# Multithreading ATLAS offline software: a retrospective

*Attila* Krasznahorkay[1], *Charles* Leggett[2], *Serhan Alaettin* Mete[3], *Scott* Snyder[4,*], *Vakho* Tsulaia[2], and *Frank* Winklmeier[5] on behalf of the ATLAS Computing Activity

[1]CERN, CH-1211 Geneva 23, Switzerland
[2]Lawrence Berkeley National Laboratory, 1 Cyclotron Road, Berkeley, CA 94720, USA
[3]Argonne National Laboratory, High Energy Physics Division, 9700 S. Cass Avenue, Argonne, IL 60439, USA
[4]Brookhaven National Laboratory, PO Box 5000, Upton, NY 11973, USA
[5]University of Oregon, 1584 E 13th Ave., Eugene, OR 97403, USA

**Abstract.** For Run 3, ATLAS redesigned its offline software, Athena, so that the main workflows run completely multithreaded. The resulting substantial reduction in the overall memory requirements allows for better use of machines with many cores. This note will discuss the performance achieved by the multithreaded reconstruction, the process of migrating the large ATLAS code base, and tools and techniques that were useful in debugging threading-related problems.

## 1 Introduction

Run 3 of the ATLAS experiment [1] at the CERN Large Hadron Collider started last year, bringing with it new demands on computing. But in the years leading up to this, there were significant changes in the computing landscape. CPU clock speeds largely plateaued, with systems instead including more cores and wider vector units. Memory prices have also not been decreasing much. The result is that the ratio of memory to cores has tended to decrease in deployed systems.

While typical computational problems in high-energy physics tend to be embarrassingly parallel, involving the processing of independent events, they also tend to require a lot of memory. Requirements of 4–8 GB of memory per job are not uncommon. This means that if the core count of a system grows faster then the available memory, then eventually one will not be able to keep all the cores busy by running multiple independent jobs. To fully use all available resources, one must reduce the memory required per core.

For Run 2, ATLAS reduced memory requirements using *multiprocessing*. After initialization, a job forks subprocesses which then work on 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 [2]. This, however, was not sufficient for Run 3. Fully multithreaded (MT) solutions, however, had shown substantial memory savings by allowing much greater memory sharing. ATLAS therefore adopted this strategy for Run 3 [3–5].

---

*e-mail: snyder@bnl.gov

Section 2 introduces the Athena framework and summarizes the changes that were made in order for it to run multithreaded. Section 3 discusses the process of the migration, Section 4 covers diagnostics and techniques used for debugging, and Section 5 presents some performance results.

## 2  Athena framework and modifications for multithreading

The ATLAS software framework, Athena [6, 7], is built on top of the Gaudi project [8], developed jointly with LHCb and other experiments. As shown in Figure 1, an Athena application consists of dynamically-loadable *components*, including Algorithms, Services, and Tools. Algorithms read, process, and write items of event data contained in a separate 'event store'; ideally, they do not contain event data themselves. Services are singletons providing functionality such as error logging and metadata handling. Tools serve as helpers for other components and are usually accessed via an abstract interface. Tools may be owned by Algorithms, Services, or other Tools.
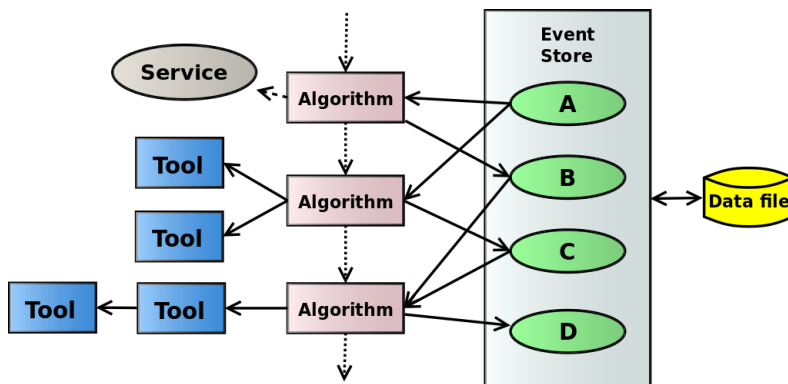


**Figure 1.** General structure of an Athena application.

When Athena is running serially, Algorithms are executed in a fixed sequence defined during job configuration. To allow for multithreaded execution, Algorithms are modified so that they declare their input and output data dependencies. A scheduler can then execute an Algorithm in a free thread once all its input dependencies are satisfied. Further, by using multiple event stores (or 'slots'), each holding a different event, multiple events can be processed in parallel (see Figure 2). However, simply declaring dependencies is not always sufficient. For example, the serial version of Athena contained Algorithms which modified existing data in the event store. This is not permissible in an MT environment, so those Algorithms needed to be redesigned.

