KEK
High Energy Accelerator Research Organization

GEANT4
A SIMULATION TOOLKIT

Introduction to Geant4 Multi-Threading

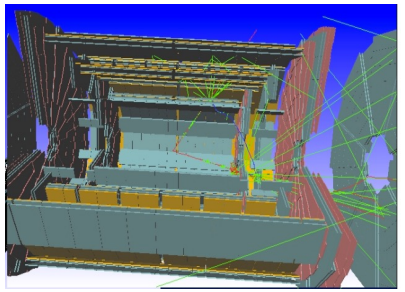
Geant4 Training Course in Medicine 2023
@ Hokkaido University, Sapporo, Japan

2023/9/28

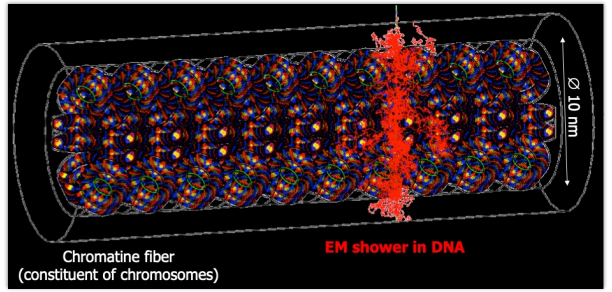
Shogo OKADA (KEK-CRC)

Need For Geant4 Parallelization

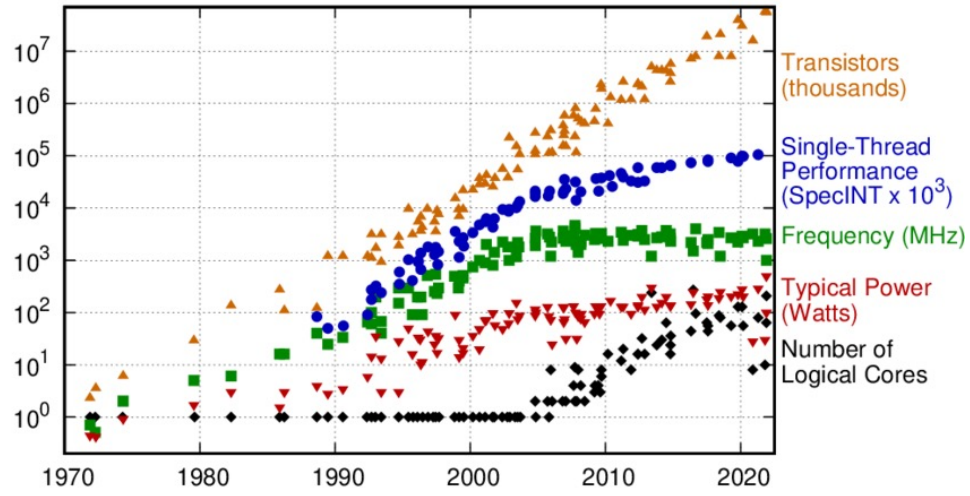
- **Larger and more complex Geant4 simulation scale, longer time users spend.**
  - Users desire to reduce simulation time and memory usage.
- **CPU Histories**
  - Single-core CPU performance increased following Moore's law until the 2000s
  - No more increases in CPU frequency due to power consumption issues
  - **Multi-core CPU era**
    - The number of transistors per chip continues to grow
      - ex) AMD EPYC 9654 CPU : 96 cores
    - Single core performance growing slowly and less memory per core
      → Applications need to introduce parallelism
- **Geant4 has supported multi-threading since ver. 10**
  - Heavy simulations using multi-threading mode can be accelerated with reducing memory usage
  - Users should be familiar with multi-thread programing



Detector Simulation for HEP (ALTAS Experiment)



