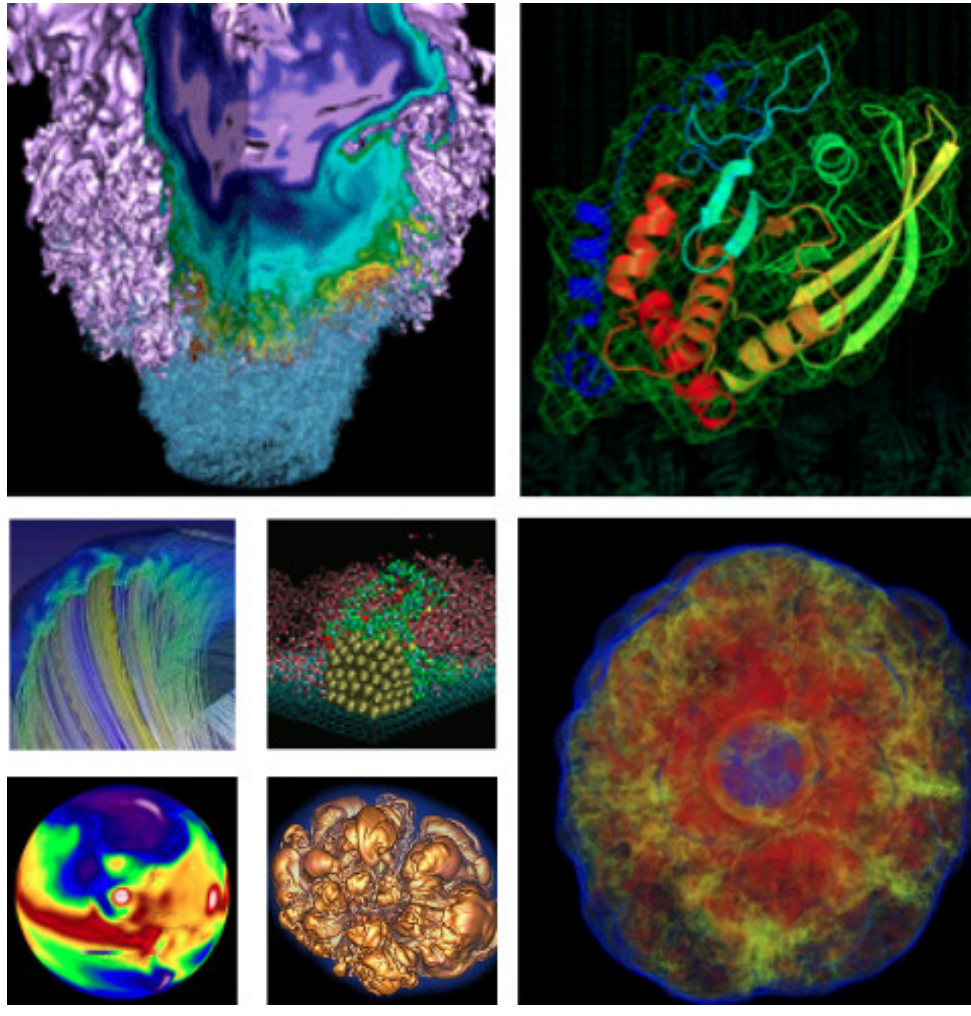
Geant4 Kernel Updates

C++11 Threading, Scoring, & Statistics



National Energy Research Scientific Computing Center

U.S. DEPARTMENT OF ENERGY | Office of Science

BERKELEY LAB

Jonathan R. Madsen

✉ jrmadsen@lbl.gov

National Energy Research Scientific Computing Center
Lawrence Berkeley National Laboratory

August 29, 2018

- All POSIX threading implementation updated to C++11 threading
- Windows + MT enabled
  - Windows + MT + shared libraries requires "construct-on-first-use" paradigm
- G4THitsVector
  - Similar to G4THitsMap but smaller memory footprint
- G4StatAnalysis
  - Low memory footprint class for basic statistics
    - Mean
    - Standard deviation / variance
    - Relative error
    - FOM
    - Efficiency
  - Size: 2 int, 2 doubles (constant)
  - Compatible with G4THitsMap and G4THitsVector

- Use of macros is *mostly* deprecated
  - Exception: condition variables
  - `G4Mutex` ⇒ `std::mutex`
  - `G4RecursiveMutex` ⇒ `std::recursive_mutex`
- Any "initializer" / "destructor" assignment or function is no longer needed – these are C requirements, C++ has constructors/destructors
  - `... = G4MUTEX_INITIALIZER;`
  - `G4MUTEXINIT(...);`
  - `G4MUTEXDESTROY(...);`
  - These are literally everywhere and do absolutely nothing. If you see them, remove them please.

```
#define G4MUTEX_INITIALIZER {}
#define G4MUTEXINIT(mutex) ;;
#define G4MUTEXDESTROY(mutex) ;;
```

- `G4MUTEXLOCK` and `G4MUTEXUNLOCK` retain previous functionality for backwards compatibility but are not necessary to use

