How to migrate to multi-threading with Geant4 version 10

Makoto Asai (SLAC PPA/SCA)

SLAC NATIONAL ACCELERATOR LABORATORY
U.S. DEPARTMENT OF ENERGY
Office of Science

Five steps to migrate

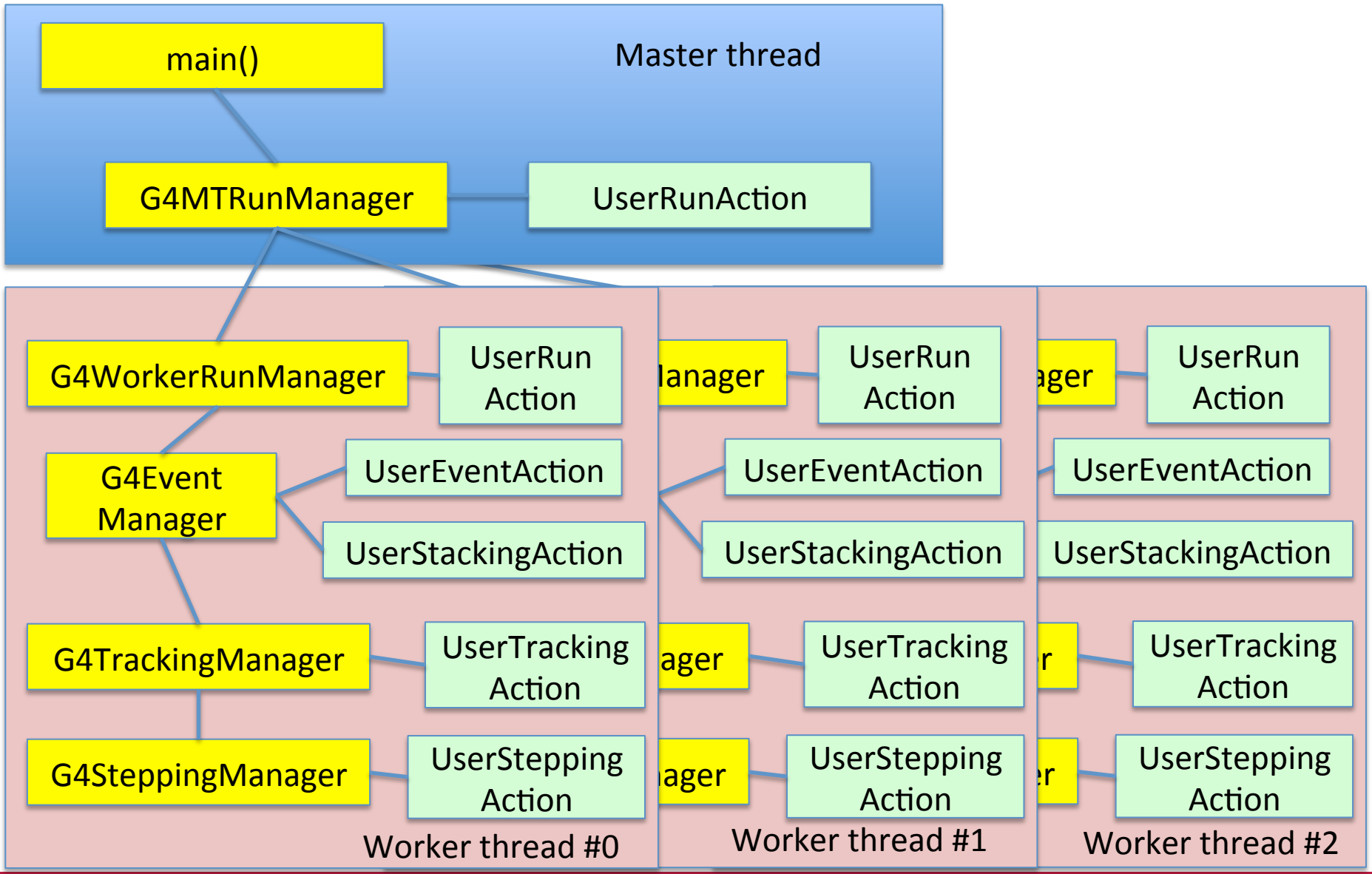
- Assuming that you have a running code built on Geant4 v9.6.p02, there are only five simple steps to migrate your code to the multi-threaded mode of Geant4 version 10.
  1. Create Action Initialization class
  2. Update main()
  3. Update Detector Construction
  4. Update/create Run and Run Action
  5. Update G4Allocator
- Steps 3~5 are optional depending on your application.
- Please note that your migrated code works for both multi-threading and sequential modes of Geant4 version 10.0.
  - The switch is
    - Instantiate G4MTRunManager for multi-threaded mode
    - Instantiate G4RunManager for sequential mode