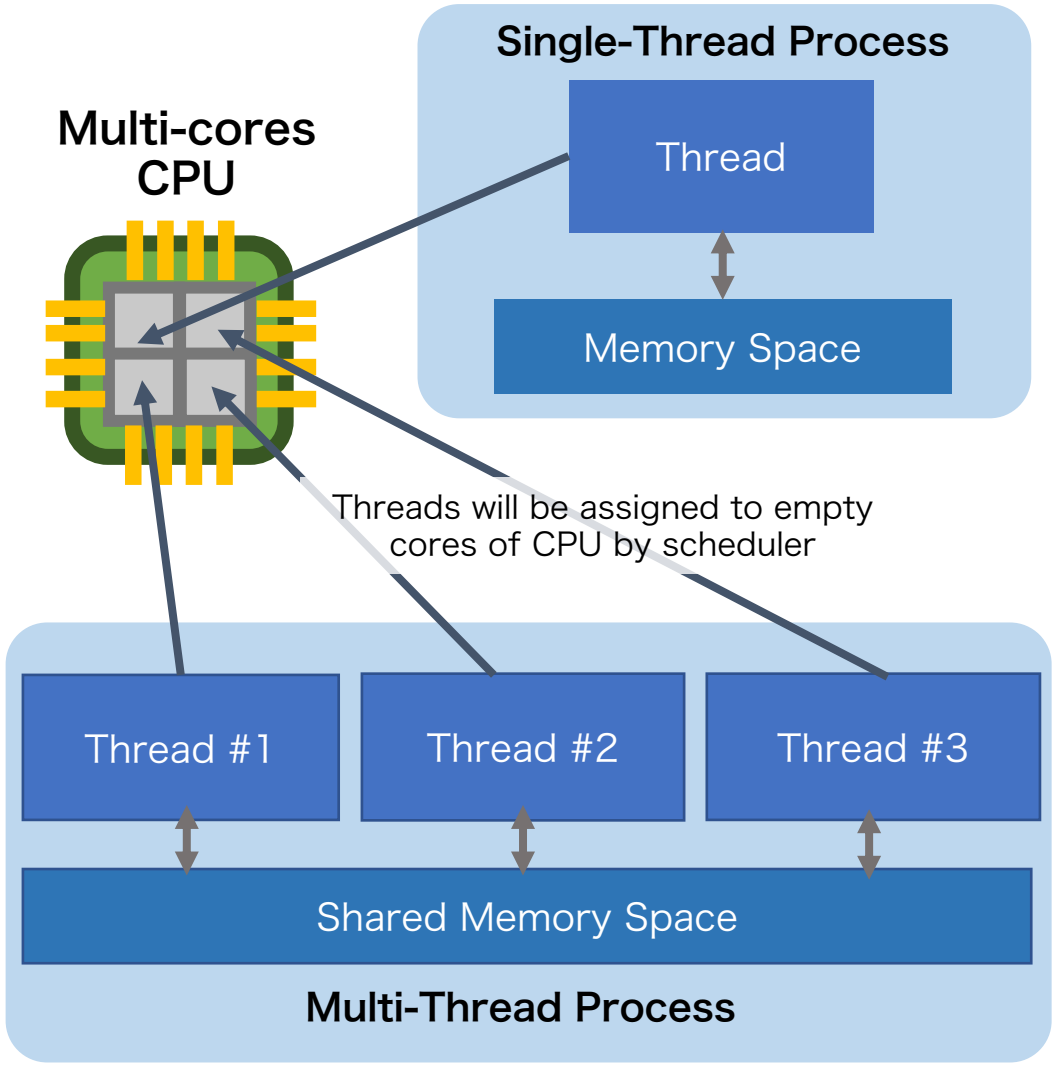
Microdosimetry Simulation by Geant4-DNA



Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2021 by K. Rupp

Process and Thread

- **Process**
  - Active programs
    - Office Software, Web Browser, Simulations, etc.
  - Separate from other processes to avoid interference
    - Take its own memory space and do not share it with other processes
- **Thread**
  - Subsets of execution within a process
    - At least a single thread will be created and assigned to a CPU core
    - A Geant4 application with sequential mode runs as a single-thread process.
  - **Multi-thread Process**
    - Running multiple threads within a process
    - Share memory space among threads
      - Efficient execution but potential data conflict
      - A thread-safe program is vital



