

Multithreaded Explorations with Geant4

Lisa Goodenough
Argonne National Lab

art Users Meeting
June 17, 2016

The art-HPC Team and Contributors

The art-HPC Team is a small group of physicists and computer scientists working to enable the g-2 simulation with art to run on high performance computing (HPC) systems.

- at Argonne
 - Lisa Goodenough
 - Tom LeCompte

- at Fermilab
 - Jim Kowalkowski
 - Adam Lyon
 - Marc Paterno
 - James Stapleton

Special thanks to Kyle Knoepfel, Chris Green, Lynn Garren, and Jessie Melhuish (U Kentucky) for help on this project.



Why Utilize High Performance Computing Resources?

- Because we can. The resources are out there and they have been made available to us when we have asked.
- Because memory and its access and I/O are limitations of the current paradigm. Both can be alleviated with the use of HPCs.



HPC Resources - Argonne Leadership Computing Facility (ALCF)

- Mira
 - 10 PFLOPS IBM Blue Gene/Q supercomputer
 - processor: sixteen 1.6 GHz Power PC A2 cores, each with 4 hardware threads
 - 48 racks; 49,152 nodes; 786,432 cores; 16 GB RAM/node
 - users have access to a 24 PB GPFS file system and the HPSS data archive
 - compile and link procedures are performed on login nodes with a cross-compilation technique
 - gcc, xl, and bgclang compilers available
- Cetus and Vesta
 - same architecture, software environment, and file systems as Mira
 - Cetus: primary role is to be used for debugging of problems that occurred on Mira; 65,536 cores
 - Vesta: used for testing and development work; 32,768 cores

HPC Resources - National Energy Research Scientific Computing Center (NERSC)

- Cori
 - Phase 1 (in operation now): 1.92 PFLOPS Cray XC40; two 2.3 GHz 16-core Haswell processors per node; 1630 compute nodes → 52,160 cores; 128GB RAM/node
 - Phase 2 (to be installed and merged with Phase 1 mid-2016): CrayXC based on second generation of Intel Xeon Phi products; over 9300 nodes, 64 cores/node → 632,400 cores, 96 GB Ram/node
 - users have access to HPSS data archive
 - compile and link procedures are performed on login nodes with a cross-compilation technique
 - Intel, Cray and GNU compilers available
- Edison
 - Cray XC30, delivers 2.57 petaflops at peak performance
 - 5576 compute nodes; 24 cores/node → 133,824 cores; 64 GB RAM/node
 - Intel Hyper-Threading enabled so you can run with 48 logical cores/node

The Plan of Attack

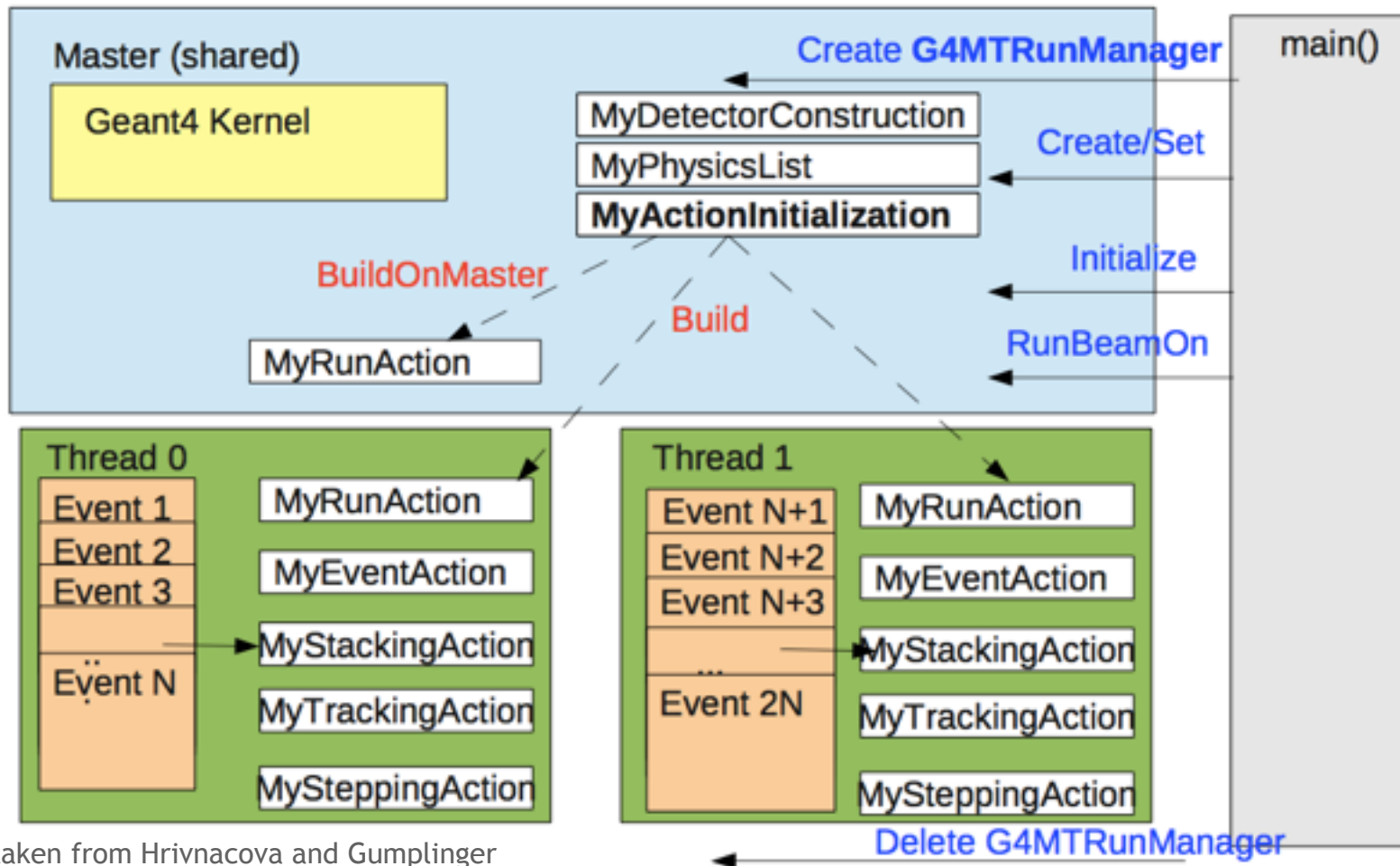
Phase 1, in which multithreading is completely managed within G4MT:

