Recap on Geant4 Multithreading

Geant4 Collaboration Workshop

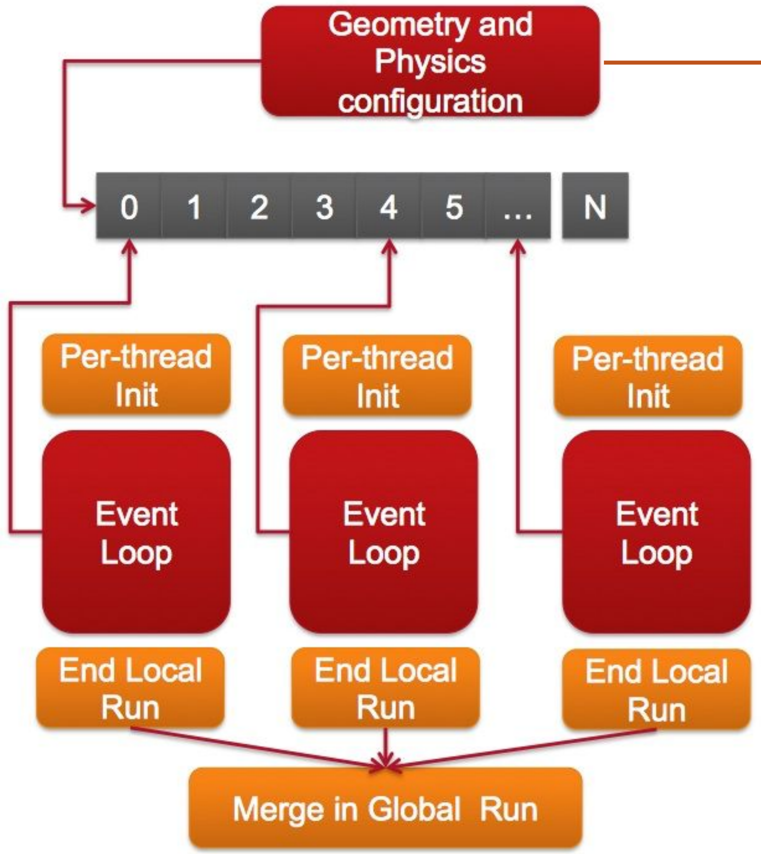
Ben Morgan (The University of Warwick)

WARWICK
THE UNIVERSITY OF WARWICK

Wait, isn't Geant4 multithreading done?

- Several topics in development and R&D are touching the multithreading system, so a recap of the technology and issues is worthwhile
  - *Possible 4th Technical Paper would cover Tasking, **ideally also lead to Tech Note***
  - *Remember that the Geant4 MT system and design has stood test for a **decade** now!*
- Subevent parallelism
  - *Sequential events, split into subevents (groups of tracks) per thread/task*
  - *See next presentation from Makoto*
- Initialization in parallel
  - *Geometry, physics tables*
  - *Working session tomorrow afternoon*
- **Only a very high level overview of core aspects and debugging tools, see the Toolkit Developer's Guide for a more in depth guide on thread local memory management types in particular.**

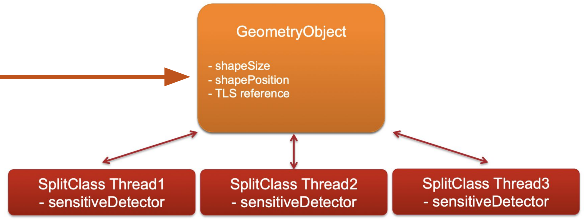
High Level Structure



Per-event seeds pre-prepared in a “queue”

Threads compete for next event to be processes (new in ref-08)

Command line scoring and G4tools automatically merge results from threads



- Memory-consuming common data (geometry, physics) uses “Split Class” mechanism for thread-safety
  - *Read-only part shared globally*
  - *Read-write part in thread-local storage*
- ***In pseudocode:***

```
struct G4SplitClass
{
  G4GlobalShared a_;
  G4Split<G4ThreadPrivate> b_;
};
```

Aside: G4Allocator and Thread-Local Storage

- `G4Allocator` provides fast memory pool allocation, typically used to implement new/delete operators for very frequently constructed classes
  - *E.g. G4Track, hits collections*
- Being shared between all instances of a given type, they are thread-local:

```
G4Allocator<G4Track>*& aTrackAllocator() {
  G4ThreadLocalStatic G4Allocator<G4Track>* _instance = nullptr;
  return _instance;
}

inline void* G4Track::operator new(std::size_t) {
  if(aTrackAllocator() == nullptr) aTrackAllocator() = new G4Allocator<G4Track>;
  return (void*) aTrackAllocator()->MallocSingle();
}
```

- **Thus instances allocated this way on a thread A cannot be deallocated on another thread B**

Aside-to-Aside: Note odd static construction! Possibly a no longer needed optimization.