Geant4 Parallelism

Two options of parallelism in Geant4

1. Multi-Processing
2. Multi-Threading (MT)

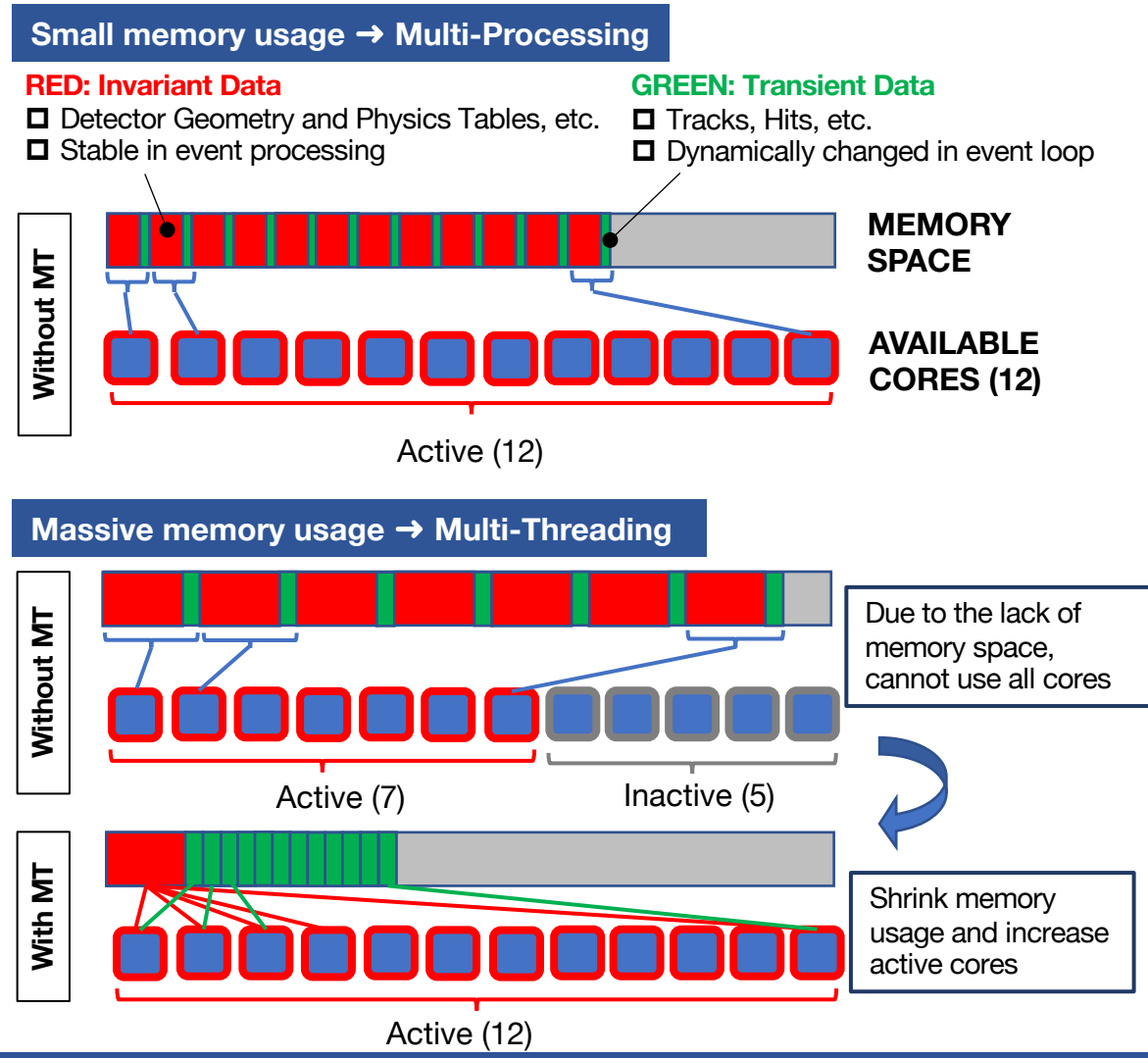
→ **Select one depending on the resources required**

Requiring small memory

- If total memory usage for your application fits the capacity, multi-processing is a good choice
- Run as multiple jobs simultaneously
- Write application codes in the traditional way (= sequential mode)
  - No need for multi-thread programming