- Upgrade the g-2 software to Geant4 v10.x - this does not require the use of multithreading
- Get g-2 code built on NERSC and ALCF machines
- Upgrade g-2 code to incorporate Geant4's multithreading capabilities
- Make changes to art to incorporate thread safety into events
- Run on test machine (e.g. at Argonne) with 1 thread, 2 threads, 10 threads
- Get g-2MT code built and running on NERSC and ALCF machines
- Test the scaling, figure out the load balancing
- Simulate some large number of events. 10^{12} or more muons needed to reach desired statistical sensitivity. We would like to contribute significantly to this need.



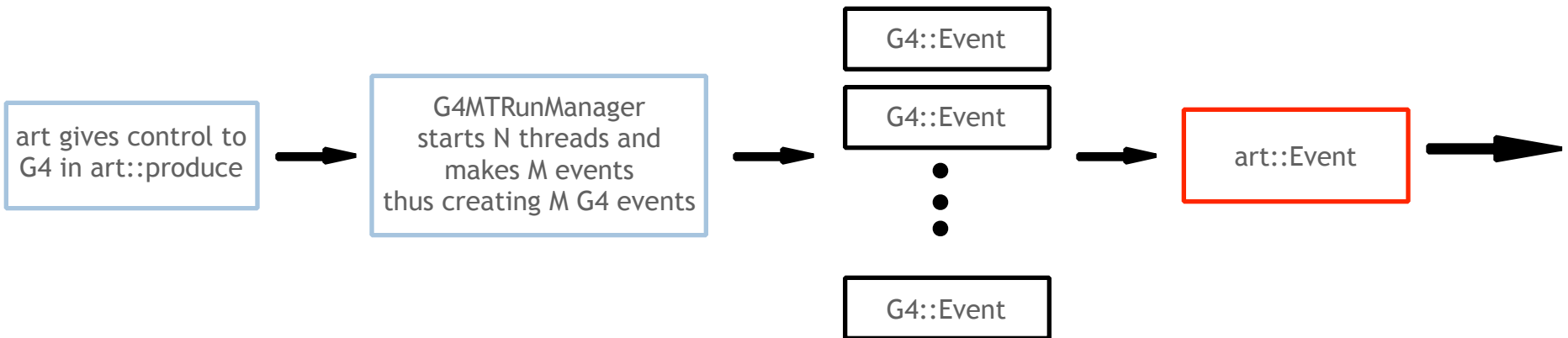
Geant4MT - Event Level Parallelism

Geant4 10.x in MT mode offers 'event level parallelism'. Each worker thread is tasked to simulate one or more events independently, while the master thread controls the overall initialization and also distributing events and merging results to/from worker threads.