The Basic MT Initialization/Event Loop sequence

- ***Essentially*** identical for **Classic** (`std::thread`) and **Tasking** (PTL library)
- Differences down to Classic manually managing the thread creation/destruction, whilst Tasking defers this to a *thread pool* in PTL
- **Initialization** is done in two steps:
  - *Construct geometry, physics data on main thread*
  - *Start worker threads, copying/setting up thread-local data to/on them*

Classic Mode:

- `G4MTRunManager` creates 1-N `G4Threads`
- Each thread executes `G4MTRunManagerKernel::StartThread(...)`
  - Sets up data, then waits for work requests in `G4WorkerRunManager::DoWork()`

Tasking Mode:

- `G4TaskRunManager` creates a `PTL::TaskManager`
- `G4TaskRunManagerKernel::InitializeWorker()` executed on each thread in pool
  - Sets up data and finishes (no waiting)
- **No tasks: have to guarantee run on all threads**

- **Same end result: Local run manager and data setup on each thread**

MT Event Loop 1: std::thread

- BeamOn: main run manager requests worker threads start a new run
  - *Remember that threads waiting on requests in `G4WorkerRunManager::DoWork()`*
  - *Managed by `G4MTBarrier`, ultimately `std::condition_variable` (s)*
- Threads each start their own event loops
  - *Number of events each thread will process **not determined a priori***
  - *Loop calls `SetUpAnEvent/SetUpNEvents` of main thread's run manager, which returns false if no more events are to be processed, thread then terminated its loop*
  - *Is a syncronization point for event ids and corresponding random number seeds*
- On event loop termination for a thread
  - *It notifies main thread run manager that it's done*
  - *Main thread blocks until all worker threads finished*

MT Event Loop 2: Tasking

- BeamOn: ultimately call to `G4TaskRunManager::CreateAndStartWorkers()`
  - *At this point, threads in `PTL::ThreadPool` just waiting for tasks*
  - *Task creation/management handled by `PTL::TaskManager/Group`*
- Number of Tasks nominally √`NEvent` to evenly distribute work(*)
- **Task** == wrapped call of `G4TaskRunManagerKernel::ExecuteWorkerTask()`, submitted to the `PTL::TaskManager` for execution on some thread in the pool
  - *Just confirms thread-local run manager exists, calling the `G4WorkerTaskRunManager::DoWork()` member function*
  - *Fundamentally same operations as `G4WorkerRunManager::DoWork()` in Classic*
- (*) …but only the first `NThread` Tasks usually process events, rest are "empty"
  - *Like Classic MT, Tasks query main thread run manager to determine if there are still events to process*
- `G4TaskRunManager::CreateAndStartWorkers()` submits Tasks to `PTL::TaskManager`, and then calls `wait()` to block until completion
  - *Underlying synchronization uses `std::promise/future`, Tasks return `void`*

Contrasting the two MT Systems

- Memory management essentially identical in terms of having per thread run managers and split data
  - *Classic mode theoretically has better guarantees of lifetime of these as it owns threads*
  - *Threads **could** leave Tasking's `PTL::ThreadPool`, depending on how this is managed (e.g. by experimental frameworks)*
- Event loops structurally the same, key difference in synchronization
  - *Classic: G4MTBarrier and std::condition_variable*
  - *Tasking: std::promise/std::future, though largely hidden by PTL interface*
- Nominally Tasking workflow cleaner/more obvious, but still have worker-main thread communication due to Event ID/Seeds distribution
  - *Tasks not used in worker thread initialization phase due to requirement that this is executed on all threads in the underlying pool*
  - *However, mechanism for running these is identical in concept to Tasks (pass a callable "thing" to something that will run it at a later point in time)*

