Evolving LHC Data Processing Frameworks for Efficient Exploitation of New CPU Architectures

B. Hegner, P. Mato, D. Piparo

Abstract – **Software engineering is undergoing a paradigm shift in order to accommodate new CPU architectures with many cores, in which concurrency will play a more fundamental role in programming languages and libraries. Development of new models and specialized software frameworks is needed to assist LHC scientists in developing their software algorithms and applications that allow for maximally parallel execution. In this paper we present our current ideas for evolving the frameworks in use by the LHC experiments to support the decomposition of the data processing of each event into smaller tasks that can be executed simultaneously on different CPUs, together with the ability to process several events at the same time. Results from the prototype used to exercise the key aspects of the new frameworks are described.**

I. INTRODUCTION

A major effort to reengineer existing HEP software is needed for the future efficient exploitation of resources being invested in computer centers used by HEP experiments. In fact, many campaigns have already been conducted to measure and improve the performance of existing HEP codes. At the LHC this continues to be a major point of concern, as increasing trigger rates and more event complexity result in ever increasing demands on both CPU and memory. However this situation is taking on a new dimension due to the fact that the latest generations of computers have started to introduce higher levels of parallelism, i.e. new CPU microarchitectures and computing systems with multiple CPUs. Significant agility will be needed to adapt, and even redesign, the algorithms and data structures of existing HEP code to fully utilize the available processing power. Evidently much work needs to be done to evaluate and select the best emerging software technologies, and to adapt our codes to new programming models that can exploit the potential offered by the new hardware.

Today, processors employing 2,4,6 or more cores (multi-core) are in widespread use; moreover this trend is expected to continue in the future towards chips having hundreds of cores. Parallelism is also present within the single cores in the form of vector instructions, instruction pipelining, multiple instructions per clock cycle and hardware threading. As a consequence, the improvement in software performance that can be achieved depends very much on the algorithms used and their implementation. In particular, possible gains are limited by the fraction of the software that can be parallelized to run on multiple cores simultaneously, whilst exploiting at the same time the opportunities offered by parallelism within the single CPU.

Traditionally HEP experiments exploit multiple cores by having simultaneously each core process different HEP *events*; this is an example of a so-called embarrassingly parallel problem that results in speedup factors that scale with the number of cores. However, as already mentioned, a trend towards many (100's) cores on a single socket is expected in the near future, whilst technical limitations on connecting them to shared memory could reduce the amount of memory that can be accessed efficiently by a single core.

There is general recognition of the need for a collective response by the whole community to these trends[1]. As a first step an initiative has already been taken to establish a new forum[2], open to the whole HEP community, making activities in this area visible. The goals are to establish a consensus on technology choices that need to be made, such as the best concurrent programming models and software libraries that support multi-threading, and a common view on new software components that should be developed for use in the data processing frameworks of the HEP experiments.

The investment made by each LHC experiment in the data processing software is huge. It amounts to several million lines of working C++ code and is actually producing good results. Ideally the evolution of these data processing applications should be made in a way that preserves this investment.

II. DATA PROCESSING FRAMEWORKS

LHC experiments have been developing object-oriented software frameworks on which they base all their data processing applications such as trigger, reconstruction, simulation and analysis. An example of such frameworks is Gaudi [1], which is used by LHCb and ATLAS experiments at the LHC. The central component in this framework is the *Algorithm*[3], which transforms raw *event* data into processed data e.g. from detector digitizations to hits, from hits to

Manuscript received November 19, 2012
B. Hegner, P. Mato and D. Piparo are with Physics Department, CERN, 1211 Geneva 23, Switzerland

[2] http://concurrency.web.cern.ch
[3] Also called *Module* in other frameworks such as CMSSW, Marlin

clusters or tracks, from tracks to jets, etc. The implementation of these *Algorithms* encapsulates the knowledge physicists have of detector and physics performance, and represents the real substance of these data processing applications. The software integrators then combine a fairly large number of these *Algorithms*, together with other components that provide core functionality, in order to assemble and configure a complete application.

