ATL-SOFT-PROC-2018-038
27 November 2018

The ATLAS multithreaded offline framework

Sami Kama[1], *Charles* Leggett[2], *Scott* Snyder[3,*], and *Vakho* Tsulaia[2], on behalf of the ATLAS collaboration

[1]Southern Methodist University, Dallas, TX, 75205, USA
[2]Lawrence Berkeley National Laboratory, 1 Cyclotron Road, Berkeley, CA 94720, USA
[3]Brookhaven National Laboratory, PO Box 5000, Upton NY, 11973, USA

Abstract. In preparation for Run 3 of the LHC, scheduled to start in 2021, the ATLAS experiment is revising its offline software so as to better take advantage of machines with many cores. A major part of this effort is migrating the software to run as a fully multithreaded application, as this has been shown to significantly improve the memory scaling behavior. This note outlines changes made to the software framework to support this migration.

1 Introduction

Run 3 of the Large Hadron Collider (LHC) at CERN is planned to start in 2021, with an ever-increasing demand for computing to simulate, reconstruct, and analyze the data recorded by the experiments. Meanwhile, the computing landscape is also evolving. Commodity CPU clock speeds have not increased significantly for some time now; rather, systems are including more cores and wider vector units. Further, memory prices have not been decreasing recently. Thus, the ratio of memory to cores will likely be decreasing in deployed systems.

Typical computational problems in high-energy physics are embarrassingly parallel, involving the processing of independent events. These can be trivially parallelized simply by running multiple jobs on a given host. However, these jobs tend to require a lot of memory: ATLAS reconstruction requires at least 4 GB of memory per job. If the number of cores per system grows faster than the available memory, then eventually one cannot keep all the cores occupied in this way. To fully use all the cores, one must reduce the memory required per core.

For Run 2, ATLAS reduced memory requirements with *multiprocessing*. After initialization, a job forks subprocesses processing events in parallel. Due to the copy-on-write behavior of the operating system, unmodified memory pages remain shared between all subprocesses, yielding memory savings up to about a factor of two [1]. This, however, is not expected to be sufficient for Run 3. Fully multithreaded (MT) tests have demonstrated substantial additional memory savings [2]; therefore, ATLAS is adopting this solution [3].

Section 2 discusses changes that were made to the framework to support multithreading. Section 3 briefly discusses the use of the multithreaded framework in the high-level trigger, and Section 4 discusses the migration of algorithmic code and a compiler plugin that is used to statically check for thread-safety violations. Section 5 then discusses the current status and presents some preliminary scaling results.

[*] e-mail: snyder@bnl.gov

2 Athena framework and modifications for multithreading

The ATLAS offline software framework, Athena [4], is based on the Gaudi project [5], which is developed jointly with LHCb and other experiments. An Athena application consists of dynamically-loadable *components*, including Algorithms, Services, and Tools; see Fig. 1. Algorithms process data which reside in a separate ‘event store’; they read objects (identified by type and a string key) from the store and write new objects back to the store. Ideally, an Algorithm itself does not store any event data. Services are singletons; examples include the event store as well as error logging and random number generation. Tools serve as helpers for other components. They are ideally accessed via an abstract interface, and may be owned by Algorithms, Services, or other Tools.

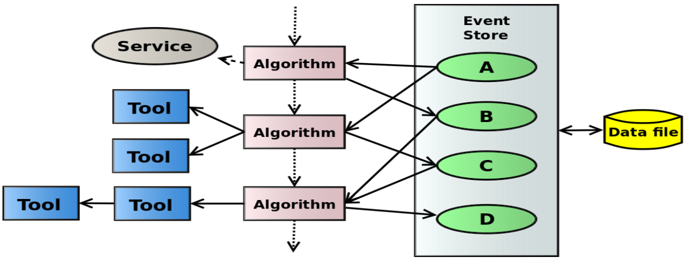


Figure 1. General structure of an Athena application.