- Migration guide
  https://twiki.cern.ch/twiki/bin/view/Geant4/QuickMigrationGuideForGeant4V10
- Most recent Geant4 tutorial course
  http://geant4.slac.stanford.edu/SLACTutorial14/Agenda.html

Step 1 – G4VUserActionInitialization

- **G4VUserActionInitialization** is a newly introduced class for the user to instantiate user action classes (both mandatory and optional).
- As described in the next slide, all the user action classes are thread-local, with the only exception of UserRunAction, which could be defined for both thread-local and global.
- **G4VUserActionInitialization** has two virtual method to be implemented, one is *Build()* and the other is *BuildForMaster()*.
  - *Build()* should be used for defining user action classes for local threads (a.k.a. workers) as well as for the sequential mode.
  - *BuildForMaster()* should be used for defining only the UserRunAction for the global run (a.k.a. master).
- All user actions must be registered through *SetUserAction()* protected method defined in the **G4VUserActionInitialization** base class.

Multi-threaded mode



G4UserActionInitialization – a new user class

- Main() for v9.6.p02

```
runManager->SetUserAction(
    new MyPrimaryGeneratorAction);
runManager-> SetUserAction(
    new MyRunAction);
runManager->SetUserAction(
    new MySteppingAction);
 . . .
```

- MyActionInitialization for v10.0

```
void MyActionInitialization::Build() const
{
 SetUserAction(
   new MyPrimaryGeneratorAction);
 SetUserAction(new MyRunAction);
 SetUserAction(new MySteppingAction);
 . . .
}

void MyActionInitialization:BuildForMaster()
const
{
 SetUserAction(new MyRunAction);
}
```

Step 2 – main()

- Instantiate G4MTRunManager instead of G4RunManager.
- Your Action Initialization has to be instantiated and set to the Run Manager.

```
// Construct the run manager
G4MTRunManager* runManager = new G4MTRunManager;
runManager->SetNumberOfThreads(8);

// Detector construction
runManager->SetUserInitialization(new MyDetectorConstruction);

// Physics list
runManager->SetUserInitialization(new FTFP_BERT);

// User action initialization
runManager->SetUserInitialization(new MyActionInitialization);
```