The way an *Algorithm* interacts with the framework is kept very simple. It interacts solely with a special piece of code called the *Transient Event Data Store* (also called *Whiteboard* later in this paper) in order to retrieve its input data and eventually also to store the results it produces, called the data products. The execution of each *Algorithm* is completely independent of those other algorithms (the producers) that provide its input data, as well as those algorithms (the consumers) that make use of its results (Fig. 1).

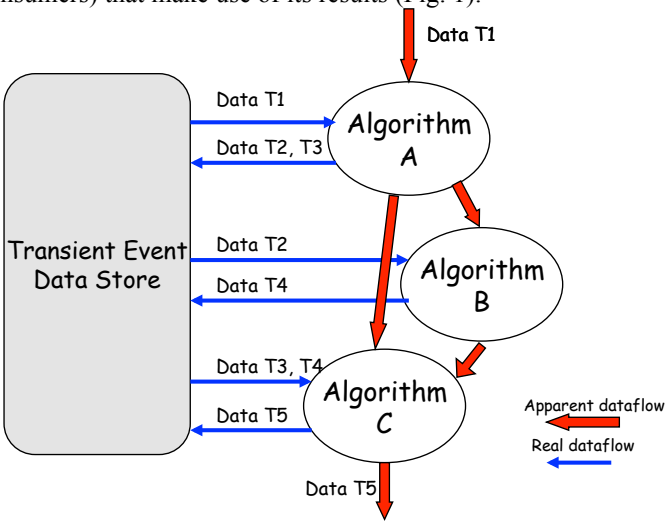

Fig. 1. The dataflow between *Algorithms* is implemented using a central service in the framework, which is the *Transient Event Data Store (Whiteboard)*. The apparent dataflow is actually implemented in terms of "gets" and "puts" to the data store such that the *Algorithm* developer does not need to know how and by what the input data was produced and what will consume the results.

Algorithms are in today's frameworks "initialized" at the beginning of the job and run sequentially in a predefined order for each *event* in the main *event* loop, and "finalized" at the end to output any statistical quantities. The sequencing of algorithms such that some can run in parallel is central to enabling the concurrent execution of code when processing a single event. The one-job-per-core approach, which is currently used for LHC data processing, will soon fail due to memory limitations and the fact that the merging of output files produced by each job needs to be done sequentially, thus reducing the overall speedup (Amdahl's law [3]).

III. Levels of Concurrency

In the execution structure of current frameworks three levels of parallelism may be identified: parallelism among events, among algorithms and within single algorithms. All three categories need to be considered when designing a concurrent HEP framework.

The first step is to determine the data dependencies between the full set of *Algorithms*; those that are data-independent can be run in parallel. A careful study of several large data processing applications has revealed that most time is spent in the execution of a relatively small number of complex algorithms such that, on average, only a few can be executed simultaneously [4].

This limitation can be overcome by processing multiple events concurrently. It increases the probability to be in the right condition to schedule a certain *Algorithm* at a given time, therefore incrementing the maximum achievable level of concurrency. The two aforementioned means of expressing parallelism have the advantage to be completely encapsulated in the framework therewith shielding the *Algorithms* developers from the difficulties linked to parallel programming and, above all, to re-use, only with a limited set of changes, existing code.

Finally, parallelism within algorithms offers a means to further exploit the available resources. In addition, it naturally allows shrinking its runtime and therefore to reduce the congestion generated in the scheduling of long-running entities.

An implementation should aim to dovetail these domain specific categories of parallelism with the features offered by modern hardware. We will restrict ourselves to CPUs in this document. In this context, three main types of hardware support for parallelism can be itemized: instruction level parallelism (ILP), vector units and multiprocessor systems (including multicore processors).

ILP is the potential overlap among the execution of instructions by the CPU, and is realized by using components such as the instruction pipeline or superscalar architectures. Despite its importance, a comprehensive characterization of ILP is beyond the scope of this paper due to its complexity (see for example [5] for details).

Vector units are processor registers, which can be used to execute single instructions on multiple data (SIMD). They therefore offer the most low-level implementation of data parallelism. Hardware vendors have been supporting different flavors of SIMD instructions for more than ten years [6]. Notable examples of SIMD instruction sets are the SSE, which allows four single precision floating point or two double precision numbers to be treated, and the more recent AVX that allows for eight single precision floating point or four double precision numbers. The extrapolation of present trends in hardware technologies clearly suggests that vector units will be an important feature of future chip design.

