AthenaMT: a retrospective

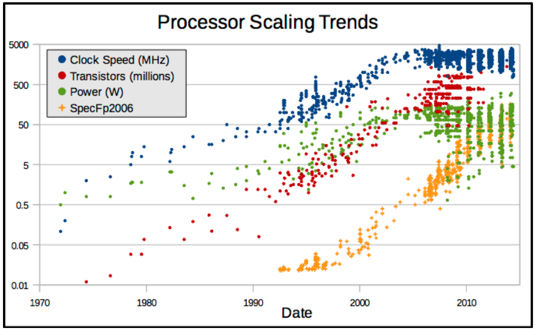
Scott Snyder
On behalf of the ATLAS Collaboration

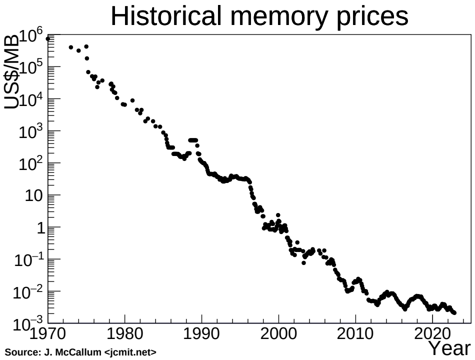
Brookhaven National Laboratory, Upton, NY, USA

May, 2023
CHEP 2023

Motivation

- For Run 3, ATLAS migrated its offline/trigger software to run fully multithreaded.
- Since $\sim$ 2010, CPU clock speeds have plateaued and memory prices have not decreased much.
- Processors getting more cores, so ratio of memory to cores tends to decrease.
- ATLAS reconstruction requires a large amount of memory ($\approx$ 6 GB job for serial reconstruction).
- Can't make full use of all cores simply by running multiple jobs on one machine. Need to reduce memory required per core.

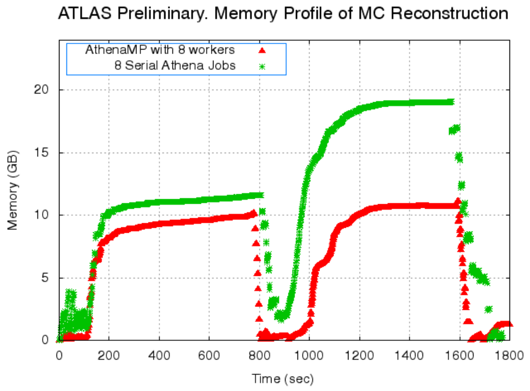




AthenaMP

For Run 2, ATLAS reduced memory requirements via *multiprocessing*.

Job forks subprocesses to process events in parallel. Memory is shared automatically via copy-on-write.



Yields significant memory savings but not sufficient for Run 3.

Go to a fully multithreaded solution.

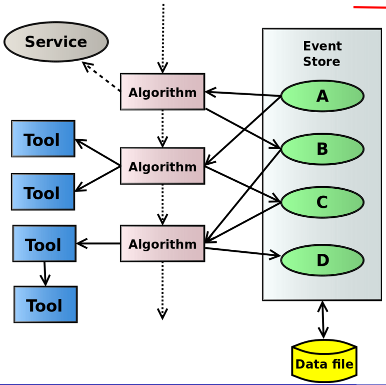
AthenaMT

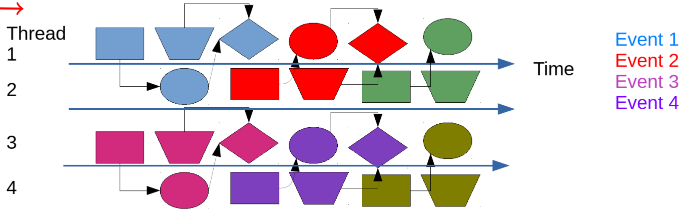
Serial execution: Single event at a time.

Execute algorithms on each event in a fixed order set during job configuration.

MT execution: Multiple events at a time.

Execute an algorithm in an available thread when its input dependencies are available.





Different shapes: different algorithms; different colors: different events.

Multithreaded migration

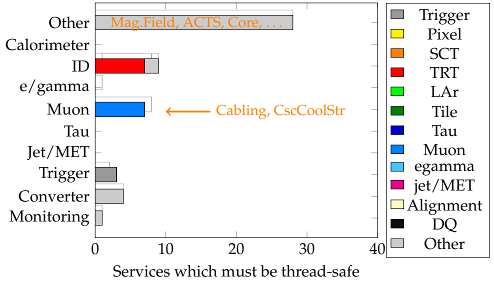
- Algorithms must declare their input and output data dependencies.
  - Modifying an object read from the event store is no longer allowed.
- Rework conditions access:
  - Declare dependencies.
  - Implement derived conditions using algorithms.
- Avoid thread-unfriendly code: use of statics, const-correctness violations.
- Services (global singleton objects) need to be explicitly thread-safe.
- Make algorithms reentrant (no mutable data) whenever possible.

ATLAS offline code (excluding externals)

About 5M lines of C++ code, 2000 packages, 7000 components.

Migration was a multi-year project involving many people.