Besides event data, Algorithms may also depend on 'conditions' data; that is, data which is valid over some range of events, such as calibration information. Algorithms declare their dependencies on conditions data in a similar manner to event data. In many cases, items of conditions data need to be transformed in various ways. This is also implemented by schedulable Algorithms which act on conditions data similarly to how other Algorithms act on event data [9].

Services, which are essentially global singleton objects, need to be made explicitly thread-safe, using locking or other techniques. However, a typical Algorithm, which retrieves data
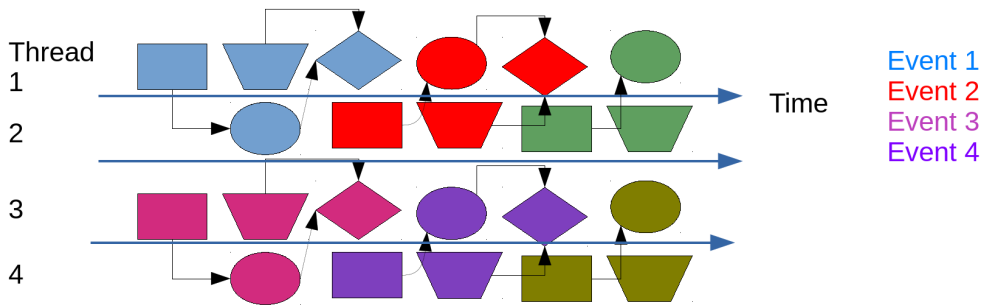
**Figure 2.** Multithreaded Athena. Colors represent different events, and shapes represent different Algorithms.

objects from the event store, processes them, and stores new objects back, will often not need to be explicitly aware of threading. It must, however, avoid thread-unfriendly constructs such as `const_cast` and static data. Where feasible, Algorithms should also be made reentrant (that is, with no mutable data, allowing the same instance to be used in multiple threads).

For Algorithms with more complicated requirements, a small library of helpers is available to factor out code important for thread-safety. These helpers use atomic operations rather than locking where possible to improve scalability. Some of these helpers include `SlotSpecificObj`<T>, which holds a separate `T` instance for each event slot, and allows access without locking; `CachedValue`<T>, which implements a value that can be set from multiple threads but always to the same value, and `LockedPointer`<T>, which bundles a pointer along with a lock and can be returned from an accessor function. There is also a set of container classes (bitset, hash maps, and a specialized container for conditions ranges) that allow for concurrent, lockless reads. Many of these are described in more detail in Ref. [10] and are available in Ref. [11]
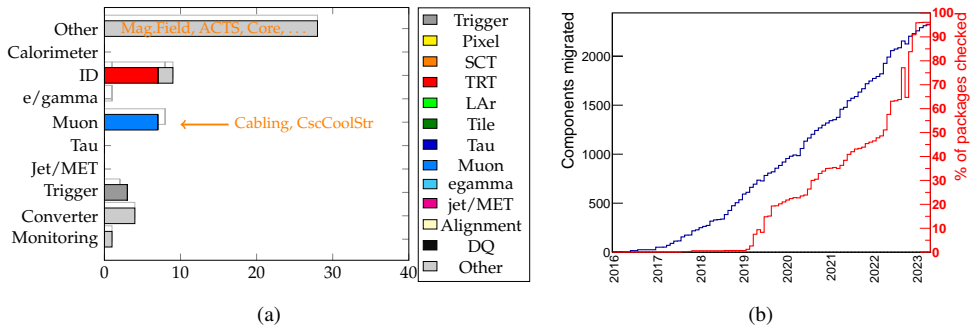
## 3 Multithreaded migration

Although most components were reasonably straightforward to migrate, the ATLAS code base is quite large: about five million lines of C++ (ignoring other languages, such as Python) comprising about 2000 packages and thousands of components. The migration was thus a multi-year project involving many people. It was tracked by semi-automated reports of components yet to be migrated, which were posted on a dedicated web page and presented at weekly meetings (see Figure 3(a)).