In the currently-deployed, serial, version of Athena, Algorithms are executed in a fixed sequence, which is defined during job configuration. To enable parallelism within an event, Algorithms are modified so that they declare their inputs from, and outputs to, the event store. This allows the construction of a dependency graph for Algorithms. A scheduler [6] then looks for Algorithms that have all their inputs satisfied and queues them for execution, using the task facility of the Intel Threading Building Blocks library (TBB) [7]. Further, by allowing multiple event stores (‘slots’), multiple events can also be processed in parallel.

Algorithms declare their data dependencies as a side effect of data access [8]. Every Gaudi component has a set of named *properties*, which are set via job configuration. To access data, an Algorithm should first declare a property with the special type of either `ReadHandleKey` or `WriteHandleKey`:

```
class MyAlg : public AthReentrantAlgorithm { ...
  // Declare that this Algorithm will read an object of type X.
  // The name of the property is ‘XKey’, and its default value is ‘x’.
  SG::ReadHandleKey<X>  m_xKey { this, "XKey", "x" };
  // This Algorithm will also output an object of type Y.
  SG::WriteHandleKey<Y> m_yKey { this, "YKey", "y" };
```

To actually read or write data, one creates a `ReadHandle` or `WriteHandle` from the `HandleKey` object; the handles then act like smart pointers. (`Handle` objects are distinct from `HandleKey`s to avoid requiring mutable state in Algorithm members.)

```
StatusCode MyAlg::execute_r (const EventContext& ctx)
{
  // Create handle objects.
  // The ctx argument identifies the event being processed.
  // If omitted, it is read from thread-local storage.
  SG::ReadHandle<X>  xH (m_xKey, ctx);
  SG::WriteHandle<Y> yH (m_yKey, ctx);
  // New object to be written to event store.
  auto y = std::make_unique<Y>();
  // Initialize new object; the handle dereference reads the X object
  // at this point.
  y->fillFrom (*xH);
  // Record new object in store.
  ATH_CHECK( yH.record (std::move (y)) );
  // The new object may be modified through the handle.
  // Once the WriteHandle is destroyed, the object becomes immutable.
  yH->setSomething();
  return StatusCode::SUCCESS;
}
```

Handles can be set from the job configuration (expressed in Python):

```
# Create a new algorithm, overriding the XKey and YKey properties.
alg = MyAlg (’myalg’, XKey = ’myX’, YKey = ’myY’)
topSequence += alg # Add algorithm to the configuration.
```

The scheduler infers an Algorithm’s data dependencies from the set of HandleKeys that the Algorithm has declared (ignoring any that have a null key). In addition to Algorithms, Tools may also declare HandleKeys; such dependencies are automatically propagated to the owning Algorithm. Services should not declare HandleKeys.

In the ATLAS Run 2 ‘xAOD’ event data model [9], objects are stored in a structure-of-arrays fashion: object data consist of a set of named vectors of various types. Even after an object has been recorded, new named variables, called ‘decorations,’ may be dynamically added to an object, provided that they do not conflict with existing variables. Decorations are also made known to the scheduler, using additional Handle types:

```
class MyAlg : public AthReentrantAlgorithm { ...
  // Read decoration ‘indec’ and write decoration ‘outdec’
  // on object ‘x’ of type ‘XContainer’.
  SG::ReadDecorHandleKey<XContainer>  m_inDecKey
    { this, "InDecKey", "x.indec" };
  SG::WriteDecorHandleKey<XContainer> m_outDecKey
    { this, "OutDecKey", "x.outdec" };
...
StatusCode MyAlg::execute_r (const EventContext& ctx) {
  // Declare handles corresponding to the handle keys.
  // Second template argument is the type of the decoration.
  SG::ReadDecorHandle<XContainer, float> inDec (m_inDecKey, ctx);
  SG::ReadDecorHandle<XContainer, float> outDec (m_outDecKey, ctx);

  // Loop over objects in the container.
  for (const X* x : inDec) {
    // Copy the decoration for this container element.
    outdec(*x) = indec(*x);
```

Decoration handles act like ordinary read handles, except that they have additional methods to read and write the decorations themselves. Both read and write decoration handles will set up a read dependency on the underlying object to which the decorations are attached.

