# Evolution of the ATLAS Software Framework towards Concurrency

**RWL Jones**[1]

Department of Physics, Lancaster University, Lancaster, UK

**GA Stewart**

School of Physics and Astronomy, University of Glasgow, Glasgow, UK

**C Leggett**

Physics Division, Lawrence Berkeley National Laboratory and University of California, Berkeley, CA, USA

**BM Wynne**

School of Physics and Astronomy, University of Edinburgh, Edinburgh, UK

E-mail: Roger.Jones@lancaster.ac.uk

**Abstract**. The ATLAS experiment has successfully used its Gaudi/Athena software framework for data taking and analysis during the first LHC run, with billions of events successfully processed. However, the design of Gaudi/Athena dates from early 2000 and the software and the physics code has been written using a single threaded, serial design. This programming model has increasing difficulty in exploiting the potential of current CPUs, which offer their best performance only through taking full advantage of multiple cores and wide vector registers. Future CPU evolution will intensify this trend, with core counts increasing and memory per core falling. Maximising performance per watt will be a key metric, so all of these cores must be used as efficiently as possible. In order to address the deficiencies of the current framework, ATLAS has embarked upon two projects: first, a practical demonstration of the use of multi-threading in our reconstruction software, using the GaudiHive framework; second, an exercise to gather requirements for an updated framework, going back to the first principles of how event processing occurs. In this paper we report on both these aspects of our work. For the hive based demonstrators, we discuss what changes were necessary in order to allow the serially designed ATLAS code to run, both to the framework and to the tools and algorithms used. We report on what general lessons were learned about the code patterns that had been employed in the software and which patterns were identified as particularly problematic for multi-threading. These lessons were fed into our considerations of a new framework and we present preliminary conclusions on this work. In particular we identify areas where the framework can be simplified in order to aid the implementation of a concurrent event

[1] To whom all correspondence should be addressed.

processing scheme. Finally, we discuss the practical difficulties involved in migrating a large established code base to a multi-threaded framework and how this can be achieved for LHC Run 3.

## 1. Introduction

The upcoming upgrades to the Large Hadron Collider both in energy and in luminosity, coupled with upgrades to various components of the detector, are in themselves enough to motivate changes to the ATLAS software framework. The changes bring with them increases in the pile-up of many collisions in one read-out of the detector, increased acceptance rates from the trigger to the offline and increased complexity in the events. These all present a challenging computing requirement. To illustrate this, the pile-up of events at the end of the run 1 data-taking was about 25; this will rise to well over 100 by run 4, presenting a particularly challenging pattern recognition problem for the tracking. This will imply a much larger processing load, which will be matched by increased simulation requirements. This also results in a larger raw event size of about 2MB (c.f. 1MB in run 1), which coupled with a planned factor 5-10 increase in the trigger output rate (to 5-10kHz in run 4) implies a much higher data rate, and a much bigger load on the storage, retrieval and archival systems.
The need for framework improvements is compounded by the changing hardware landscape. While the Moore's Law doubling of the number of transistors every two years still holds, the architectures are changing to sustain this. In particular, our existing code risks leaving transistors waiting for something to do, with the introduction of vector registers, out of order execution, hyperthreading etc. While such developments increase the theoretical power of the processor, achieving this with HEP applications is a challenge. The dominance of Xeon x86_64 as a platform for HEP is unlikely to last, with more inhomogeneous architectures becoming common: low power, low memory processors such as the ARM, Atom and Avaton; very high core count stronger processors such as the Xeon Phi; and even GPGPU architectures, such as NVidia CUDA. With computing budgets projected to be flat, at best, the experiments must make use of every available resource, often shared with other users; and this calls for performant and flexible code, run under a performant and flexible framework.
The offline processing in Run 1 was trivially coarse-grained parallel, with events handled by independent processes, each demanding about 2GB of physical memory per core. This was efficient in the usage of the cores, but becomes increasingly demanding in costly memory as the core count increases. ATLAS has made an important advance in addressing this for Run 2 by introducing a multi-process version of the Athena framework [1], Athena-MP. This makes use of the Linux kernel's copy-on-write feature. The framework is initialised only once by the initial mother process. Then worker processes are forked and each fork handles a distinct stream of events. All the memory pages corresponding to large static data structures (such as magnetic field and detector geometry) are only read from during event processing, so remain shared between the mother and workers, with only one physical page actually used. Memory pages that are written to by worker processes during event processing are unshared so the overall memory usage per processor is significantly reduced as the core count per processor increases. However, tests indicate that the memory saving of ~0.8GB/worker would be insufficient for post-Xeon architectures.
To go beyond this in terms of memory saving, multi-threading is required. This can provide huge savings, as all heap memory is shared. The cost is amore complicated programming model, as locks are required and data races and deadlocks need to be avoided. This is made doubly difficult, as a large (approximately 7 million line) successful code base exists that was written for serial mode and has been in use for over a decade. The evolution path needs to be considered very carefully, and adiabatic change made when possible.
There are three levels at which parallelism can be applied: the event, the algorithm and in-algorithm. While a less parallel option may be possible, with one event per thread, experience so far indicates that the requirement of multi-threading is sufficiently strong that the additional levels of parallelism are easily gained, and add to the flexibility of the code-base.