One of the most prominent manifestations of Moore's law [7] in recent years has been the presence of an increasing number of CPU cores within the same die of commodity chips. This new situation paves the way towards great increases of software performance, achievable for example via a task based programming model.

IV. Task-Based Parallelism

On transforming a sequential problem into a parallel one, *chunks* of work must be identified that are independent of each other, such that they can be executed concurrently. At the

level of the operating system there are two ways of mapping the identified *chunks* onto concrete entities i.e. multiple (independent) processes or a process with multiple threads. However, the creation and deletion of both processes and even threads introduces a big overhead, which may cancel out the potential speedup. A common software design pattern, so-called thread pools, can be adopted in order to alleviate this problem. In this approach, a fixed number of threads is re-used for various *tasks* that are placed in a *task queue*.

Conceptually one can distinguish two types of behavior of these tasks and their scheduling, namely preemptive and cooperative. While in the preemptive case tasks may be interrupted by the scheduler to give way for other tasks, a cooperative scheduling requires a given task to free its resources explicitly. The former case is essential for highly interactive use cases; the latter usually yields the better throughput performance, due to a reduced number of context switches, and is best-suited for a non-interactive application. One possible implementation of the cooperative task model is provided by the Intel Threading Building Block (TBB) library [8]; the work presented in this paper is based upon use of TBB.

TBB offers higher a high-level abstraction layer based on the task model. One example is *parallel_for*, which slices a loop into several tasks. In order not to have these probably short-lived ad-hoc tasks compete with the execution of older, more heavy-weight tasks, the TBB task scheduler processes the task queue in last-in-first-out (LIFO) order. This as well dramatically improves the potential for cache re-usage. However, LIFO processing usually comes with a bad latency behavior, since tasks may be sitting in the queue for a rather long time. Section VI explains how this issue is being addressed in the present study.

V. The GaudiHive Prototype

We have started to re-design the Gaudi framework to embrace concurrency. For this we have created a prototype (GaudiHive), which follows the concept of task parallelism. Here a task executes a given *Algorithm* on a given *event*. The dependencies of these tasks on each other can be expressed in terms of a directed acyclic graph (DAG) formed from the input-output relation of the *Algorithms* (Fig. 1). This graph can be exploited for parallel scheduling in three ways:

1. Scheduling following a fully static approach.
2. Launching algorithms whenever their output is being requested, i.e. data-on-demand. This implies traversing the dependency graph backwards.
3. Launching algorithms whenever the required input is available, which implies traversing the dependency graph forwards.

Ideally the first option seems to be the best approach. However, the lack of dynamic components imposes a major hurdle when considering a use-case such as triggering in which *Algorithm* execution may become optional. In this case the second approach seems to be the most efficient as there are never algorithms executed whose output is not needed. However, data-on-demand interrupts an *Algorithm* in the middle of execution and can only request one input at a time, thus placing a limit on the level of concurrency that can be achieved.

For these reasons, the GaudiHive prototype is based on driving the execution according to the availability of data i.e. option 3. In practice, this is achieved as follows. The central elements are depicted in Fig. 2 and include a special parallel *Scheduler* and the *Whiteboard* as thread-safe event store. In addition, all *Algorithms* are required to declare their required input data as part of the initialization step. The other parts of Gaudi are left unaltered.

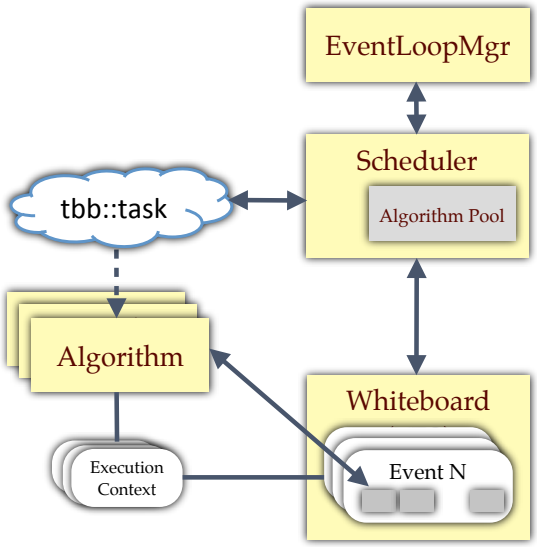

Fig. 2. A diagram illustrating collaboration between framework components developed for the GaudiHive prototype.