The migration also made use of a custom static checker [5, 12] to detect potential thread-safety problems, implemented as a gcc [13] plugin. The problems about which it warns include the use of non-`const` static data and `const`-correctness issues. For example, the checker will warn about the call in this example passing a non-const member pointer to another function from a `const` member:

```
1  void fee(int*);
2  struct S {
3    int* p;
4    void bar() const { fee(p); }
5  };
```

**Figure 3.** (a) Sample of semi-automatic report generated to track the progress of the multithreaded migration. (b) Number of components migrated for multithreading and fraction of packages that pass the thread-safety checker as a function of time.

Warnings may be suppressed on a case-by-case basis by annotating the source with macros that expand to custom C++ attributes known by the checker. The checker runs only on packages for which it is requested. This can be either explicitly declared package by package, or the checker can be requested to run on entire subtrees of the source tree from a configuration file.

The effort to make Athena multithreaded started around 2014 with small-scale prototyping. By mid-2018, most of the design for the core components was complete and reconstruction was working for the calorimeter on simulated data. At this point, work started to migrate the bulk of the reconstruction to run multithreaded. By early 2020, the full reconstruction was running on simulated data with an observed failure rate of about $10^{-4}$. By early 2021, the reconstruction was running on real data with no irreproducibilities with a failure rate of about $10^{-6}$. By mid-2021, the failure rate was reduced to $10^{-7}$. The multithreaded reconstruction was running in production by October 2021, reprocessing data from Run 2. Over this period, very approximately three to six full-time-equivalents were working on the migration. The progress over time at migrating components for multithreading and updating code to pass the thread-safety static checker is shown in Figure 3(b).

## 4 Testing, debugging, and diagnostics

Issues related to multithreading tend to be both rare and irreproducible. The fact that they are rare means that regular, high-statistics tests are needed. ATLAS set up procedures to run weekly tests of the latest reconstruction code over about 100 million events. This allowed seeing crashes of frequency $10^{-7}$ or less. This was invaluable in finding and tracking rare problems. However, because available resources did not allow running this more than about once a week, there was often a considerable turnaround time when testing fixes and adding code to help debug problems.

The fact that issues are irreproducible means that one needs to collect good diagnostics in the case of crashes. In a few special cases, ATLAS members were able to work with computing facility managers to obtain core dumps from failing jobs, or to attach debuggers to jobs that had gotten stuck. But mostly this means stack dumps.

On a crash, Athena will print out the currenty-executing Algorithm in each thread, followed by the ROOT [14] stack trace for all threads. This however is not always reliable. For

example, the ROOT stack trace code allocates dynamic memory, so if the program crashed due to corrupt heap, the ROOT stack trace would often fail as well, providing no information.

Therefore, Athena first produces a robust or 'fast' stack dump from the faulting thread [15]. This is carefully written to avoid any use of dynamic memory allocation and most use of C library functions that may depend on internal state. For example, `write/pipe/fork` are used directly rather than library functions like `printf/std::cout/popen`. Any memory needed is taken from the stack, or in a few cases, from pre-allocated memory regions. In addition, an alternate signal stack is defined (using `sigaltstack`), so that a trace can be generated even if the stack is exhausted or the stack pointer is corrupt. The trace starts with a dump of the machine registers; this proved to be essential to working out some crashes. In addition, the dump calculates addresses as offsets within each shared library. This is very useful for locating the precise point of the crash even when address space randomization is in use: one can simply search for the offset in a disassembly of the shared library to find the faulting instruction. An example of such a stack trace is given in Figure 4.

```
1  (pid=2120 ppid=8969) received fatal signal 11 (Segmentation fault)
2  signal context:
3    signo  = 11, errno = 0, code = 1 (address not mapped to object)
4    pid    = 0, uid = 0
5    value  = (0, (nil))
6    addr   = (nil)
7    stack  = (0, 202000, 0x7f8d68001a40)
8
9    rip: 0033:00007f8d7432da43 eflags: 0000000000010203
10   rax: 0000000000000000   rbx: 00007f8d68203b20
11   rcx: 0042676c41657669   rdx: 0000000000000000
12   r08: 00007f8d68000090   r09: 0000000000000060
13   r10: 00007f8d7cc1e3e8   r11: 00007f8d7cc3d418
14   r12: 000056534400fbb8   r13: 00007f8d7327e900
15   r14: 00007f8d7327e940   r15: 0000000000000000
16   rsi: 0000000000000000   rdi: 0000565343f20060
17   rbp: 00007f8d7327e8a0   rsp: 00007f8d7327e5d0
18    gs: 0000      fs: 0000
19
20 stack trace:
21  0x7f8d7432da43 HiveAlgB::execute() Control/AthenaExamples/
22     AthExHive/src/HiveAlgB.cxx:60:24   + 0x143 [build/libs/
23     libAthExHive_components.so D[0x32da43]]
24  0x7f8d83d4ba61 Gaudi::Algorithm::sysExecute(EventContext
25     const&) GaudiKernel/src/Lib/Algorithm.cpp:366:23   + 0x181
26     [build/libs/libGaudiKernel.so D[0x34ba61]]
```