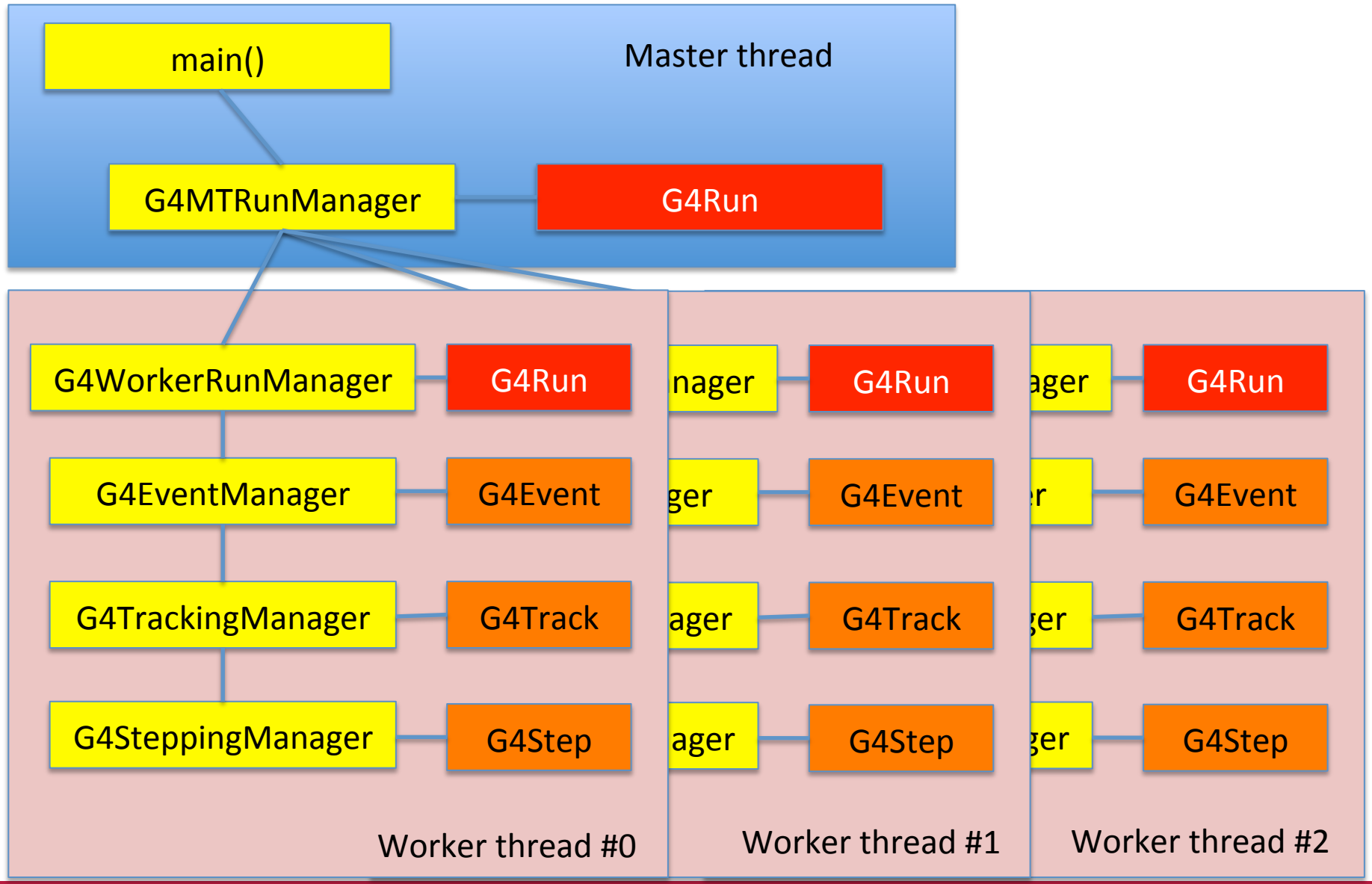
Step 3 – UserDetectorConstruction

- If you have no sensitive detector or field, skip this step.
  - You may still use command-based scorer.
- G4VUserDetectorConstruction now has a new virtual method *ConstructSDandField()*.
- Given sensitive detector class objects should be thread-local, instantiation of such thread-local classes should be implemented in this new ***ConstructSDandField()*** method, which is invoked for each thread. *Construct()* method should contain definition of materials, volumes and visualization attributes.
- To define a sensitive detector in ***ConstructSDandField()*** method, a new protected method *SetSensitiveDetector("LVName",pSD)* is available to make ease of migration, This *SetSensitiveDetector("LVName",pSD)* method does two things:
  - Register the sensitive detector pointer *pSD* to G4SDManager, and
  - Set *pSD* to the logical volume named *"LVName"*.
- If the user needs to define sensitive detector(s) to the volumes defined in a parallel world, (s)he may do so by implementing G4VUserParallelWorld::ConstructSD() method. Please note that defining field in a parallel world is not supported.

