I/O in the ATLAS multithreaded framework

Jack Cranshaw[1,*], *David* Malon[1], *Marcin* Nowak[2], and *Peter* Van Gemmeren[1]on behalf of the ATLAS collaboration

[1]Argonne National Laboratory, 9700 S Cass Ave, Argonne, IL 60439
[2]Brookhaven National Laboratory, Upton, NY 11973

Abstract. Scalable multithreading poses challenges to I/O for the ATLAS experiment. The performance of a thread-safe I/O strategy may depend upon many factors, including I/O latencies, whether tasks are CPU- or I/O-intensive, and thread count. In a multithreaded framework, an I/O infrastructure must efficiently supply event data to and collect it from many threads processing multiple events in flight. In particular, on-demand reading from multiple threads may challenge caching strategies that were developed for serial processing and may need to be enhanced. This I/O infrastructure must also address how to read, make available, and propagate in-file metadata and other non-event data needed as context for event processing. We describe the design and scheduling of I/O components in the ATLAS multithreaded control framework, AthenaMT, for both event and non-event I/O. We discuss issues associated with exploiting the multithreading capabilities of our underlying persistence technology, ROOT, in a manner harmonious with the ATLAS framework's own approach to thread management. Finally, we discuss opportunities for evolution and simplification of I/O components that have successfully supported ATLAS event processing for many years from their serial incarnations to their thread-safe counterparts.

ATL-SOFT-PROC-2018-027
24 November 2018

1 Introduction

The next period of data taking at the LHC will stress the ATLAS[1] computing infrastructure in two important ways, the event rate and the event complexity. Combined with computing architectures which scale by adding cores and coprocessors rather than clock cycles, this requires an approach which emphasizes parallelism, either using multiple processes across the cores or using multithreading within a single process. For memory constrained processes such as event reconstruction of raw data, this requires a multithreaded approach due to the increasing event complexity (expanding memory footprint) and the expanding core count per node (reduced memory/core). Moving data in and out of these processes is a necessary component to their efficient utilization.

2 Use cases

These are the three primary use cases for ATLAS and their expected uses of multicore parallelism:

*e-mail: cranshaw@anl.gov

- *High Level Trigger (HLT)*: The increasing complexity of events challenges both the memory capacity and throughput of the computing farms used for the HLT. Improvements here directly impact Run 3 data taking. The HLT uses ATLAS raw event data format[2] as its persistent storage.
- *Reconstruction/Simulation*: The problem here is the same as for the HLT, memory and throughput exceed the rate that serial Athena can handle. Reconstruction/Simulation uses the ATLAS ROOT[3] data model[4] except for trigger simulations.
- *Derivation and Analysis*: Derivations and Analysis are data reduction activities, where derivations are centrally produced while analysis is processed independently. These activities are more I/O limited and less memory challenged. Although a multithreaded approach may help in some areas, a multi-process approach may be a better way to utilize multiple cores. Derivation uses the ATLAS ROOT data model.

As noted in the final use case, a single approach to utilizing multi-core architectures is insufficient to meet ATLAS needs for Run 3, so it is expected that in addition to the multithreaded approach described in this paper there will be continued use of multiprocess[5] running.

2.1 AthenaMT

To address the needs of the first two use cases ATLAS has developed and is deploying a multithreaded version of its event processing framework called AthenaMT[6]. It shares components with the concurrent Gaudi framework[7] used by the LHCb experiment. Gaudi is a task-based framework built on top of Intel's Thread Building Blocks (TBB)[8]. The framework loops over input events and processes them in parallel in event 'slots'. Each AthenaMT slot has its own context and event data store based on the ATLAS whiteboard technology StoreGate. Each slot is designed to operate independently and consume events as fast as it can receive data. A scheduler schedules algorithm execution based on data dependency between algorithms. These algorithms can themselves be singletons, reentrant, or cloned.

Within AthenaMT the supporting services, including the I/O services, which read and write data from/to the event store, run outside the context of the event slots within the event loop. These services must support one of the following behaviors to be thread-safe within AthenaMT.