Assisted by semi-automated reports of components yet to be migrated, plus a static checker to identify thread-unfriendly code.



Thread-safety static checker

- Migration helped by a custom static checker to detect possible thread-safety problems.
  - Mostly relating to const-correctness and use of static data.
- Implemented as a gcc plugin and enabled both for local development and release builds.

```
void fee(int*);
struct S {
  void bar() const
  { fee(p); }
  int* p;
};
```

```
x.cc:6:25: warning: Argument from member
'S::p' of type 'int*' in const member function
passed to non-const pointer argument of
function 'void fee(int*)'; may not be thread-safe
    6 |   void bar() const { fee(p); }
      |                      ~~~~~~
```

- Can be enabled on a package-by-package basis or on entire directory trees.
- Currently, almost all ATLAS offline code passes the checks.
- Also has checks related to naming conventions and other coding style issues.
- `gitlab.cern.ch/atlas/atlasexternals/tree/master/External/CheckerGccPlugins`

Coding for MT

Algorithms generally retrieve objects from the event store, process them, and then create and store new objects.

Algorithms generally don't need to be explicitly aware of threading.

But avoid unfriendly constructs like `static`, `const_cast`.

For algorithms with more complicated requirements, a small library of helpers is available to factor out code important for thread-safety.

Use atomics rather than locking when possible to improve scalibility.

`SlotSpecificObj<T>`

A vector of `T` instances, one per slot, that can be accessed without locking.

`CachedValue<T>`

A value that can be set from multiple threads, but always to the same value. Useful for caching results. Lockless. Also special cases for pointers, `unique_ptr`.

`LockedPointer<T>`

A pointer along with a lock; can be returned from an accessor.

Also have a set of container classes (bitset, hashmaps, conditions IOV map) allowing for concurrent, lockless reads.

Migration timeline

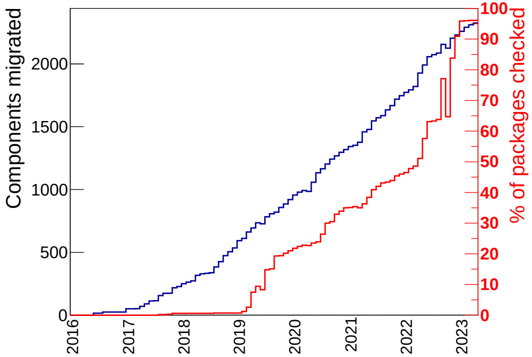
Plan as of mid-2018:

mid-2018: Calorimeter reconstruction working.
end 2019: Full reconstruction working.
2020: Debugging, validation, performance improvements.
early 2021: Run 3 starts.

Actual timeline:

Early 2020: Full reconstruction running on MC; failure rate $O(10^{-4})$.
Early 2021: Failure rate $O(10^{-6})$, no irreproducibilities. Running on data.
mid-2021: Failure rate $O(10^{-7})$. Performance tests on 64-core machines.
Oct 2021: Reprocessing of Run 2 data started. Performance improvements continue.

Progress over time at migrating components to MT and updating code to pass the thread-safety static checker:



Very roughly 3–6 FTEs directly working on this over this time period.

Debugging, testing, and diagnostics

- Issues related to threading tend to be both *rare* and *irreproducible*.
- Need regular high-statistics tests.
    - ATLAS ran weekly test runs of the latest reconstruction code over about 100M events.
    - Saw crashes of frequency 10^{-7} or less.
- Need good diagnostics in case of crashes.
    - In a few special cases, we were able to work with facility managers to obtain core dumps or attach debuggers to stuck jobs.
    - But mostly this means stack traces.

On a crash, print out algorithms running in all threads, followed by stack traces of all threads generated by ROOT.

```
Caught signal 11(Segmentation fault). Details:
  errno = 0, code = 1 (address not mapped to object)
  pid   = 0, uid = 0
  value = (0, 0)
  vmem = 1444.48 MB
  rss  = 808.676 MB
  total-ram = 15843.8 MB
  free-ram  = 248.289 MB
  buffer-ram= 208.316 MB
  total-swap= 63187 MB
  free-swap = 62330.1 MB
  addr  = 0

Event counter: 1
Slot   0 : Current algorithm = HiveAlgB
         : Last Incident = BeginIncFiringAlg:BeginEvent
         : Event ID      = [0,1,t:0,l:0]
Slot   1 : Current algorithm = <NONE>

===========================================================
There was a crash.
This is the entire stack trace of all threads:
===========================================================
```

Robust stack dump

- If the program state is corrupt, the ROOT stack trace may fail: It allocates memory, which will fail if the heap is corrupt.
- So we first produce a ‘fast’ robust stack dump from the faulting thread.
  - Avoids dynamic memory allocation and stdio/iostreams.
  - Define an alternate stack so we can proceed even in the case of a bad stack pointer.
  - On linux-gcc-x86_64 platforms, modify the stack unwinder to enable progress beyond an invalid frame (such as from a virtual call with a corrupt vtable).
  - Includes a dump of machine registers and offset within each DSO.