A common use case is that of *reprocessing*, in which some objects existing in an input file are remade from other data in the file. To support this case, Athena arranges that objects in the input file with keys that match any declared WriteHandleKeys are ignored, rather than being read. Alternatively, objects existing in an input file may need to be modified, for example to correct a deficiency in an earlier reconstruction version. For this case, an object being read from an input file may be renamed. For example, if an input file contains a container called '`Electrons`', one may set up job configuration to rename '`Electrons`' to, say, '`Electrons_in`' on input. One can then schedule an Algorithm to apply a correction and produce '`Electrons`' from '`Electrons_in`'. Any downstream Algorithms that used the '`Electrons`' container will then use the corrected versions without needing further modification.

Besides event data, reconstruction Algorithms may depend on *conditions* data, which are valid over some range of events or time. Since the framework may be processing multiple events at one time, it needs to be able to manage having potentially several versions of a conditions object active at any one time. Conditions data are kept in a separate 'conditions store' analogous to the event store; objects recorded here are containers that can hold multiple versions of a conditions object. A special Algorithm, `CondInputLoader`, runs at the start of an event and ensures that the versions of conditions objects needed for the current event are present in the store. A garbage collection procedure, invoked from the event loop, removes conditions object versions when they are no longer needed. In some cases, Algorithms apply some transformation to conditions data. In this case, the transformation is factored out into a separate 'conditions Algorithm', which acts on data in the conditions store in the same way in which other Algorithms act on data in the event store. Algorithms (and conditions Algorithms) access conditions data using additional handle types, `ReadCondHandle` and `WriteCondHandle`, and the corresponding key classes. These handles transparently resolve to the proper version of the conditions object appropriate for the current event. They also make the conditions dependencies known to the scheduler, which uses them to run conditions Algorithms at the appropriate times. See also [10] for more information about conditions in the multithreaded framework.

The dependency information provided by handle declarations, referred to as 'data flow,' is one of the inputs to the Gaudi scheduler [6], It is also possible to declare 'control flow' rules, that explicitly state the sequence in which some set of Algorithms must run. Control flow may also be used to implement filtering, which terminates processing of an event at a given point if a condition is not satisfied. The scheduler uses both data and control flow rules to find an efficient order for executing Algorithms. The scheduler was designed to have low response time and constant amortized complexity, and to scale well to many threads. It is capable of looking ahead in the dependency graph to maximize intra-event parallelism, which is preferred to inter-event parallelism.

By default, a given Algorithm instance cannot be executing simultaneously in more than one thread. To achieve more parallelism, an algorithm may be declared as *clonable*. In this case, multiple copies of the Algorithm instance are made and may be executing simultaneously (though any given copy will be executing in no more than one thread at a time). An Algorithm is declared as clonable by overriding the virtual function '`bool isClonable() const`' to return `true`; the number of copies created is then controlled by the `Cardinality` property of the Algorithm. Finally, an Algorithm may be declared as *reentrant*. In this case, only a single instance of the Algorithm is used, but that instance may execute in multiple threads simultaneously. An Algorithm is declared as reentrant by deriving from the special base class `AthReentrantAlgorithm`. A non-reentrant Algorithm has a non-`const` method `execute()` that is called for every event. For a reentrant Algorithm, the corresponding interface is instead:

```
  virtual StatusCode execute_r (const EventContext& ctx) const;
```

The context argument identifies the particular event being processed. Since the execute method of a reentrant Algorithm is `const`, the Algorithm should be thread-safe as long as `const`-correctness is strictly observed. But if the Algorithm does have any internal mutable state, then that must be written to be explicitly thread-safe.