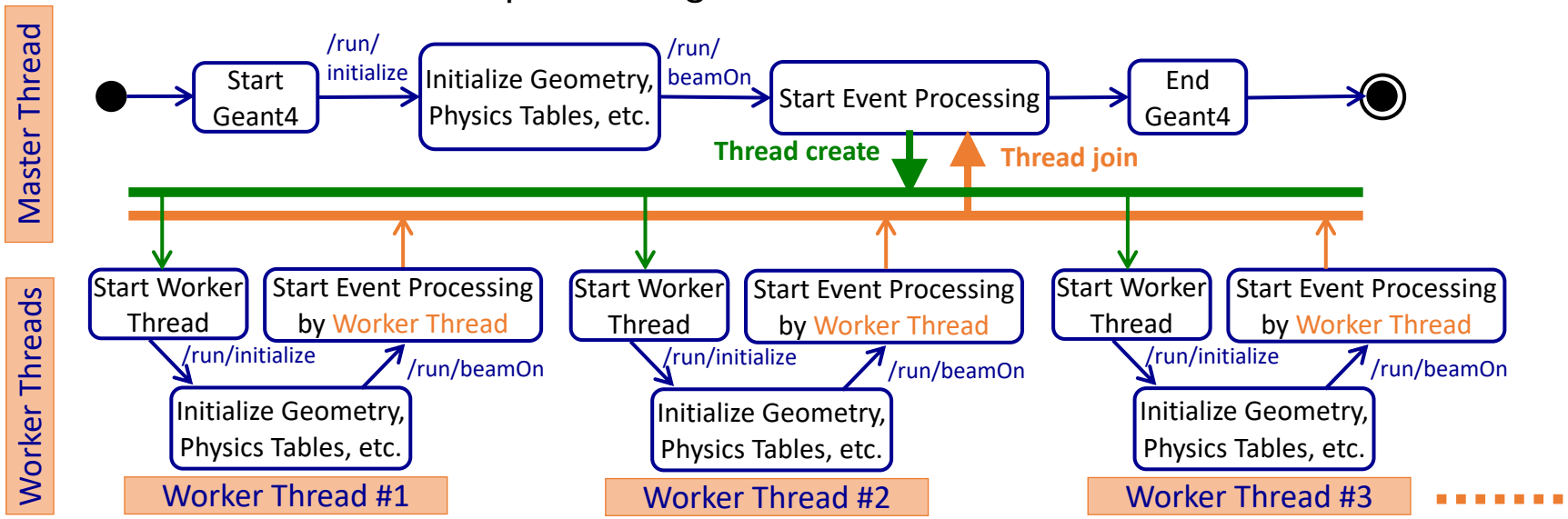
Requiring massive memory

- You should choose multi-threading
- Reduce memory usage and increase active CPU cores
  - Invariant data : Shared objects among threads
  - Transient data : Thread-local objects
- Competency in multi-thread programming is necessary