The g-2 MT Event Loop

- In MT mode, a single art event consists of some number N of Geant4 events, which is set by the G4MTRunManager in main().
- The multithreading is controlled by Geant4. Once the N events are complete, data accumulation is done in art. The art event is currently not thread safe.



Geant4MT - Shared versus Thread Local Objects

- In MT mode, objects that are **invariant** during the event loop are **shared among threads**. This helps to reduce the memory footprint.
 - Geometry and physics tables
 - User initialization classes `G4VUserDetectorConstruction`, `G4VUserPhysicsList`, and newly introduced `G4VUserActionInitialization`
- Objects that are **transient** during the event loop are **thread-local**. Thread-local objects are instantiated and initialized at the end of the first execution of `G4RunManager::Initialize()`.
 - Events, tracks, steps, trajectories, hits, etc.
 - Several G4 manager classes such as `EventManager`, `TrackingManager`, `SteppingManager`, `TransportationManager`, `GeometryManager`, `FieldManager`, `Navigator`, and `SensitiveDetectorManager`
 - User action classes and sensitive detector classes



Code Modifications Made to Incorporate G4MT

- We have moved user-action instantiation from artg4Main_module.cc to the G4UserActionInitialization class.

artg4Main BEFORE:

```
art::ServiceHandle<ActionHolderService> actionHolder;
actionHolder->initialize();

// Store the run in the action holder
actionHolder->setCurrArtRun(r);

// Declare the primary generator action to Geant
runManager_->SetUserAction(new ArtG4PrimaryGeneratorAction);

// Note that these actions (and ArtG4PrimaryGeneratorAction above) are all
// generic actions that really don't do much on their own. Rather, to |
// use the power of actions, one must create action objects (derived from
// @ActionBase@) and register them with the Art @ActionHolder@ service.
// See @ActionBase@ and/or @ActionHolderService@ for more information.
runManager_ -> SetUserAction(new ArtG4SteppingAction);
runManager_ -> SetUserAction(new ArtG4StackingAction);
runManager_ -> SetUserAction(new ArtG4EventAction);
runManager_ -> SetUserAction(new ArtG4TrackingAction);
runManager_ -> SetUserAction(new ArtG4RunAction);

runManager_->Initialize();
physicsListHolder->initializePhysicsList();
```

artg4Main AFTER:

```
art::ServiceHandle<ActionHolderService> actionHolder;
actionHolder->initialize();

// Store the run in the action holder
actionHolder->setCurrArtRun(r);

// User action initialization
runManager_->SetUserInitialization(new ArtG4ActionInitialization());

runManager_->Initialize();
physicsListHolder->initializePhysicsList();
```



Code Modifications Made to Incorporate G4MT

- We have moved user-action instantiation from artg4Main_module.cc to the G4UserActionInitialization class.

New ArtG4ActionInitialization class AFTER:

```
// ArtG4ActionInitialization.cc provides implementation of Art G4's built-in action initialization.

// Author: Lisa Goodenough
// Date: December 2015

// Include header
#include "artg4/geantInit/ArtG4ActionInitialization.hh"

// Other local includes
#include "artg4/geantInit/ArtG4EventAction.hh"
#include "artg4/geantInit/ArtG4PrimaryGeneratorAction.hh"
#include "artg4/geantInit/ArtG4RunAction.hh"
#include "artg4/geantInit/ArtG4StackingAction.hh"
#include "artg4/geantInit/ArtG4SteppingAction.hh"
#include "artg4/geantInit/ArtG4TrackingAction.hh"

// used for defining only the UserRunAction for the master thread
void artg4::ArtG4ActionInitialization::BuildForMaster() const
{
    SetUserAction(new ArtG4RunAction);
}

// used for defining user action classes for worker threads as well as for the sequential mode.
void artg4::ArtG4ActionInitialization::Build() const
{
    SetUserAction(new ArtG4EventAction);
    SetUserAction(new ArtG4PrimaryGeneratorAction);
    SetUserAction(new ArtG4RunAction);
    SetUserAction(new ArtG4StackingAction);
    SetUserAction(new ArtG4SteppingAction);
    SetUserAction(new ArtG4TrackingAction);
}
```



Code Modifications Made to Incorporate G4MT

- We use G4MTRunManager exclusively in main(). For sequential mode, we use runManager_ -> SetNumberOfThreads(1).

```
// At begin job
void artg4::artg4Main::beginJob()
{
    // Set up the run manager and the number of threads
    mf::LogDebug("Main_Run_Manager") << "In begin job";
    runManager_.reset( new ArtG4MTRunManager );

    // if this is not set, then the default is 2
    runManager_->SetNumberOfThreads(1);
}
|
```

- We also determine the number of G4 event per art event.

```
// Begin the art event, making "MTEvents" G4events per art event

G4int MTEvents = 1;
runManager_ -> BeamOnDoOneMTEvent(e.id().event(),MTEvents);
```



Code Modifications Made to Incorporate G4MT

- We have split detector construction, putting sensitive detectors and fields, thread local quantities, in the new ConstructSDandField method. Shared quantities such as definitions of materials, volumes, etc. are left in the Construct method.