The I/O components of Athena have been made thread-safe. Currently, there is one service for reading event data, one for writing event data, and one for reading conditions data. Each of these is serialized internally, but they may run concurrently with each other. The ROOT [11] `TTreeCache` is used to perform read-ahead and caching of event data to prevent I/O thrashing when multiple events are being processed simultaneously. Additional parallelism on writing is gained by using the implicit multithreading mode of ROOT. See [12] for more information on I/O in the multithreaded framework.

Gaudi supports a form of structured callbacks called ‘incidents.’ At any time, any piece of code can raise an incident of some type. Arbitrary Gaudi components can then register to receive a callback when a given incident type is raised. The callbacks are brokered by a Gaudi service, IncidentSvc. Examples of incident types are starting and ending a file, starting and ending an event, and so on. This is problematic for MT because incidents could in principle be asynchronous with respect to event processing, and they don’t respect event boundaries.

Upon examining how incidents were used in the reconstruction, it was found that almost all were used to signal discrete state changes and were generated outside of Algorithms, from the event loop. Incident handling was therefore redesigned for MT: rather than having incidents being called directly from the IncidentSvc, they are instead called from special Algorithms that are run at the beginning and end of event processing. Incidents themselves are extended to include an event context. Algorithms no longer receive incidents directly. Instead, incidents are received by services, which can, if needed, retain data separately for each active event context. Algorithms can then call these services in order to observe the effects of incidents.

Random number generation is another issue that can be problematic for MT jobs. The generator state must be protected against concurrent access; but using locking or thread-local storage for every random number call can add a significant performance overhead. Further, the sequence of random numbers should be reproducible from run to run, regardless of the order in which Algorithms are scheduled.

In Athena, this is managed by a service, `AthRNGSvc`. Clients call this to get a ‘wrapper’ object that contains an array of generator states, one for each event slot. The client Algorithm identifies itself to the service, so that each Algorithm gets a distinct wrapper object. The generator state can then be retrieved from the wrapper and used without further synchronization. This is safe because no algorithm can be executing on the same event slot in more than one thread. An example of the usage of `AthRNGSvc` is shown below.

```
  ServiceHandle<IAthRNGSvc> m_rngSvc;  // Reference to service.
  ....
StatusCode MyAlg::execute_r (const EventContext& ctx) { ...
  // Get wrapper object.
  ATHRNG::RNGWrapper* wrapper = m_rngSvc->getEngine (this);
  // Reseed from current run and event number.
  wrapper->setSeed (this->name(), ctx);
  // Get random generator.
  CLHEP::HepRandomEngine* engine = wrapper->getEngine();
```

3 Use in the high-level trigger

Besides offline reconstruction, an important use case for the ATLAS framework is the high-level trigger (HLT), which reuses many of the offline components. In the Run 2 HLT, all trigger processing takes place within a single Gaudi Algorithm, which internally steers the execution of trigger-specific algorithms. In practice, this leads to substantial code duplication, as the same Algorithm often needs to exist in both offline and HLT versions. For Run 3, the HLT steering has been redesigned to use the common Athena scheduler to run HLT Algorithms, allowing the same Algorithm classes to be used both offline and for the trigger.

A key requirement for the HLT is the ability to do reconstruction only within a geometrically limited region of interest (ROI). This is implemented in the multithreaded framework via an 'EventView'. The classes used to access event data, `ReadHandle` and `WriteHandle`, communicate with the event data store only through an abstract interface. An EventView object implements this same interface but provides only a subset of the detector data. In this way, Algorithms that access the data using handles can transparently be restricted to the subset of event data provided by an EventView. At the start of processing, a specialized Algorithm creates the EventView, populates it with region-specific data, and requests that the scheduler run a sequence of Algorithms in the context of the view. Once processing has finished in the EventViews for all ROIs, the results are concatenated and entered into the primary event data store.

See [13] for additional information about the implementation of the HLT software for Run 3.