PTL/Tasking Examples for Geant4 Developers

- PTL is a very simple library to use, the only gotcha usually to do with copy/move of objects (see https://github.com/jrmadsen/PTL/issues/49)
    - *… but the same as raw std::thread, so consistent with its behaviour.*
- **Kick started by Issue 22 on initialization in parallel, prepared some basic examples of PTL use:**
    - *Branch and README on GitHub*
    - *Further info in comments on Issue 22*
- Should cover most Geant4 use cases except for sending data to a thread-shared object (locking), though this is trivial to try out yourself!
    - *ptl_vector_subtask.cc additional shows ability for Tasks to create Tasks themselves*
    - *Specialized use case, possibly less relevant in event loop if pool takes all threads, but capability is there.*

MT Debugging: Using Thread Sanitizer

- Two or more threads accessing same memory with at least one access being a write is a *data race*
  - *Can be tricky to trigger/reproduce due to relative timing/sequencing of threads*
  - *Thankfully, GCC and Clang provide a tool, ThreadSanitizer , which instruments code to detect these in an application run*
- Geant4 and example/integration tests can be built with this enabled via:

```
$ cmake \
  -DGEANT4_BUILD_SANITIZER=thread     \
  -DGEANT4_USE_PTL_LOCKS=ON           \
  -DCMAKE_BUILD_TYPE=RelWithDebInfo   \
  -DGEANT4_ENABLE_TESTING=ON          \
  … any other arguments …
```

Avoids warnings from PTL internals (see MR 1744 for background) ← `-DGEANT4_USE_PTL_LOCKS=ON`

Ensure debugging into attached, so sanitizer will report code line numbers ← `-DCMAKE_BUILD_TYPE=RelWithDebInfo`

MT Debugging: ThreadSanitizer-enabled applications

- Examples/Tests in build of Geant4 also have ThreadSanitizer enabled, but to use it in external applications linking to Geant4, appropriate compile/link flags are needed.
- If you're using CMake, then these are in the `GEANT4_CXX_FLAGS` CMake variable:

```
find_package(Geant4 …)
string(APPEND CMAKE_CXX_FLAGS ” ${GEANT4_CXX_FLAGS}”)
…
```

- Otherwise the relevant flags to compile/link with are:
  - *`-fno-omit-frame-pointer -fsanitize=thread`*

MT Debugging: Checking for data races

- Simply run the application under test with any arguments needed, for example
  - *`ctest -VV -R example-bas-b1`*
  - *`./exampleB1 exampleB1.in`*
- Note that the instrumentation does introduce a runtime penalty
  - *Documentation states "...memory usage may increase by 5-10x and execution time by 2-20x."*
- Runtime flags may be passed in the `TSAN_OPTIONS` environment variable to adjust reporting and behaviour
  - *See the relevant page of the ThreadSanitizer documentation*