- Be stateless, i.e., not depend on the event context.
- Be 'slot-safe' so that when called with the event context they return the proper response.
- Be mutexed/locking and not support parallel access.

3 Multithreaded I/O for events

The basic I/O infrastructure within Athena (both serial and multithreaded) is shown in figure 1. Input data moves from files to the event store through several layers of services. On output, data is retrieved from the event store by an output stream and then passes through these same services which write it to a file (or increasingly to some sort of output buffer). The I/O layers shown serve specific purposes:

- The conversion service layer handles any transformations between the persistent format stored in the file and the transient format that the algorithms expect to see in the event store. Multiple type-specific converters are managed by the conversion service.
- The input services, such as the PoolSvc, manage the connection between the Athena/Gaudi conversion layer and the technology specific persistency layer.

- The persistency services are technology specific and handle the settings and actions associated with reading and writing from/to a specific file. Multiple persistency services can operate in parallel for separate data sources.

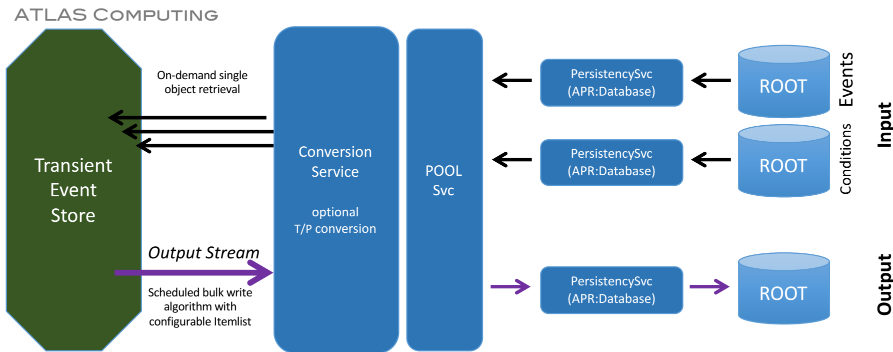

Figure 1: Basic components of the I/O services in Athena when reading and writing from ROOT files.

The first step in evolving the Athena I/O services was to find the finest granularity of locking that would make them thread-safe. This was accomplished by locking the persistency services and the individual converters used by the conversion service. The persistency services require locking because they contain handles to the ROOT file, and accesses to this handle cannot be done in parallel.

3.1 Multithreading on input

At the same time as locking on the persistency service was introduced for thread-safety, a feature was added so that we could create separate persistency services for different objects/branches within a ROOT file, which could then be accessed in parallel using separate file handles. The converters require locking because some of them contain internal state, but by locking at the converter level, individual converters can run in parallel as shown in figure 2.

The move to multithreading also affected how ATLAS does caching on input, both for ROOT files and for raw event data files. ATLAS already makes extensive use of ROOT's TTreeCache system when reading ROOT data. This system was designed for reading data sequentially, whereas a multithreaded framework will process events in no particular order. This, combined with on-demand reading, can lead to cache thrashing at cache boundaries, when retrieval of a data object for a not-yet-cached event causes the cache to be flushed while the framework is still processing an event that may still need to retrieve data from the current cache. This was observed in ATLAS data processing. A solution was implemented at the ROOT level and has now been incorporated into the ROOT code base. Raw event data had previously not implemented caching as it read full chunks of bytes for each event. For multithreaded input this required changes to the data provider service to provide a separate cache of event data for all configured slots. The interface to the data provider service was updated to be event context aware, and the conversion service forwards the event context to that service when objects are accessed in StoreGate. When data for a new event is requested for a given slot, the previous event is flushed from the cache for that slot.

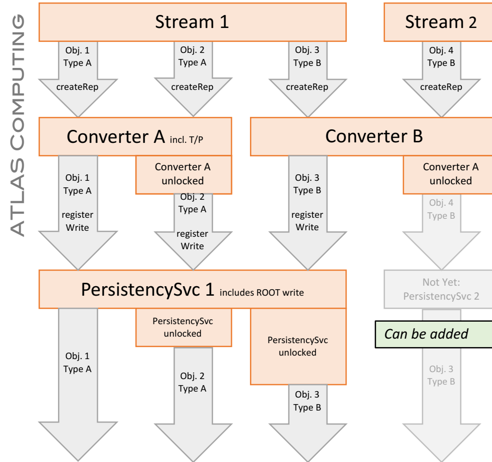

Figure 2: Opportunities for parallelism within the conversion service.