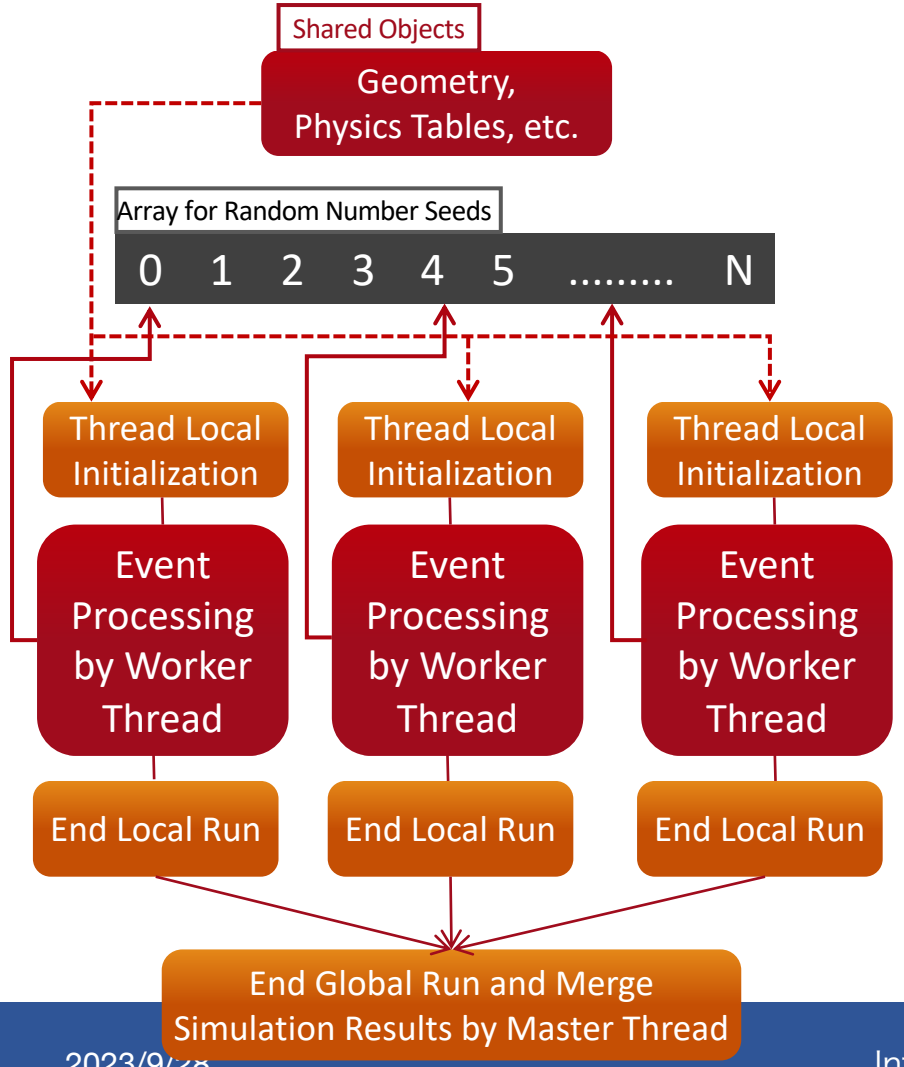


Basis for Geant4 Multi-Threading Mode (1)

- **Geant4 adopts event parallelism**
  - Geant4 processes each event independently. So, in the Geant4 multi-threading mode, multiple threads execute event processing simultaneously.
- **"Master" and "Worker" threads**
  - **Master thread** controls Geant4 simulation overall
    - Initialize shared data like geometry, physics tables, etc.
    - Create worker threads to run event processing, and then merge simulation results
  - **Worker threads** execute event processing



Basis for Geant4 Multi-Threading Mode (2)

