating to the data processing.

Cloning would not be needed if all algorithms were re-entrant and side-effect free. Unfortunately, almost all experiments' software was designed and implemented in the past years without any notion of re-entrancy in mind. For this reason, cloning can be seen as a trade-off between an additional, usually small, memory footprint and a pervasive modification of the existing codebases. It must be noted that some changes in the code may be eventually needed to make an entity cloneable, for example in presence of a class whose data members are pointers that could point to the same region of memory.

4. Scheduling strategies

The scheduler is at the heart of a parallel framework. Different strategies for algorithms' scheduling are possible and some of them were studied, namely the *Forward scheduler*, the *Parallel Sequential* scheduler and the *Round Robin* scheduler. All of these schedulers share the same interface, called *IScheduler*. Gaudi offers total flexibility in selecting with configuration scripts, called *option files*, the implementation to be selected at runtime, allowing a transparent change of scheduling strategy. Concurrent Gaudi is therefore invariant with respect to the scheduling strategy chosen.

The ultimate goal is to compare these scheduling strategies among themselves and with the multi-job and multi-process [7] approaches.

4.1. Forward Scheduler

The idea behind this procedure is to schedule an algorithm as soon as its data dependencies are available. The requirements for a performant forward scheduling are the explicit input data dependencies for all the algorithms, the treatment of multiple events simultaneously and algorithms' cloning.

Forward scheduling is intrinsically immune to deadlocks and fits well a many-core architecture. It is also suited for heterogeneous systems. Indeed, the main problem of offload of calculations on external accelerators such as GPGPUs is that the transfer time must be negligible with respect to the time needed for the calculation. This state can hardly be reached with input collected only within a single event for the vast majority of algorithms under almost all data taking conditions of present detectors. Forward scheduling, in presence of an *accumulator algorithm*, allows to lump together input from different events. In addition, with enough potential for parallelism, work could continue on the CPUs at the same time of the offloaded computations, allowing for a complete occupation of the resources.

Furthermore, forward scheduling can offer the possibility to better use the instruction and data caches of the CPU. Indeed, when the execution of an algorithm finishes, the same thread can be reused to execute the same algorithm for a different event, benefiting from "hot" processor caches.

It is important to note that the Queue of Actions pattern described in section 2 is re-used for the implementation of the forward scheduler, see figure 3.

4.2. Parallel sequential scheduler

The parallel sequential scheduler allows to run the full sequence of algorithms in parallel for several events simultaneously, see figure 4. The requirements for it are the ability to treat

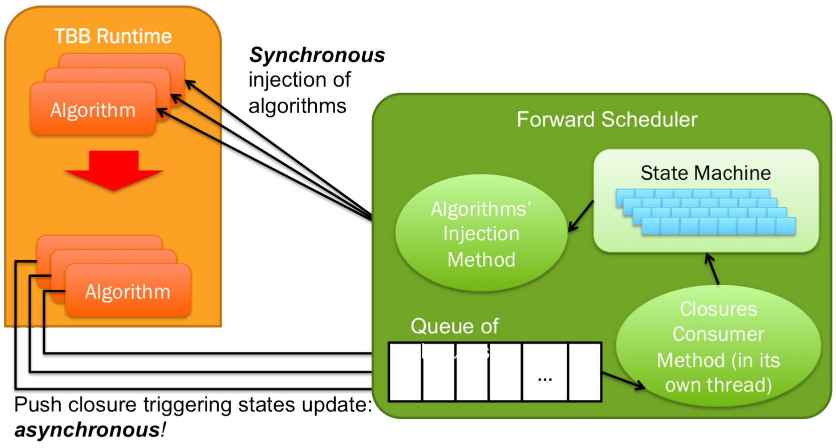
TBB Runtime
Algorithm
Algorithm
Synchronous injection of algorithms
Forward Scheduler
State Machine
Algorithms' Injection Method
Closures Consumer Method (in its own thread)
Queue of
...
Push closure triggering states update: asynchronous!

Figure 3. The scheduler submits tasks for execution according to the information available in a state machine. The *queue of actions*' pattern is used to manage the asynchronous termination of algorithms. When an algorithm terminates, a lambda function that updates the state machine and can trigger the checking for new data objects in the whiteboard is injected in the queue.

multiple events simultaneously and algorithms cloning, without which it would be equivalent to a pipeline.

As the order of algorithms' execution is preserved, explicit modelling of data dependencies is not required. In addition, when compared to the multiprocess approach, it has two advantages: the memory savings are much more aggressive and the mechanisms to dispatch inputs and merge outputs to/from the workers can be avoided with the usage of the Whiteboard. On the other hand, the flexibility offered is much less than the one of the forward scheduler, for example in the field of calculations' offload.

4.3. Round Robin scheduler

The Round Robin scheduler is designed to exploit one core only, processing the events in a round robin fashion and needs only the capability of the framework to treat multiple events in flight and that the sequence of algorithms is identical for every event, see figure 4.

Provided that the algorithms are short enough in terms of number of machine instructions, this strategy allows an optimised usage of the L1 instruction cache and of the prefetching strategies, since a more predictable memory access pattern is used. Despite the fact that the transfer of data to an external accelerator cannot be masked by other work ongoing on the CPUs, inputs can be lumped together across events. In addition, an application steered by this mechanism can be transparently plugged in the present workload management systems, since it is single-threaded.