3.2 Multithreading on output

In a similar manner to using separate persistency services for different objects in an input file, separate persistency services can also be used if writing multiple output streams. AthenaMT also makes use of ROOT's implicit multithreading capability where the data is passed to ROOT, and it handles multithreaded writing to the file. In the future, we also plan to improve the access to data in the transient event store by the output stream, which is currently a serial process.

3.3 ROOT dependencies

The ROOT-technology implementation of the Athena I/O layer explicitly relies on thread-safety in two related ROOT features: C++ reflection and actual file I/O. For example, the information about the object shape is retrieved from ROOT::TClass in many different places with a high chance of different threads doing it concurrently. As mentioned previously in the persistency service description, our parallelism is also limited by how ROOT implements its file handles. ROOT is constantly improving its multithreading capabilities and we will continue to leverage, and where possible contribute to, these improvements.

4 Non-event data: metadata and conditions

Within a slot the event store is well defined and unshared. The same is not true for conditions data (calibrations, temperatures, etc.), which are indexed by interval of validity, and metadata (production parameters, event counts, etc.), which are typically associated with a given input file. These are shared across events. For serial Athena this was not a problem. The conditions or metadata store contained the objects appropriate for the current event, with the contents managed by associated I/O services. In a multithreaded environment, however, multiple versions of a given condition may be needed concurrently by different slots. The situation is similar for metadata, where processing events in parallel involves processing files in parallel.

Both conditions and metadata are stored in separate instances of StoreGate similar to the separate event stores for each event slot. A solution was developed for accessing conditions[9] in the conditions store where the conditions objects are not stored directly in the store, but in containers which are indexed by the interval of validity. The event context for a given slot, as it contains the run and event number and timestamp, is used to choose the correct interval of validity. Special conditions handles make these containers invisible to the algorithms, which simply request an object appropriate to their event context. A similar approach has worked for metadata, where the index is the file identifier. Two other differences for metadata are that the system must support writing as well as reading and that metadata must be propagated for eventless files. Consequently, the semantics of the conditions and metadata systems have been kept parallel, but they require different implementations. Both systems are in the process of being deployed, but are still under revision as more workflows are tested.

5 Conclusion

Athena I/O services have been upgraded to support multithreaded Athena: AthenaMT. This includes caching of multiple events for analysis in parallel both for data stored in ATLAS raw event data format and data stored in ROOT. Parallelism has been improved and expanded within the I/O services and locking has been implemented where necessary. We maximize our use of, and contribute directly to, multithreading tools developed by others, such as within ROOT itself. A system for handling non-event data (conditions and metadata) is in the later stages of development. This means that ATLAS now supports event processing on multi-core architectures for both multithreaded processing (AthenaMT) and multiprocess processing (AthenaMP). This process has also provided an opportunity to refactor and in some instances simplify the existing services based on experience running serial Athena.

Copyright 2018 CERN for the benefit of the ATLAS Collaboration. CC-BY-4.0 license.

References

[1] ATLAS Collaboration, JINST 3 **S08003** (2008)
[2] ATLAS TDAQ Collaboration, JINST 11 no.06, **P06008** (2016)
[3] R. Brun et. al., **A389** 81-86 (1997)
[4] J. Cranshaw, et. al.,PoS ICHEP2016 **852** (2016)
[5] P. Calafiura, et. al.,J.Phys.Conf.Ser. **664** no.7, 072050 (2015)
[6] C. Leggett, et. al., J.Phys.Conf.Ser. **898** no.4, 042009 (2017)
[7] M. Clemencic, et. al., J.Phys.Conf.Ser. **608** no.1, 012021 (2015)
[8] J. Reinders *Intel Threading Building Blocks* (O'Reilly Media 2007).
[9] C. Leggett, et. al, PoS ICHEP2016 **188** (2016)