- `G4AutoLock` ⇒ `std::unique_lock<std::mutex>` (documentation)
  - lock/unlock
  - `bool G4AutoLock::owns_lock()`
  - `bool G4AutoLock::try_lock()` + for/until `std::chrono` variants
  - `G4AutoLock::G4AutoLock(G4Mutex&, std::defer_lock_t); // or G4Mutex*`
  - `G4AutoLock::G4AutoLock(G4Mutex&, std::adopt_lock_t); // or G4Mutex*`
  - `G4AutoLock::G4AutoLock(G4Mutex&, std::try_to_lock_t); // or G4Mutex*`
- When `G4MULTITHREADED` is not defined, default constructor uses `std::defer_lock_t` for `std::unique_lock` constructor
  - *i.e.*, `G4AutoLock` does not lock mutex when constructed in serial mode
  - However, if passed to external code that uses MT, it converts to `std::unique_lock<std::mutex>` and will lock/unlock
- Introduced `G4RecursiveMutex` and `G4RecursiveAutoLock`
  - Use when function requires locking and calls another function that also requires locking and the latter is publicly accessible

- If you need one or more mutex(es) for a class, there is an easy way to get one now

```
// G4Threading.hh (serial and MT)

// Standard variant
template <typename _Tp>
G4Mutex& G4TypeMutex(const unsigned int& _n = 0);

// Recursive variant
template <typename _Tp>
G4RecursiveMutex& G4TypeRecursiveMutex(const unsigned int& _n = 0);

// example for output file
G4Mutex& _mtx = G4TypeMutex<std::ofstream>();
_mtx.lock();
//  ...
_mtx.unlock();

// using decltype(...)
G4Mutex& _this_mtx = G4TypeMutex<decltype(this)>();
// ...
```

```
// G4DummyClass.cc

// old declaration and usage
namespace
{
    G4Mutex dummyClassMutex0;
    G4Mutex dummyClassMutex1;
}
G4AutoLock lock0( &dummyClassMutex0 );
G4AutoLock lock1( &dummyClassMutex1 );

// new usage with G4TypeMutex
G4AutoLock lock0( G4TypeMutex<G4DummyClass>()  );
G4AutoLock lock1( G4TypeMutex<G4DummyClass>(1) );

// new usage with G4RecursiveTypeMutex
G4RecursiveAutoLock rlock0( G4TypeRecursiveMutex<G4DummyClass>()  );
G4RecursiveAutoLock rlock1( G4TypeRecursiveMutex<G4DummyClass>(1) );
```

- `G4Thread` ⇒ `std::thread` (MT)
- `G4Thread` ⇒ `G4DummyThread` (serial)

```
class G4DummyThread
{
public:
    typedef G4int                   native_handle_type;
    typedef std::thread::id         id;

    G4DummyThread();

    template <typename _Func, typename... _Args>
    G4DummyThread(_Func func, _Args&&... _args);

    native_handle_type native_handle() const;
    bool joinable() const;
    id get_id() const noexcept;
    void swap(G4DummyThread&);
    void join();
    void detach();
    static unsigned int hardware_concurrency() noexcept;
};
```

```
//============================================================================//
//                                  *** OLD ***
//============================================================================//
// header
extern G4PART_DLL G4Allocator<G4PrimaryParticle>* aPrimaryParticleAllocator;

// implementation
G4ThreadLocal G4Allocator<G4PrimaryParticle>* aPrimaryParticleAllocator = 0;

//============================================================================//
//                                  *** NEW ***
//============================================================================//
// header
extern G4PART_DLL G4Allocator<G4PrimaryParticle>*& aPrimaryParticleAllocator();

// implementation
G4Allocator<G4PrimaryParticle>*& aPrimaryParticleAllocator()
{
    G4ThreadLocalStatic G4Allocator<G4PrimaryParticle>* _instance = nullptr;
    return _instance;
}
```

- Summary: replace global thread-local objects with function returning reference to (static) thread-local object created within function

```
//============================================================================//
//                                  *** OLD ***
//============================================================================//
inline void* G4PrimaryParticle::operator new(size_t)
{
    if (!aPrimaryParticleAllocator)
    { aPrimaryParticleAllocator = new G4Allocator<G4PrimaryParticle>; }
    return (void*) aPrimaryParticleAllocator->MallocSingle();
}

//============================================================================//
//                                  *** NEW ***
//============================================================================//
G4Allocator<G4PrimaryParticle>*& aPrimaryParticleAllocator()
{   // modify to create on first use
    G4ThreadLocalStatic G4Allocator<G4PrimaryParticle>* _instance =
            new G4Allocator<G4PrimaryParticle>();
    return _instance;
}

inline void* G4PrimaryParticle::operator new(size_t)
{   // no need to check/assign pointer
    return (void*) aPrimaryParticleAllocator()->MallocSingle();
}
```

```
public:
    // typedefs that simply make allocator stuff easy
    typedef G4Task<_Ret, _Arg>                  this_type;
    typedef G4Allocator<this_type>              allocator_type;

public:
    // define the new operator
    void* operator new(size_type)
    {
        return static_cast<void*>(get_allocator()->MallocSingle());
    }
    // define the delete operator
    void operator delete(void* ptr)
    {
        get_allocator()->FreeSingle(static_cast<this_type*>(ptr));
    }

private:
    // static function to get allocator
    static allocator_type*& get_allocator()
    {
        G4ThreadLocalStatic allocator_type* _allocator = new allocator_type();
        return _allocator;
    }
```