4 Algorithmic code migration and static code checker

Numerous changes must be made to algorithmic code in order to work properly with threading. These include:

- Use handles to access event and conditions data.
- Algorithms that transform or cache conditions data must be changed to use a conditions Algorithm.
- Some existing Algorithms were relying on modifying data objects that other Algorithms had recorded in the event store. These must be redesigned.
- Avoid thread-unfriendly constructs, including non-`const` static data and `const`-correctness violations.
- Services and reentrant Algorithms must be explicitly thread-safe.

ATLAS has developed a static checker [14] to assist in locating some categories of thread-unfriendly code. This was inspired by the static checker used by CMS [15], but the ATLAS checker is written as a plugin for gcc [16]. As this is the primary compiler used by ATLAS, it can be enabled for all compilations by the ATLAS build system. Problems for which it checks include the use of non-`const` static data and `const`-correctness violations, including use of mutable members, casting away `const`, and returning non-`const` pointer members from a `const` member function. The checker operates on an 'opt-in' basis, with authors tagging packages or source files to be checked. Violations detected by the checker can be suppressed by annotating the source using macros that expand to custom C++ attributes. For example, discarding `const` would normally elicit a warning, but this may be suppressed:

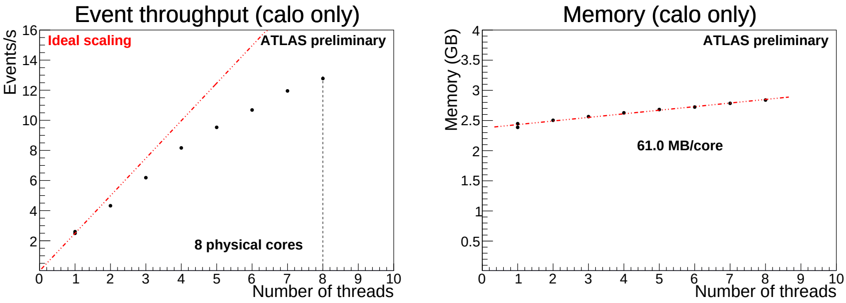
```
const int * y = ...;
int* yy ATLAS_THREAD_SAFE = const_cast<int*> (y);
```

In addition to identifying thread-safety issues, the static checker also enforces some elements of the ATLAS coding standards, in particular naming conventions.

5 Status and preliminary results

At the present time, almost all of the framework functionality thought to be needed for the MT migration is complete and available. Migration of algorithmic code is well underway. The task of changing event data access to use handles is essentially complete. Migration to the new conditions infrastructure is continuing, as well as making thread-safety fixes. The goal is to have the full reconstruction working as a MT job by the end of 2019, leaving 2020 for validation, debugging, and performance improvements.

Fully MT jobs have been successfully run for the calorimeter subset of the reconstruction on simulated data. Some preliminary scaling data for this job are shown in Fig. 2. For these tests, writing the output was disabled; this is a known bottleneck, but it is expected to become less important as more of the reconstruction is implemented. The CPU scaling behavior is promising for this stage of the project; work continues to identify and resolve instances of lock contention that affect the scaling. The memory usage is seen to increase very modestly with the number of threads.



(a) CPU scaling

(b) Memory scaling

Figure 2. Scaling of CPU time and memory required versus number of threads for a calorimeter-only reconstruction job with output disabled reading simulated data. Tested on an Intel Xeon X5560 system with eight physical cores.

6 Summary

In preparation for Run 3, ATLAS is redesigning its offline and trigger software to become fully multithreaded. This is in order to reduce the amount of memory required per core, but it will also serve as a foundation for future developments to adapt to expected trends in hardware, such as the use of heterogeneous systems. The migration is currently well underway, with some selected pieces of the reconstruction already working in a fully multithreaded job. It is expected to be complete before Run 3 starts in early 2021.

This work is supported in part by the U.S. Department of Energy under contract DE-AC02-98CH10886 with Brookhaven National Laboratory.