BEFORE:

```
G4VPhysicalVolume * artg4::ArtG4DetectorConstruction::Construct()  
{  
    return world_;  
}
```

AFTER:

```
G4VPhysicalVolume * artg4::ArtG4DetectorConstruction::Construct()  
{  
    return world_;  
}  
  
void artg4::ArtG4DetectorConstruction::findSD(G4VPhysicalVolume* physicalV){  
  
    G4LogicalVolume* logicalV = physicalV->GetLogicalVolume();  
    G4int numdaughterLVs = logicalV->GetNoDaughters();  
  
    if (numdaughterLVs == 0){  
        return;  
    }  
  
    for (G4int i=0; i < numdaughterLVs; ++i) {  
  
        G4VPhysicalVolume* daughterPV = logicalV->GetDaughter(i);  
        G4LogicalVolume* daughterLV = daughterPV->GetLogicalVolume();  
  
        G4VSensitiveDetector* SD = daughterLV->GetMasterSensitiveDetector();  
  
        if (SD != 0) {  
            SetSensitiveDetector(daughterLV,SD);  
        }  
  
        findSD(daughterPV);  
  
    }  
}  
  
void artg4::ArtG4DetectorConstruction::ConstructSDandField()  
{  
  
    findSD(world_);  
}
```



Code Modifications Made to Incorporate G4MT

- We have transformed G4Allocator to G4ThreadLocal G4Allocator for hit and trajectory classes.

BEFORE:

```
typedef G4THitsCollection<CaloHit> CaloHitsCollection;
extern G4Allocator<CaloHit> CaloHitAllocator;
} // namespace gm2ringsim

inline void* gm2ringsim::CaloHit::operator new(size_t)
{
    void *aHit;
    aHit = (void *) CaloHitAllocator.MallocSingle();
    return aHit;
}
```

AFTER:

```
typedef G4THitsCollection<CaloHit> CaloHitsCollection;
extern G4ThreadLocal G4Allocator<CaloHit>* CaloHitAllocator
} // namespace gm2ringsim

inline void* gm2ringsim::CaloHit::operator new(size_t)
{
    if(!CaloHitAllocator){
        CaloHitAllocator = new G4Allocator<CaloHit>;
        G4AutoDelete::Register( CaloHitAllocator ); //Uses
    }
    return (void *) CaloHitAllocator->MallocSingle();
}
```




Additional Modifications One May Need/Can Make

- If data is accumulated in the RunAction classes, then G4Run's 'Merge' method would need to be implemented. Our data accumulation is all done within the art event using the art framework. So, we don't need to worry about G4Run's 'Merge' method at this time.
- Custom thread behavior is available using G4UserThreadInitialization. This could be useful/interesting down the road.
 - Change the way events are assigned to threads, e.g. from the default run-robin to a queue-based model
 - Add user-specific initialization code in thread initialization/termination functions
 - Completely replace the threading mode (by default based on pthreads) to allow custom threading frameworks

The Plan of Attack

Phase 1, in which multithreading is completely managed within G4MT:

- X ■ Upgrade the g-2 software to Geant4 v10.x - this does not require the use of multithreading
- X ■ Get g-2 code built on NERSC and ALCF machines
- X ■ Upgrade g-2 code to incorporate Geant4's multithreading capabilities
-  ■ Make changes to art to incorporate thread safety into events
 - Run on test machine (e.g. at Argonne) with 1 thread, 2 threads, 10 threads
 - Get g-2MT code built and running on NERSC and ALCF machines
 - Test the scaling, figure out the load balancing
 - Simulate some large number of events. 10^{12} or more muons needed to reach desired statistical sensitivity. We would like to contribute significantly to this need.

The Future

- In Phase 1, the multithreading is completely managed within the G4 event loop.
- We would like to move into a Phase 2 in which a project called ‘multi-schedule’ art is utilized.
- In multi-schedule art, the thread scheduling is managed by Intel Thread Building Blocks (TBB). The event coordination and workflow is managed by the art Scheduler.
- In this way, multiple events could be active at the same time across each thread.