## 2. Prototypes and testbeds

### 2.1. CPU Crunchers
ATLAS has undertaken a series of experiments to determine the way forward. The first step, reported previously, was to use simulated algorithms (known as 'CPU-crunchers') to establish that multi-threading provides the gains expected. To do this, the GaudiHive [2] prototype was chosen (see other contributions to this meeting). This is a natural choice given ATLAS already shares the Gaudi framework, of which Athena is a derivative. The conclusion of the CPU-cruncher tests is that multi-threading events clearly works, with good use of the machine resources. In this regard, Athena-MP is already doing well. However, the Hive solution demonstrates that as well as scaling with multiple-events in flight, as with Athena-MP, significant gains can be made by having multiple copies of cpu-intensive algorithms (algorithm cloning).

### 2.2. Real reconstruction
The next step was to apply these ideas to real reconstruction. The current full reconstruction runs several hundred algorithms that produce hundreds of objects in the event store. A data flow and dependency analysis is sobering: there are tangled relations, and the simple analysis turns out to understate the issue. There are also hidden dependencies through the use of public tools, which would make scheduling for parallel running problematic. Rather than tackle the full problem at once, the approach was to take a more manageable part of the reconstruction, the calorimeter reconstruction, involving only seven algorithms and 16 data objects, and to implement it in GaudiHive. This had the additional benefit of being of interest to the software trigger activity.
Even so, many code modifications were required. The event selector was redesigned to work with Hive. There was also work on the event IO in the form of the convertors into the event storage, StoreGate. These were made thread safe by simple locking, with no attempt at parallelisation. Multiple events were bottlenecked by a single convertor algorithm. (There is clear scope for improvement here, as the IO step is about 12% of the processing time.) There was work on the AthAlgorithm and AthAlgTool base classes to enhance the thread and event slot information. A significant piece of the work was on the bypassing many uses of incidents, where execution jumps to plugins defined by the user. (These work well in serial processing, but in the parallel case even incidents such as BeginEvent and EndEvent become ambiguous with many events in flight). There was also work required in the user code to avoid the use of incidents to trigger actions, which would render the workflow unscheduleable. The user code was also changed such that data is only added to the event store when complete for use by other algorithms, not changed in-situ; again, without this scheduling becomes nearly impossible.
The code was benchmarked against regular Athena, serial Hive, and Hive with 1 event in flight and 30 threads. The number of events in flight with 5 algorithms and 30 threads was then compared, but without algorithmic cloning. The throughput improves using up to 3 events in flight, with a modest increase in memory usage and a factor 2x speed up. Beyond 3 events in parallel throughput is bottlenecked by the most time consuming algorithm and no further gain is achieved.
The exercise was then repeated with cloning of the most expensive algorithms and their tools, where this was possible. This is an intermediate solution on the way to full thread-safe algorithms, and helps alleviate the choke point of the most expensive algorithms. Even with this limited cloning, the bottlenecks were removed and the improvements continue to many events in flight. The best results, achieved with cloning just 4 algorithms, gave a factor 3.3 increase in speed (with the initialisation removed) for a 28% increase in memory usage.

### 2.3. Conclusions
The tests showed that multi-threading can speed up processing at a lower memory cost than serial or multi-process techniques, although not being able to clone certain algorithms is a handicap. Several approaches can be considered to avoid this issue, from making the tools fully thread safe to splitting-

up very large algorithms into more tractable ones. It was also learned that incidents are a problem as these express workflows that are hidden from the scheduler. It became evident that  incidents have often been used in the ATLAS code when alternatives exist. It is clear that access to the event store will need clearer patterns for concurrent processing, and that data dependencies need to be propagated to the lowest level where they are used; algorithms may know nothing of data they are to use via tools.

## 3. Requirements Capture

These lessons have been gathered and lead the formation of the Future Frameworks Requirement Group (FFREQ), which is charged with planning the evolution of the framework to Run 3. The ambition is to incorporate the requirements of the offline and of the high level trigger, to minimize or eliminate the differences between the two code bases. The initial emphasis has been on requirement capture and use cases. The event processing pattern, which has served well in particle physics, is still used, but now updated to incorporate concurrency. Events pass through a sequence of algorithms that read the event and other data, and generate new object. Much of the work is done through tools, which are now private to the algorithm. Incidents still exist and can be triggered by any event during the workflow, but are now handled by the scheduler in a controlled way.