- Map
  - Base:
    - `template <typename _Tp, typename Map_t = std::map<G4int, _Tp*>> class G4VTHitsMap`
  - Derived:
    - `template <typename T> class G4THitsMap`
    - `template <typename T> class G4THitsMultiMap`
    - `template <typename T> class G4THitsUnorderedMap`
    - `template <typename T> class G4THitsUnorderedMultiMap`
- Vector (new)
  - Base:
    - `template <typename _Tp, typename Map_t = std::vector<_Tp*>> class G4VTHitsVector`
  - Derived:
    - `template <typename T> class G4THitsVector`
    - `template <typename T> class G4THitsDeque`

- `G4VTHitsVector` is `G4VTHitsMap` with an underlying vector or deque container
- Compatible with existing `G4THitsMap` usage
  - *e.g.*, `G4THitsVector<T>::GetMap()` returns a conversion to `G4THitsMap<T>`
  - Supports using `std::vector<T>` in addition to default `std::vector<T*>`
    - `double*` requires 2x more memory than `double` ⇒ 8 bytes for pointer to double address, 8 bytes for double
    - Extensive use of template meta-programming
- Not intended to replace `G4THitsMap` in `G4VPrimitiveScorers`
  - Associative containers are better for sparse data w.r.t. indexes ⇒ sparsity is probable for one `G4Event`
  - STL map is a binary tree ⇒ significant increase in memory when dense data w.r.t. indexes
- Intended for use in `G4Run`
  - Accumulation over run increases probability of dense data w.r.t. indexing

```cpp
// existing usage
G4THitsMap<G4double>* doseHits = reRun->GetHitsMap(fSDName[i] + "/DoseDeposit");
if(doseHits && doseHits->GetMap()->size() != 0)
{
    std::map<G4int,G4double*>::iterator itr = doseHits->GetMap()->begin();
    for(; itr != doseHits->GetMap()->end(); itr++)
    {
        if(!IsMaster()) { local_total_dose += *(itr->second); }
        total_dose += *(itr->second);
        G4cout << itr->first << " " << *(itr->second) << G4endl;
    }
}
// new usage
auto* doseHits = reRun->GetHitsMap(fSDName[i] + "/DoseDeposit");
if(doseHits)
{
    for(auto itr = doseHits->begin(); itr != doseHits->end(); ++itr)
    {
        auto* obj = doseHits->GetObject(itr); // pass iterator
        if(!obj) { continue; } // vector<T*> will return nullptr
        if(!IsMaster()) { local_total_dose += *obj; }
        total_dose += *obj;
        G4cout << doseHits->GetIndex(itr) << " " << *obj << G4endl;
    }
}
```

- Using non-pointer version is done by changing the underlying container type
  - Second template parameter in the virtual template class
- The access example on previous slide remains the same, you just will get pointers to zero instead of null pointers

```
// same as G4THitsVector<double>
G4VTHitsVector<double, std::vector<double*> G4THitsVectorPtr;

// not using pointers
G4VTHitsVector<double, std::vector<double>> G4THitsVectorRef;
```

- Enabling this capability requires extensive template meta-programming
- No current plans to extend this functionality to `G4VTHitsMap` unless desired
  - Already contains significant TMP to handle *map* vs. *multimap*

- `G4ConvergenceTester` provides extensive statistical analysis
  - Large, variable memory usage per object (3 bool, 6 int, 20 doubles, 15 vectors, 1 map, 1 `G4Timer`, ... etc.)
  - "Merge" operation neither available nor straight-forward to implement
- `G4StatDouble` provides limited statistics (count, mean, RMS)
  - Small, fixed memory usage per object (1 int, 5 doubles)
  - Updating with weight is important
    - `G4THits[Map,Vector]` will always update with weight $= 1$
    - `obj += (value * weight)` $\neq$ `obj.add(value, weight)` for RMS calculation
- `G4StatAnalysis` provides subset of `G4ConvergenceTester` fields: mean, variance, relative error, efficiency, r2int, r2eff, FOM
  - Small, fixed memory usage (2 int, 2 doubles)
  - `obj += (value * weight)` $=$ `obj.add(value, weight)` for all calculations
  - NOTE: `G4StatAnalysis` FOM uses CPU time since start of run, `G4ConvergenceTester` FOM uses CPU time since creation of object

- `G4THitsMap<G4StatAnalysis>` $\Rightarrow$ individual statistics for every hits index
- Defines `operator double()` (implicit conversion) returning sum $\Rightarrow$ can be treated as a `double` when passing to functions, etc.
- Currently defined in `~/source/global/HEPNumerics` $\Rightarrow$ global CMake include folder changes to enable `G4BestUnit` + printing statistics
  - Should it be elsewhere or include this folder in essentially every sources.cmake?