**Figure 4.** Example of a robust stack dump (abridged).

On x86_64 platforms, the base pointer register (`rbp`) can be used as an additional general-purpose register. However, this implies that one needs to rely on debugging information to unwind the stack, in order to find the proper offset between the stack pointer and the start of the stack frame at any point within a given function. However, it was observed that because of

this, the stack trace would often not proceed beyond a virtual function call from an object with a corrupt virtual table, making it impossible to find from where the bad call actually occurred. To improve this, the Athena stack dump code detects when the stack trace is truncated early. In such a case, it injects a synthetic block of unwinding information based on the actual location seen in the stack. In most cases, this allows the stack trace to proceed past problems such as calls with a corrupt virtual table.

One of the most difficult problems to diagnose is heap corruption. Even in the single-threaded case, a visible crash may not happen until long after the actual corruption. While the exact strategies used are often quite specific to the particular issue being diagnosed, two that may be of interest are summarized here.

ATLAS uses TCMalloc [16] as the default memory allocator for production jobs. TC-Malloc maintains a per-thread cache of free blocks, sorted into size classes, with free blocks for each class organized as a singly-linked list. A rare crash was observed in which a forward pointer in the free list was corrupt. To diagnose this, ATLAS modified TCMalloc to add additional checking. When a block is freed, the forward link pointer is duplicated in the block and, if the block has enough room, a specific magic value is written into the block. When a block is removed from the free list, these words are checked and if they do not match what is expected, the program aborts. The consistency of the blocks at the start of the free list is also checked on all allocate/free operations. This modified version of TCMalloc very quickly located the problem that prompted it (a race condition leading to a double-deletion) after several weeks of failing to find it by other means.

In another case, heap corruption was seen where a forward pointer was being overwritten, but always with the same distinctive value, which could be seen in the `rax` register in the stack dump:

```
rax: 3fc0be57ef09fe55    rbx: 0000000151ed8d80
```

The value here is not a small integer, not a valid pointer, and not a likely floating-point number. The crash was also particularly rare, at a rate of below $10^{-6}$. While one might like to try to trap the write of this value in the debugger, this was not possible because only the value is known, not the address. Instead, ATLAS created a custom Valgrind [17] checker that logged all writes of that particular value:

```
static VG_REGPARM(2) void trace_store(Addr addr, SizeT size){
  if (size == 8) {
    unsigned long long val = *(unsigned long long*)addr;
    if (val == 0x3fc0be57ef09fe55) {
      VG_(printf)(" wrote %08lx %08llx\n", addr, val);
      VG_(get_and_pp_StackTrace) (VG_(get_running_tid)(),20);
```

This immediately located the problem (which was a rare race condition in writing a `std::vector` containing seed information for a random number generator).