Round Robin scheduling could represent a good checkpoint between the fully parallel and old sequential running modes.

5. Caches and their classification

The existing code in the implementation of algorithms and, in general, of all framework components contains several types of non constant caches. Some of such entities can lead to invalid behaviours in the case of multiple events treated simultaneously or algorithms running concurrently. A possible way to classify caches is based on their validity span: an event, a run or the duration of the whole application.

Another possible assortment of value caches is to classify them as local, for example within

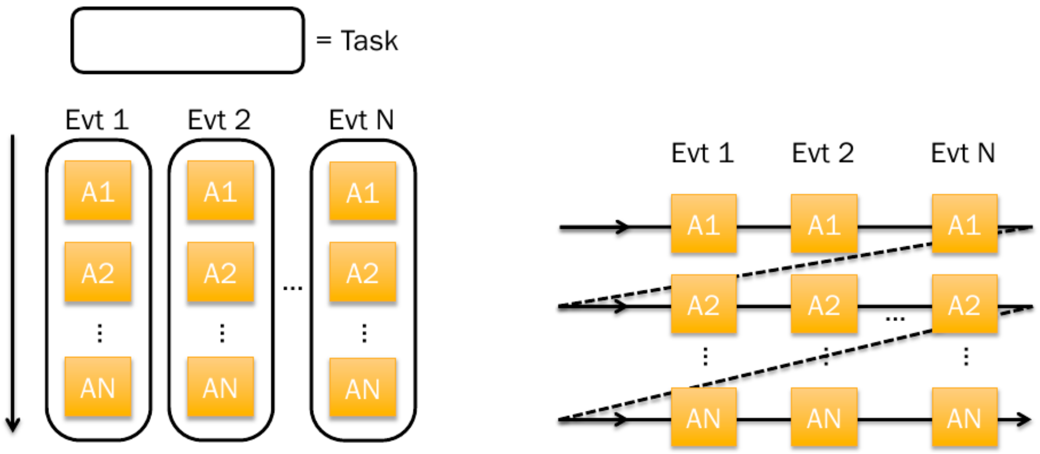

Figure 4. Left: schematic representation of the parallel sequential scheduler. The whole sequence of algorithms is executed in a predefined order in a single task. The similarity with the multiprocess approach is evident. Right: schematic representation of the round robin scheduler. The algorithms are ran on all the events which are being treated in a round robin fashion on a single core.

an algorithm, or globally shared. The caching entities of the former category are not invalid a priori: an example of perfectly valid local caches are the event counters (in absence of cloning, or with cloning but with a final reduction of the accumulated information). On the other hand, globally shared non-constant caches are in general invalid. An example which is particularly dangerous in the context of parallel execution is the one of communication channels among algorithms and other framework entities established to bypass the Event Store.

There are several possible solutions to the problem of caches. First of all, a review of existing caches is to be performed to understand if they are actually needed, e.g. they are responsible for a concrete performance benefit. For the non constant caches initialised once per event and never modified, the best solution is to transfer them into the whiteboard, as a concrete data object. For caches not addressed by the previous strategies, a possible methodology could consist in providing a separate cache per event processed simultaneously.

6. Conclusions

To meet the challenge of the future HEP data processing, software frameworks must support parallelism at the level of events and algorithms, and within algorithms themselves

Starting from a very concrete usecase, the LHCb reconstruction and Gaudi, knowledge was distilled in terms of general strategies and patterns to express parallelism in HEP frameworks, with minimal changes of the existing code bases in mind. The pattern of the AlgorithmPool for a lock-free resources' protection was discussed. The way in which the treatment of multiple events could be achieved with the Whiteboard pattern and the Execution Context was characterised. The importance of the cloning of entities acting within a parallel framework was clarified, together with recommendations for its proper handling. The properties of different scheduling techniques in relation to many cores and heterogeneous platforms were treated, stressing possible migration strategies from the purely sequential execution towards the fully parallel running mode. The implications in terms of changes to be applied to the existing codebases were characterised. Finally, a useful classification of value caches was given, together with some of the possible solutions for their usage in a parallel environment.

References

[1] G. Barrand, I. Belyaev, P. Binko, M. Cattaneo, R. Chytracek, et al. GAUDI - A software architecture and framework for building HEP data processing applications. *Comput.Phys.Commun.*, 140:45–55, 2001.

[2] R W L Jones and D Barberis. The evolution of the atlas computing model. *Journal of Physics: Conference Series*, 219(7):072037, 2010.

[3] N Brook. Lhcb computing model. Technical Report LHCb-2004-119. CERN-LHCb-2004-119, CERN, Geneva, Dec 2004.

[4] Forum on Concurrent Programming Models and Frameworks. https://concurrency.web.cern.ch/GaudiHive.

[5] J. Reinders. *Intel Threading Building Blocks*. O'Reilly Media, 2007.

[6] B. Hegner, P. Mato, and D. Piparo. Evolving LHC data processing frameworks for efficient exploitation of new CPU architectures. In *Nuclear Science Symposium and Medical Imaging Conference (NSS/MIC), 2012 IEEE*, pages 2003–2007, 2012.

[7] D Crooks, P Calafiura, R Harrington, M Jha, T Maeno, S Purdie, H Severini, S Skipsey, V Tsulaia, R Walker, and A Washbrook. Multi-core job submission and grid resource scheduling for atlas athenamp. *Journal of Physics: Conference Series*, 396(3):032115, 2012.