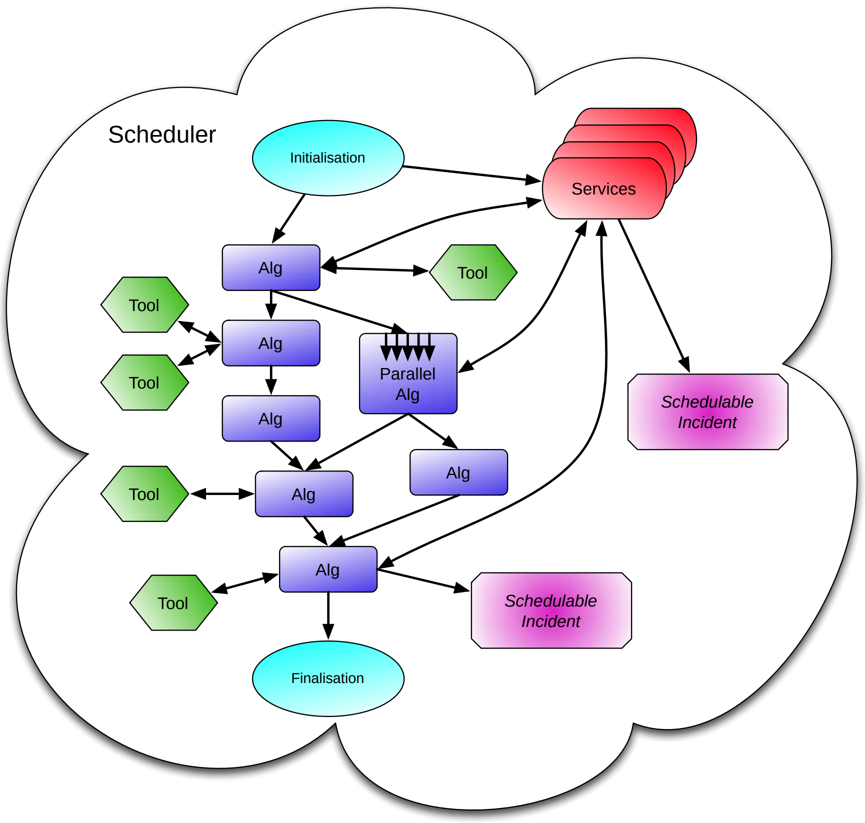

**Figure 1. The proposed new event flow, for a single event.**

Based on the experience from the prototype, public tools are avoided and replaced by thread-safe services. The reliance on incidents is minimised, and generally replaced by data flow and control flow dependencies (in a loose sense, case statements). Interactions in the event store will proceed via data handles, which provide a well-defined interface that declares the intent (read, write or update) of the algorithm with regards to an event store object. It is recognised that the event store data may be mutable – an algorithm can read and write to the same object(s) in the store. (An example might be in calibration or in augmenting an object.) This leads to the problem of multiple algorithms attempting to mutate the same data. The solution is to require that the order in which they mutate the object be declared at configuration time; if this is not done, there is a configuration error. Algorithms that only read the object will, by default, be scheduled after all mutations. The scheduling of the block of mutating algorithms is decided at run time. The block of algorithms should be treated as single schedulable unit, which will reduce the load on the scheduler during the event loop. E.g., it has been shown that such sequences can be implemented as Intel Threaded Building Blocks graphs, which are efficient even with very small computational units.

The High Level Trigger poses additional requirements on the framework. It takes multiple regions of interest in the detector and runs the reconstruction within these, and then runs multiple trigger chains in that region until a chain fails or the event is accepted. Different chains can configure the same algorithm, but for efficiency it should only be run once on the same data. Equally, data requests from the read out system and the time to clear the event build and read out buffers need to be minimised. In addition, the trigger can consider the multiple regions of interest in the same event and run the same algorithms in the trigger chains over this different input data in different detector regions. These considerations require that the event store must support multiple views. Algorithms and RoIs become mutually dependent in some contexts, which complicates the event scheduling. This concept of multiple views could be useful in the offline context, for instance allowing more expensive tracking to be run for candidate tau lepton decays.

## 4. Conclusions

Various important lessons have been learned concerning the required changes to the code base. Algorithms that are not heavy resource consumers that can be run in single instances without special constraints do not pose a problem; this generally means the physics analysis code can remain largely unchanged. One exception is the use of patterns hostile to threading, which seem to have been adopted by many coders, e.g., the use of public tools to cache data passed between algorithms without explicit use of the event store and the use of incidents, such as EndEvent to clear such caches. On the other hand, time consuming code must be made thread safe. This can mean either cloning in existing code or true thread safety in new code. This strategy allows work on thread safety to begin before the new framework is ready, and multithreading within single algorithms can already be implemented. Another key development is to ensure that services that can be called many times in different contexts must be made thread safe.

One inevitable overall conclusion is that good programming models and training must be provided to the developer community. With these lessons in mind, the requirements capture for the framework design in Runs 3 and 4 at very high luminosity is underway. There is reasonable hope that the trigger can be accommodated in this new framework, but the development of the scheduler and event store goes beyond the current Hive prototype. However, the offline may also benefit from these extensions. The plan is to design and implement the new framework during the next run, Run 2 and the code would then be adapted during the long shutdown that follows ready for Run 3.

## References

[1] ATLAS Collabroation, *ATLAS Computing Techincal Design Report*, TDR-017. CERN-LHCC-2005-022

[2] B. Hegner et al, *Evolving LHC Data Processing Frameworks for Efficient Exploitation of New CPU Architectures*, Proceedings IEEE-NSS 2012

[3] D. Crooks et al, *Multi-core job submission and grid resource scheduling for ATLAS AthenaMP,* Computing in High Energy and Nuclear Physics 2012, New York, NY, USA