As soon as new data become available in the *Whiteboard*, the *Scheduler* checks to see whether there are *Algorithms* whose input data dependencies are fulfilled. Concrete *Algorithm* instances are then requested from an *AlgorithmPool*, and then wrapped as TBB tasks. These are then pushed to the *task queue* and eventually executed. As soon as execution is finished, the instance is released again to the *AlgorithmPool*.

Exploiting this for *intra-event concurrency* the maximal speedup is limited by the real *Algorithm* dependencies. A speedup of 3-5 for a typical LHC experiment reconstruction can be typically achieved. This however does not scale with the increase of CPU core counts that we can anticipate in the near future. Therefore the option of executing multiple events in parallel has to be considered.

Introducing event parallelism has many implications. Firstly, the *Whiteboard* has to store multiple events in a thread-safe manner, as depicted in Fig. 2. In addition, *Algorithms* are usually not thread-safe and so a complex *Algorithm*, such as found in track reconstruction, cannot be applied in two events at the same time and therefore it requires that the CPU has exclusive access to internal states of the track *Algorithm*. In the present case, the management of exclusive *Algorithm* instances is done via a (thread-safe) *AlgorithmPool*. To reduce the blocking due to busy algorithms, the presented prototype allows the cloning of *Algorithms*, so that multiple instances of the same *Algorithm* are available in the *AlgorithmPool*.

Obviously, cloning imposes a problem for internal bookkeeping, including the use of counters or histograms. The copies of these data have to be combined once synchronization points such as 'end-of-run' and 'end-of-job' are reached. However, as will be shown in section VI this cloning is only necessary for a handful of algorithms. The reduction problem will thus be solvable by adjusting a limited number of well-defined parts of the code.

While in a single-event-framework the currently processed event, including event data and corresponding detector conditions data, can be treated as a global state, a multi-event framework cannot make this assumption. GaudiHive uses the concept of an *ExecutionContext*, which gives access to all event specific data relevant for the application of an *Algorithm* in a given event. The most prominent use of this feature is to reference the proper event in the *Whiteboard*.

The prototype consists of an implementation of the components shown in Fig. 2, together with the already existing components such as the thread-safe logging mechanism. This has allowed us to perform detailed studies of runtime behavior and to measure speed-up factors that can be achieved. The results of these measurements are described and discussed in the following sections.

VI. Prototype Results

In order to measure the expected performance of the GaudiHive prototype in a simplified environment, real implementations of the algorithms were replaced by emulations that reproduced the expected runtimes. They corresponded to a real workflow of the LHCb reconstruction application (Brunel), which includes about 214 reconstruction *Algorithms* and their data dependencies. The speedup normalized to the serial version was measured with respect to the size of the thread pool for different numbers of simultaneous events, enabling and disabling cloning of algorithms.

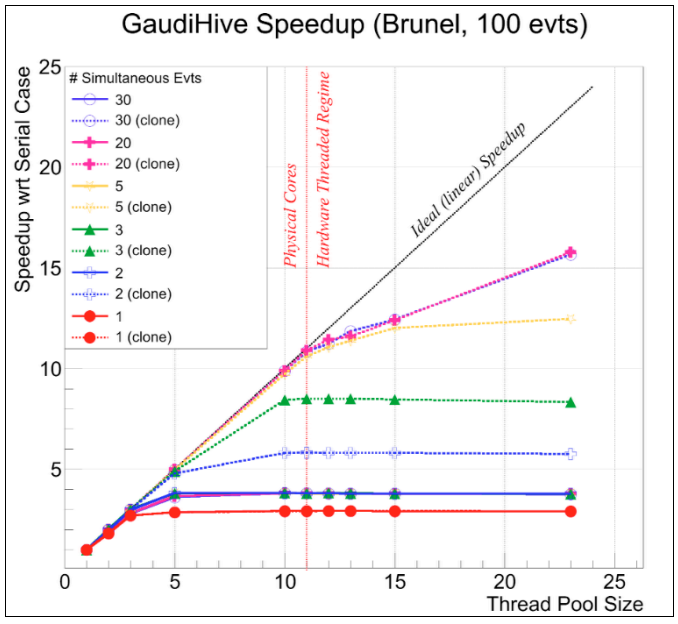

Fig. 3. Speedup normalized to the linear version as a function of the thread pool size on a 12 physical (24 hardware threaded) core machine.