## 5 Results and summary

In order to reduce the memory required per core, ATLAS has migrated its five-million-line offline code base to run multithreaded, roughly a five-year project. The resulting performance is excellent. Figure 5 shows the resulting performance as measured in mid-2021. The memory required scales at about 0.3 GB per thread, and the CPU scaling is very good up to at

least eight threads. Work has been continuing to further improve the performance. The multithreaded reconstruction is now in production for Run 3 data and reprocessing Run 2 data. This work is also expected to form a solid base for work on supporting heterogeneous systems for Run 4.
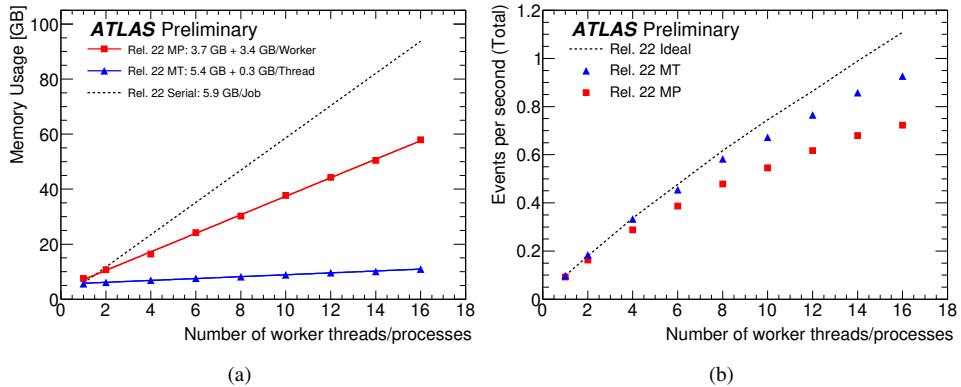


**Figure 5.** (a) Memory and (b) throughput scaling for full reconstruction as a function of number of threads/processes, comparing serial, multithreaded (MT), and multiprocessing (MP) jobs, as of the Athena release current in mid-2021. Tested on a 16-core Xeon E5-2630 processor with no hyperthreading. Results are for all phases of processing, including initialization, finalization, and output file merging. The jobs processed 250 events per thread, and the average number of collisions per crossing ($\langle\mu\rangle$) was about 50. The ideal line shown in (b) is not straight due to CPU thermal throttling at high loads. From Ref. [18].

# References

[1] ATLAS Collaboration, JINST **3**, S08003 (2008), `http://doi.org/10.1088/1748-0221/3/08/S08003`

[2] 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`

[3] 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`

[4] 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`

[5] S. Kama, C. Leggett, S. Snyder, V. Tsulaia, *The ATLAS multithreaded offline framework*, in *Proceedings, 23rd International Conference on Computing in High Energy*

*and Nuclear Physics (CHEP 2018): Sofia, Bulgaria, July 9-13, 2018* (2019), Vol. 214, p. 05018, `https://doi.org/10.1051/epjconf/201921405018`

[6] ATLAS Collaboration, *Athena*, `https://doi.org/10.5281/zenodo.4772550`

[7] 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`

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

[9] C. Leggett, I. Shapoval, S. Snyder, V. Tsulaia, *Conditions DataHandling in the Multithreaded ATLAS Framework*, in *Proceedings, 23rd International Conference on Computing in High Energy and Nuclear Physics (CHEP 2018): Sofia, Bulgaria, July 9-13, 2018* (2019), Vol. 214, p. 05031, `https://doi.org/10.1051/epjconf/201921405031`

[10] S. Snyder, *Concurrent data structures in the ATLAS offline software*, in *Proceedings, 24th International Conference on Computing in High Energy and Nuclear Physics (CHEP 2019): Adelaide, Australia, Nov 4-8, 2019* (2020), Vol. 245, p. 05007, `https://doi.org/10.1051/epjconf/202024505007`

[11] *Concurrent classes*, `https://gitlab.cern.ch/ssnyder/concurrentclasses`

[12] *CheckerGccPlugins*, `https://gitlab.cern.ch/atlas/atlasexternals/tree/master/External/CheckerGccPlugins`

[13] *The GNU Compiler Collection*, `http://gcc.gnu.org`

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

[15] *Robust stack trace*, `https://gitlab.cern.ch/ssnyder/robuststacktrace`

[16] *TCMalloc*, `https://github.com/google/tcmalloc`

[17] *Valgrind*, `https://valgrind.org/docs/manual/index.html`

[18] ATLAS Collaboration, *Performance of Multi-threaded Reconstruction in ATLAS*, ATL-SOFT-PUB-2021-002 (2021), `https://cds.cern.ch/record/2771777`