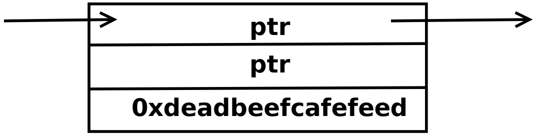
```
(pid=2120 ppid=8969) received fatal signal 11 (Segmentation fault)
signal context:
  signo  = 11, errno = 0, code = 1 (address not mapped to object)
  pid    = 0, uid = 0
  value  = (0, (nil))
  addr   = (nil)
  stack  = (0, 202000, 0x7f8d68001a40)

  rip: 0033:00007f8d7432da43 eflags: 0000000000010203
  rax: 0000000000000000   rbx: 00007f8d68203b20
  rcx: 0042676c41657669   rdx: 0000000000000000
  r08: 00007f8d68000090   r09: 0000000000000060
  r10: 00007f8d7cc1e3e8   r11: 00007f8d7cc3d418
  r12: 000056534400fbb8   r13: 00007f8d7327e900
  r14: 00007f8d7327e940   r15: 0000000000000000
  rsi: 0000000000000000   rdi: 0000565343f20060
  rbp: 00007f8d7327e8a0   rsp: 00007f8d7327e5d0
   gs: 0000     fs: 0000

stack trace:
 0x7f8d7432da43 HiveAlgB::execute() Control/AthenaExamples/
   AthExHive/src/HiveAlgB.cxx:60:24   + 0x143 [build/libs/
   libAthExHive_components.so D[0x32da43]]
 0x7f8d83d4ba61 Gaudi::Algorithm::sysExecute(EventContext
    const&) GaudiKernel/src/Lib/Algorithm.cpp:366:23   + 0x181
    [build/libs/libGaudiKernel.so D[0x34ba61]]
```

Heap corruption diagnosis

Heap corruption can be one of the hardest problems to diagnose: even in the single-threaded case, any visible crash may not happen until long after the actual corruption.

ATLAS offline uses tcmalloc for memory allocation by default.

Had some success in modifying tcmalloc to catch errors like double-deletions and overwrites.



In some cases, this promptly located errors that we had been chasing by other means for weeks.

In another instance, we had a rare failure where a free memory block was being overwritten with a distinctive 64-bit value, as determined from the register dump:

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

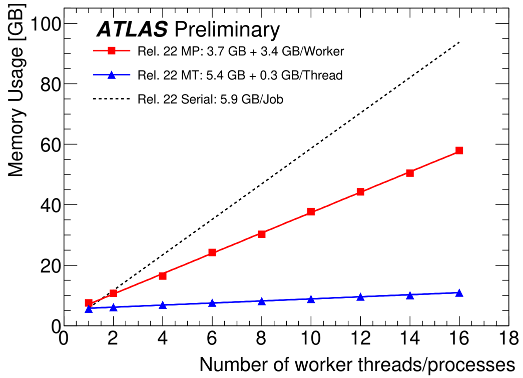
Wrote a custom Valgrind checker to log all writes of this particular value:

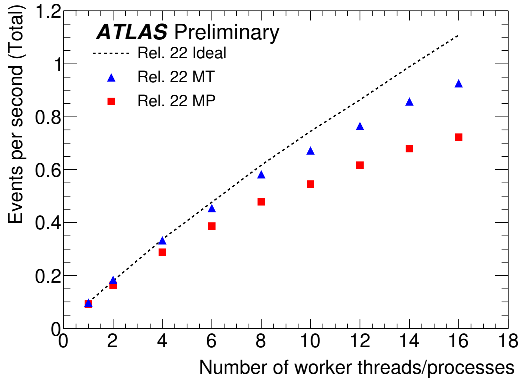
```
static VG_REGPARM(2) void trace_store(Addr addr, SizeT size){
  if (size == 8) {
    unsigned long long val = *(unsigned long long*)addr;
    if (val == 0x3fc0be57ef09fe55) {
       VG_(printf)(" wrote %08lx %08llx\n", addr, val);
       VG_(get_and_pp_StackTrace) (VG_(get_running_tid)(),20);
```

This immediately located the error (data race in writing a `std::vector` holding RNG seed information).

Results [ATL-SOFT-PUB-2021-002]

- Memory and throughput for typical reconstruction on a 16-core Xeon E5-2630 (no SMT).
- Includes all needed initialization, finalization, and output file merging.
- 250 events/thread, $\langle\mu\rangle \approx 50$.
- Results from mid-2021. Work has continued since then to improve scaling for many threads.





Summary

- In order to reduce memory required per core, ATLAS has migrated the 5M-line offline code base to run multithreaded.
- Roughly a five-year project.
- Resulting performance is excellent! Memory requirements scale at about 0.3 GB / thread. CPU scaling is very good up to at least 8 threads.
- Performance improvements continue.
- In production for Run 3 and Run 2 reprocessing.
- A solid base for work leading to supporting heterogenous systems for Run 4.

If you are starting a new C++ code base, try to make it thread-friendly from the start!

Take const seriously

Avoid `const_cast`.
Methods which process data show take `const` inputs, and usually be `const` themselves.

Avoid gratuitous use of `static` and global state.

Singleton objects should not contain event data.