Fig. 3 shows the speedup that can be achieved, normalized to the linear version, as a function of the thread pool size on a 12 physical (24 hardware threaded) core machine. A saturation speedup factor of about 4 is reached without cloning algorithms (solid lines). Once cloning is enabled, perfect scaling is present up to 11 cores (the main thread was not used to schedule algorithms), the degraded performance of hardware threads is then evident. It is important to note how the increase of the number of events simultaneously processed improves parallelism and how saturation is reached at about the value of 20 for the number of available physical threads on this particular machine.

Another benefit linked to the usage of cloning is the reduction of the event backlog, i.e. the difference, at a given time, between the largest and smallest event number among the ones of the events being processed and this is shown in Fig. 4. Therefore we can guarantee an upper limit in the event latency.

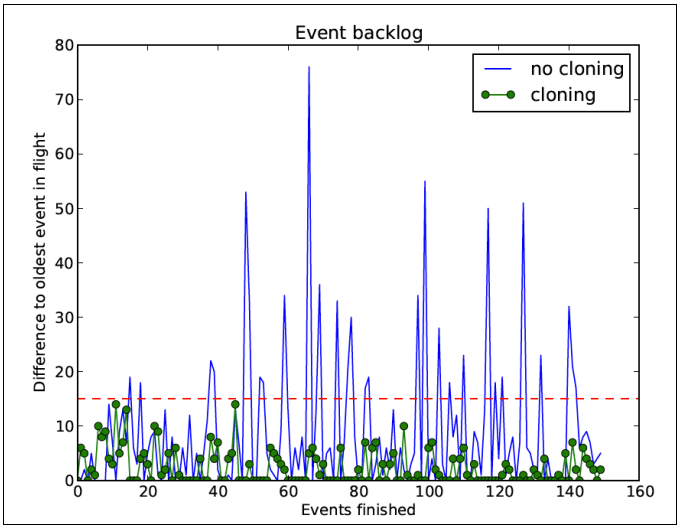

Fig. 4. Event backlog for the processing of 150 events, with 10 threads and 15 simultaneous events, in the presence (absence) of algorithm cloning.

Algorithm cloning requires additional memory resources, but in order to achieve a good scaling, cloning of the algorithms with the longest runtime may only be necessary. Fig. 5 shows the final number of *Algorithm* instances as a function of their runtime that result from the automatic cloning strategy currently implemented in the prototype. An *Algorithm* is cloned if it can be scheduled (i.e. all its required data items are available) and all its instances are busy on other events. Obviously, the longer the runtime of an *Algorithm*, the higher is the probability of needing to clone it. It can be seen in Fig. 5 that the vast majority of *Algorithms* may not require to be cloned. In addition, the ones that ended as two copies could also be avoided without lost of performance.

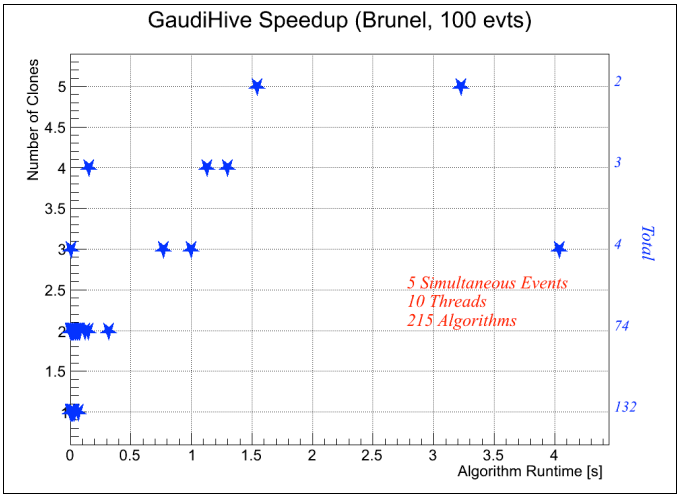

Fig. 5. The final number of *Algorithm* instances as a function of their runtime that result from the automatic cloning strategy currently implemented.