Step 4 – Create/update Run and Run Action

- If you don't need to accumulate values for a run, skip this step.
  - You may still use command-based scorer.
- MyRun
  - Create your own MyRun class derived from G4Run.
  - Add data members for physics quantities you want to accumulate.
  - Implement two virtual methods to accumulate/merge these data members.
    - RecordEvent(const G4Event*);
    - Merge(const G4Run*);
  - At the bottom of these two methods, you must invoke the corresponding base-class methods.
- MyRunAction
  - Instantiate MyRun object in your CenerateRun() method.

Multi-threaded mode



Step 5 – G4Allocator

- If you don't have your own Hit or Trajectory class that uses its own G4Allocator, skip this step.
- If the user uses G4Allocator for his/her own class, e.g. hit, trajectory or trajectory point, G4Allocator object must be thread local and thus must be instantiated within the thread. The object new-ed and allocated by the thread-local G4Allocator must be deleted within the same thread.

- In MyHit.hh

```
typedef G4THitsCollection<B2TrackerHit> B2TrackerHitsCollection;
extern G4ThreadLocal G4Allocator<B2TrackerHit>* B2TrackerHitAllocator;
inline void* B2TrackerHit::operator new(size_t)
{
  if(!B2TrackerHitAllocator)  B2TrackerHitAllocator = new G4Allocator<B2TrackerHit>;
  return (void *) B2TrackerHitAllocator->MallocSingle();
}
inline void B2TrackerHit::operator delete(void *hit)
{
  B2TrackerHitAllocator->FreeSingle((B2TrackerHit*) hit);
}
```

- In MyHit.cc

```
G4ThreadLocal G4Allocator<B2TrackerHit>* B2TrackerHitAllocator=0;
```

Additional issues – file I/O

- File I/O is always the issue for multi-threading.
- If you have an input file for primary particles, such a file reader must be unique and shared by all threads. The method for reading file must be Mutex-ed to avoid accidental coincidence.
  - You will find a couple of example code in the migration guide

    https://twiki.cern.ch/twiki/bin/view/Geant4/QuickMigrationGuideForGeant4V10
- Each thread may create its own output file. Make sure to specify different file name for each thread.
  - For example G4Threading::GetG4ThreadID() gives you the unique thread ID. Use this thread ID as a part of the file name.
- Be careful, ROOT is not thread-safe. If you need a ROOT file as an output for your histograms or n-tuples, use G4Tool.

Primary generator

```
#include "G4VUserPrimaryGeneratorAction.hh"
class G4HEPEvtInterface;
class MyHepPrimaryGenAction
 : public G4VUserPrimaryGeneratorAction
{
 public:
  MyHepPrimaryGenAction
    (G4String fileName);
  ~MyHepPrimaryGenAction();
  . . .
  virtual void GeneratePrimaries
  (G4Event* anEvent);
 private:
  static G4HEPEvtInterface* hepEvt;
};
```

```
#include "MyHepPrimaryGenAction.hh"
#include "G4HEPEvtInterface.hh"
#include "G4AutoLock.hh"
Namespace  { G4Mutex myHEPPrimGenMutex =
                                  G4MUTEX_INITIALIZER; }
G4HEPEvtInterface* MyHepPrimaryGenAction::hepEvt = 0;

MyHepPrimaryGenAction::MyHepPrimaryGenAction
        (G4String fileName) {
  G4AutoLock lock(&myHEPPrimGenMutex);
  if( !hepEvt ) hepEvt = new G4HEPEvtInterface( fileName );
}
MyHepPrimaryGenAction::~MyHepPrimaryGenAction() {
  G4AutoLock lock(&myHEPPrimGenMutex);
  if( hepEvt ) { delete hepEvt; hepEvt = 0; }
}
void MyHepPrimaryGenAction::GeneratePrimaries
    (G4Event* anEvent) {
  G4AutoLock lock(&myHEPPrimGenMutex);
  hepEvt->GeneratePrimaryVertex(anEvent);
}
```