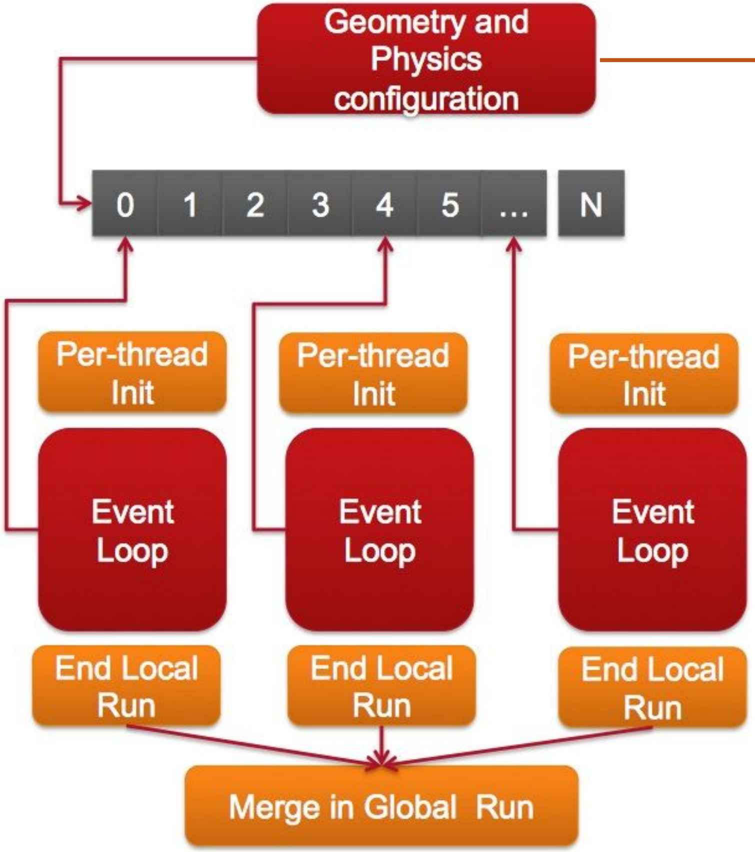
MT Debugging: Example ThreadSanitizer report

```
[macbook]$ ./exampleB1 exampleB1.in 1>/dev/null
exampleB1(96886,0x202814f40) malloc: nano zone abandoned due to inability to reserve vm space.
==================
WARNING: ThreadSanitizer: data race (pid=96886)
  Read of size 8 at 0x00010d92d528 by thread T2:
    #0 G4Trd::GetCubicVolume() G4Trd.cc:208 (libG4geometry.dylib:arm64+0x19f24c)
    #1 G4LogicalVolume::GetMass(bool, bool, G4Material*) G4LogicalVolume.cc:595 (libG4geometry.dylib:a
    #2 B1::RunAction::EndOfRunAction(G4Run const*) RunAction.cc:105 (exampleB1:arm64+0x100012754)

    #12 void* std::__1::__thread_proxy[abi:ne180100]<std::__1::tuple<std::__1::unique_ptr<std::__1::__thread_struct, std::__1::default_delete<std::__1::__thread_struct>>, void (*)(PTL::ThreadPool*, std::__1::vector<std::__1::shared_ptr<PTL::ThreadData>, std::__1::allocator<std::__1::shared_ptr<PTL::ThreadData>>>*, long), PTL::ThreadPool*, std::__1::vector<std::__1::shared_ptr<PTL::ThreadData>, std::__1::allocator<std::__1::shared_ptr<PTL::ThreadData>>>*, unsigned long>>(void*) thread.h:208 (libG4ptl.3.0.0.dylib:arm64+0xf774)

  Previous write of size 8 at 0x00010d92d528 by thread T4:
    #0 G4Trd::GetCubicVolume() G4Trd.cc:210 (libG4geometry.dylib:arm64+0x19f2c4)
    #1 G4LogicalVolume::GetMass(bool, bool, G4Material*) G4LogicalVolume.cc:595 (libG4geometry.dylib:arm64+0xd41dc)
    #2 B1::RunAction::EndOfRunAction(G4Run const*) RunAction.cc:105 (exampleB1:arm64+0x100012754)

    #12 G4UIbatch::ExecCommand(G4String const&) G4UIbatch.cc:181 (libG4intercoms.dylib:arm64+0xf0d8)
    #13 G4UIbatch::SessionStart() G4UIbatch.cc:223 (libG4intercoms.dylib:arm64+0xf51c)
    #14 G4UImanager::ExecuteMacroFile(char const*) G4UImanager.cc:286 (libG4intercoms.dylib:arm64+0x36074)
    #15 G4UIcontrolMessenger::SetNewValue(G4UIcommand*, G4String) G4UIcontrolMessenger.cc:398 (libG4int
    #16 G4UIcommand::DoIt(G4String const&) G4UIcommand.cc:223 (libG4intercoms.dylib:arm64+0x187e4)
    #17 G4UImanager::ApplyCommand(char const*) G4UImanager.cc:531 (libG4intercoms.dylib:arm64+0x3a314)
    #18 G4UImanager::ApplyCommand(G4String const&) G4UImanager.cc:442 (libG4intercoms.dylib:arm64+0x39
    #19 main exampleB1.cc:96 (exampleB1:arm64+0x10000d240)

SUMMARY: ThreadSanitizer: data race G4Trd.cc:208 in G4Trd::GetCubicVolume()
==================
ThreadSanitizer: reported 1 warnings
zsh: abort      ./exampleB1 exampleB1.in > /dev/null
[macbook]$
```

Locations of race read/write

Detailed thread creation/tracing

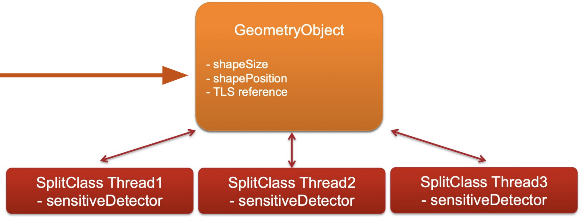
Questions, comments…



Per-event seeds pre-prepared in a “queue”

Threads compete for next event to be processes (new in ref-08)

Command line scoring and G4tools automatically merge results from threads




```
struct G4SplitClass
{
  G4GlobalShared a_;
  G4Split<G4ThreadPrivate> b_;
};
```