C++11 atomics for G4atomic

Jonathan R. Madsen
Texas A&M University

What is an atomic?

- We may need to consider a different alias given the nature of our field... G4tspod (thread-safe plain-old data)
  - An atomic is a special case of operating on plain-old data (POD) types in a thread-safe way
  - Depending on the hardware and compiler, in an atomic operation, the POD will be updated in a lock-free operation, otherwise an atomic operation reduces to wrapping the lock and unlock of a mutex around the operation
- The naming originates from the original Greek meaning of the word “atom” meaning “indivisible”
  - The idea is that the operation on the data type is indivisible – a data race isn't possible because read, operate, and write aren't separate processes. Any change by one thread is seen “instantaneously” by all other threads
- May be defined at hardware level, single machine instruction, etc.

Benefits of atomics

- With respect to performance, an atomic will perform, at the worst, the same as a mutex lock/unlock
  - Otherwise, the performance gain is dependent on the compiler/hardware but can be around multiple times faster than a mutex lock/unlock
- Depending on the implementation, leads to much cleaner code → essentially, the operation statement can reduce to looking exactly like an equivalent operation on a POD type

Benefits of atomics

- Easy implementation of thread-safety for non-advanced users
  - When data is accumulated per-event and passed at end of event to the run, the overhead of synchronization (either via atomic operations or mutexes) is *generally* very minimal and *generally* not a concern for non-HPC users
- Easily implement thread-safe "counters"
  - E.g. recording the number of events that have been completed via a counter in EndOfEventAction, not by looking at the EventID (since EventID = 5000 does not mean events have been finished)
  - May seem trivial but it is relevant for checkpoint files and intermediate outputs

C++11 atomic implementation

- Neither an assignment operator nor copy-constructor
  - Atomicity cannot be guaranteed when the atomic is not lock-free (i.e. mutex cannot be copied)
  - This makes atomics incompatible with most STL containers (vector, deque, list, stack, etc... any container that copies values when added to)
- Overloads +=, ++, -=, -- operators for integral types
  - Does not natively overload operators for floating-point types
- Two main operations for anything else: fetch-and-store and compare-and-swap

Fetch-and-store

- Primary operation for assignment
  - store(…)
- Essentially, the operation is exactly like it sounds: fetch the address and store the value provided in the function parameter at that address
  - However, unlike regular POD types, the address cannot be loaded into another core's cache after the fetch and before the store

Compare-and-swap

- The Swiss-army knife of the atomic implementation – the member function takes a minimum of 2 parameters: the expected value and the desired value
  - If the value of the atomic is the expected value, the value of the atomic is swapped out with the desired value
  - compare_exchange_strong(...) and compare_exchange_weak(...)
    - Return boolean for success
    - Weak variant can give better performance in highly-contested data

Memory Ordering

- Memory ordering specifications (parameters for FS and CAS function calls) are used to ensure updates happen a certain way
  - default memory order is sequentially consistent (seq_cst)
- 6 types:
  - memory_order_seq_cst (read-write-modify operation)
  - memory_order_acq_rel (read-write-modify operation)
  - memory_order_release (load operation)
  - memory_order_acquire (load operation)
  - memory_order_consume (load operation)
  - memory_order_relaxed (no ordering constraints)
    - Only ensure atomicity
  - See here for detailed description

Additional atomic functions

- Getting value of atomic: load()
  - e.g. atomic<double> → double

```
double example()
{
    atomic<double> a;
    a.fetch_and_store(4);
    return a.load();
}
```

- Checking if atomic is lock-free: is_lock_free()

Proposal for G4atomic

- Create G4atomic template class
  - Define assignment operator and copy-constructor (issue compiler warning if atomic is not lock-free?)
    - Former would eliminate need for fetch-and-store outside of class and latter would make the atomic class compatible with STL containers
  - Define arithmetic operations for floating-point POD types using compare-and-swap under the hood
    - Arithmetic operation statements on atomic would reduce exactly to equivalent arithmetic operation statements for POD

Proposal for G4atomic

- Recommend for data accumulation in non-highly contested situations
  - use G4Parameter in those cases, e.g. thread-global value in G4SteppingAction
- Recommend for beginners or intermediate users with less CS experience/background who want to utilize multithreading but are less concerned with high-performance -- optimization should be only considered after development is working for these users
- Recommend using pure atomics when memory ordering is crucial or creating additional G4atomic-type class with more control over memory ordering

G4atomic vs. G4Parameter

- G4atomic
  - easiest to use
  - slower
  - more defined operators
  - POD-only
  - no thread-local values
- G4Parameter
  - extra requirements
    - G4ParameterManager,
    - Register value with manager
    - Possible call to merge
  - POD-only?
  - thread-local values (faster)
  - User-defined operations beyond + and *

Implementing the proposal

- I've already done it
  - examples/extended/parallel/ThreadsafeContainers
    - This is a fully-compatible version for C++98, C++0x, and C++11 – much more complicated than what we need with the transition to C++11 (if C++11 is not available, it looks for Boost atomics or TBB atomics, and if neither are available, implements a mutex system)
  - examples/basic/B1 (maybe B1a?)
    - Discussing with Ivana who also proposed G4Parameters