VII. Plans

The results already obtained in scheduling *Algorithms* are very encouraging but we are still far from running an application with realistic *Algorithms* processing real physics data and producing results that can be compared with the sequential version of the application. We need to continue the investigation on how to make all the elements of the Gaudi framework thread-safe in an optimal manner. These elements are *Services,* such as the histogram service or the random number service, as well as the *Tools* used by the *Algorithms,* and the asynchronous messages exchanged between components that are called *Incidents*. Ideally we would like to find re-useable patterns for thread-safe access to these shared services and resources.

The strategy we are following is to start the adaptation of a reduced workflow of the LHCb reconstruction program that consists of about 30 *Algorithms* producing real results that can be compared. This will give us enough variety of multi-threading problems to solve without being overwhelmed by the task. This mini-Brunel will also provide us a solid benchmark to validate the implemented solutions. Later we plan to extend it to the full Brunel workflow once it is working satisfactory.

VIII. Conclusions

Applications will need to exploit increasing levels of parallelism if we want to fully exploit the continuing exponential CPU throughput gains. We are convinced that introducing parallelism at the level of the framework has the potential of scaling to large number of threads, or cores, and at the same time spares the developers of *Algorithms*, i.e. physicists, from having to develop new and complex parallel code. This will allow us to preserve the huge investment made in the existing LHC software.

Collaboration and sharing knowledge and findings between HEP experiments and major projects in the early days on this new endeavor is essential. Evolving the current sequential data processing applications to concurrent ones is a major paradigm shift, comparable to the introduction of object-orientation that the HEP community made about 10-15 years ago. The Concurrency Forum is serving the HEP community in this new era. Promising technologies and programming models (such as TBB) have been evaluated and a number of important results have already been achieved.

The GaudiHive prototype of the Gaudi Framework introducing concurrency has been developed. At its current state it is already an ideal test-bench for validating scheduling strategies of typical HEP applications data-flows. We plan to evolve the current prototype to be able to run real physics applications and use this to learn all possible difficulties of migrating originally written sequential code into a multi-threaded environment.

A clear trend is emerging for the future of HEP data processing applications. These new applications will need to introduce parallelism inside CPU demanding Algorithms, be able to run several independent *Algorithms* in parallel and at the time be able to process several events in parallel. Only adding the three levels we will manage to achieve the desired scalability to fully exploit the new CPUs.

IX. Acknowledgment

We thank our colleagues Riccardo Mari Bianchi, Marco Clemencic, Markus Frank and Illya Shapoval for the very fruitful and often-lengthy discussions we have had during the inception of this new framework.

We thank John Harvey for reviewing the manuscript.

X. References

[1] J. Harvey et al, *Addressing the challenges posed to HEP software due to the emergence of new CPU architectures*, paper submitted at Open Symposium on European Strategy for Particle Physics 2012, Kraków, Poland

[2] G. Barrand et al, *GAUDI - A software architecture and framework for building HEP data processing applications*, Comput.Phys.Commun. 140 (2001) 45-55

[3] G. M. Amdahl, Spring Joint Computer Conference, Atlantic City, NJ, USA, 18 - 20 Apr 1967, pp.483-485

[4] C. D. Jones et al., *Multi-core aware applications in CMS*, 2011 J. Phys.: Conf. Ser. 331 042012

[5] B. Ramakrishna Rau, Joseph A. Fisher, *Instruction-level parallel processing: History, overview, and perspective*, The Journal of Supercomputing - TJS , vol. 7, no. 1, pp. 9-50, 1993

[6] V. Innocente, D. Piparo, T. Hauth, *Development and Evaluation of Vectorised and Multi-Core Event Reconstruction Algorithms within the CMS Software Framework*, Proceedings of CHEP2012, JPCS to appear

[7] G. Moore, *Cramming more components onto integrated circuits*, Electronics, pp. 114–117, April 19, 1965.

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