References

[1] S. Binet et al., *Multicore in production: Advantages and limits of the multiprocess approach in the ATLAS experiment*, in *Proceedings of the 14th International Workshop on Advanced Computing and Analysis Techniques in Physics Research (ACAT 2011): Uxbridge, UK* (2012), J. Phys. Conf. Ser. **368**, 012018, `https://doi.org/10.1088/1742-6596/368/1/012018`

[2] G.A. Stewart et al., *Multi-threaded software framework development for the ATLAS experiment*, in *Proceedings of the 17th International Workshop on Advanced Computing and Analysis Techniques in Physics Research (ACAT 2016): Valparaiso, Chile* (2016), J. Phys. Conf. Ser. **762**, 012024, `http://doi.org/10.1088/1742-6596/762/1/012024`

[3] C. Leggett et al., *AthenaMT: upgrading the ATLAS software framework for the many-core world with multi-threading* (2017), J. Phys. Conf. Ser. **898**, 042009, `http://doi.org/10.1088/1742-6596/898/4/042009`

[4] P. van Gemmeren, D. Malon, *The event data store and I/O framework for the ATLAS experiment at the Large Hadron Collider*, in *IEEE Int. Conf. on Cluster Computing and Workshops, 2009, New Orleans, USA* (2009), pp. 1–8, `http://doi.org/10.1109/CLUSTR.2009.5289147`

[5] G. Barrand et al., *GAUDI — A software architecture and framework for building HEP data processing applications* (2001), `https://gitlab.cern.ch/gaudi/Gaudi` [accessed 2018-11-26]

[6] I. Shapoval, Ph.D. thesis, Kharkov, KIPT (2016-03-03), `http://inspirehep.net/record/1503877/files/CERN-THESIS-2016-028.pdf`

[7] *Intel threading building blocks*, `https://www.threadingbuildingblocks.org` [accessed 2018-11-26]

[8] ATLAS Collaboration, *Event data access in AthenaMT*, `https://twiki.cern.ch/twiki/bin/view/AtlasComputing/MultiThreadingEventDataAccess` [accessed 2018-11-26]

[9] A. Buckley et al., *Implementation of the ATLAS Run 2 event data model*, in *Proceedings of the 21st International Conference on Computing in High Energy and Nuclear Physics (CHEP 2015): Okinawa, Japan* (2015), J. Phys. Conf. Ser. **664**, 072045, `https://doi.org/10.1088/1742-6596/664/7/072045`

[10] C. Leggett et al., *Conditions data handling in the multithreaded ATLAS framework*, in *Proceedings of the 23rd International Conference on Computing in High Energy and Nuclear Physics (CHEP 2018): Sofia, Bulgaria* (2018), EPJ Web Conf.

[11] R. Brun, F. Rademakers, Nucl. Inst. Meth. **389**, 81 (1997), `http://root.cern.ch`

[12] J. Cranshaw et al., *I/O in the ATLAS multithreaded framework*, in *Proceedings of the 23rd International Conference on Computing in High Energy and Nuclear Physics (CHEP 2018): Sofia, Bulgaria* (2018), EPJ Web Conf.

[13] S. Martin-Haugh et al., *Implementation of the ATLAS trigger within the ATLAS Multi-Threaded Software Framework AthenaMT*, in *Proceedings of the 23rd International Conference on Computing in High Energy and Nuclear Physics (CHEP 2018): Sofia, Bulgaria* (2018), EPJ Web Conf.

[14] S. Snyder, `https://gitlab.cern.ch/atlas/atlasexternals/tree/master/External/CheckerGccPlugins` [accessed 2018-11-26]

[15] T. Hauth, P. Gartung, *Clang CMS*, `https://github.com/gartung/clang_cms` [accessed 2018-11-26]

[16] *The GNU Compiler Collection*, `http://gcc.gnu.org` [accessed 2018-11-26]