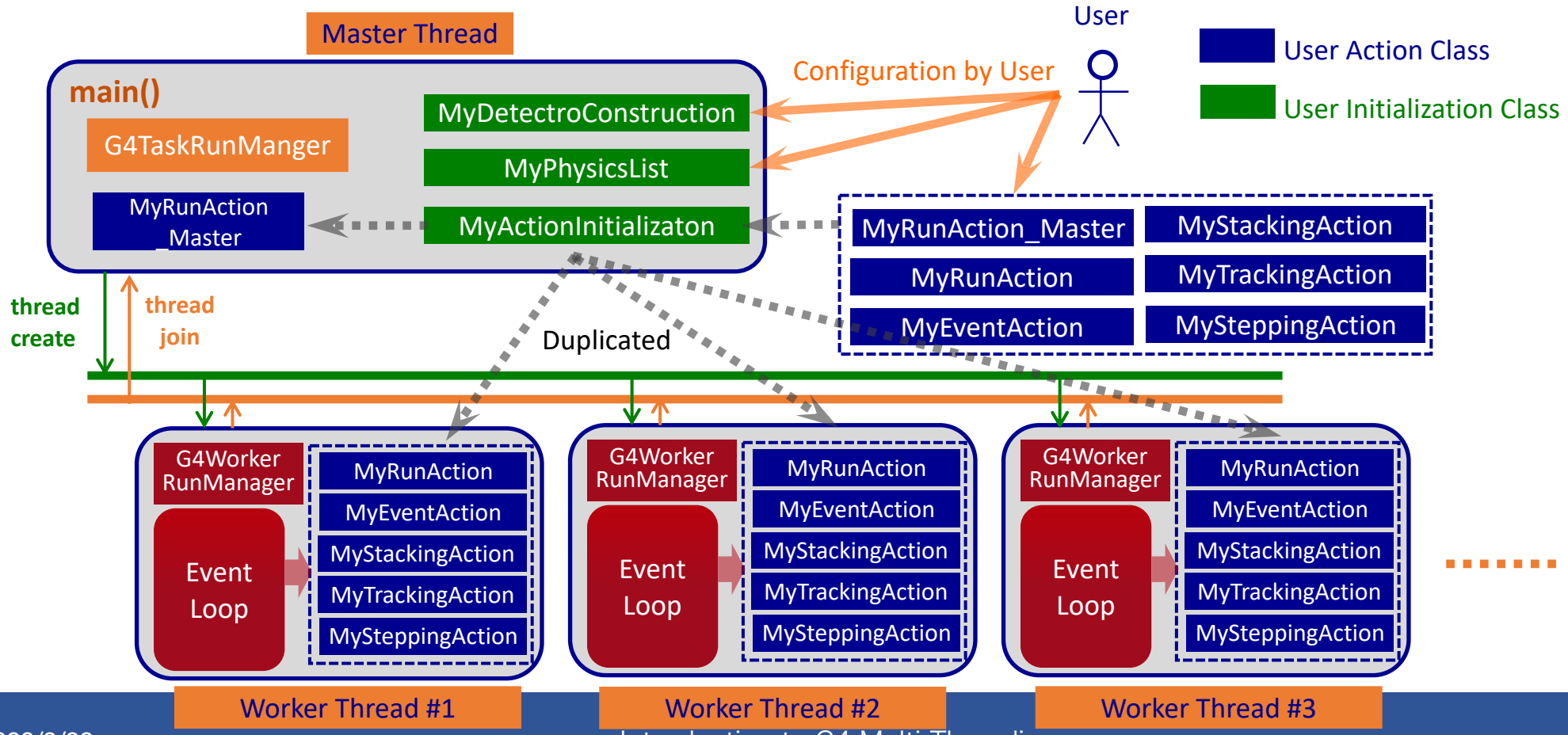


Process Flow for the G4 Multi-threading mode

- **The master thread creates a random number array that is loaded by worker threads in event processing.**
- **Also, the master thread initializes shared objects which are referred to among worker threads.**
  - Geometry, Physics Tables (= Cross-section Tables), Particle Definition, etc.
  - Stable during the event loop
  - Reducing memory consumption
- **Each worker thread will be created by the master thread. It has its own G4Run instance and starts event processing.**
  - Store simulation results locally
    - Transient objects like event, track, step, hit, etc.,
    - Sensitive Detector, User Actions (TrackAction, StepAction, …), etc.
- **After the completion of worker threads, the master thread will merge results computed by worker threads.**
  - Command line scorer and G4tools automatically merge them
  - You can use your own scorer, but you need to have a deep understanding of multi-thread programming.

User Actions in Multi-Threading Mode

- Users should understand how user actions will work in the multi-threading mode.



Migration to Multi-Threading Mode

- **"Sequential" and "Multi-threading" modes**
  - **Sequential Mode** (=Traditional mode): Run as a single-thread process
  - **Multi-threading Mode**: Using Geant4 Multi-threading libraries
- **In ver. 11, the two modes are available in a single Geant4 library set.**
  - **NOTE:** In ver. 10, a separate library is needed to be built for each mode.
- **Migration to multi-threading mode for an existing application**
  1. Replace "run manager" with one for multi-thread mode in the main program
  2. Mandatory objects like geometry, physics lists and UserActions should be configured through the UserActionInitialization class
  3. Set sensitive detector or magnetic field using `ConstructSDandField()` if necessary
  4. Merge simulation results obtained by worker thread processing
     - Geant4 Scoring Tools (Command-line scorer, G4tool)
     - User-defined scorer
       - Need multi-thread programming skills

STEP1: Main Program

- **Replace "run manager" with one for multi-threading mode in the main program**
  - **Sequential Mode** ➜ Use G4RunManager
  - **Multi-Thread Mode** ➜ Replace with G4RunManagerFactory::CreateRunManager()
    - Automatically set multi-threading mode
    - The number of worker threads can be adjusted by using `SetNumberOfThreads()`

Sequential Mode

```
01 //....................................
02 // Geant4 Application: Tutorial ...
03 //....................................
04
05 #include "Geometry.hh"
06 #include "UserActionInitialization.hh"
07
08 #include "G4RunManager.hh"
09 #include "G4UImanager.hh"
10 #include "G4VisExecutive.hh"
11 #include "G4UIExecutive.hh"
12 #include "FTFP_BERT.hh"
13
14 //....................................
15   int main( int argc, char** argv )
16 //....................................
17 {
18    // Construct a run manager
19    auto runManager = new G4RunManager;
20
21    ......
22
```

Replace

Replace

Multi-Threading Mode

```
01 //....................................
02 // Geant4 Application: Tutorial ...
03 //....................................
04
05 #include "Geometry.hh"
06 #include "UserActionInitialization.hh"
07
08 #include "G4RunManagerFactory.hh"
09 #include "G4UImanager.hh"
10 #include "G4VisExecutive.hh"
11 #include "G4UIExecutive.hh"
12 #include "FTFP_BERT.hh"
13
14 //....................................
15   int main( int argc, char** argv )
16 //....................................
17 {
18    // Construct a run manager
19    auto runManager = G4RunManagerFactory::CreateRunManager();
20    // Set the number of worker threads
21    runManager->SetNumberOfThreads(4);
22    ......
```

STEP2: User Actions

- Similarly to the sequential mode, we should register for “user actions” through the UserActionInitialization class.
- When registering for “user actions,” switch two types of “build” functions depending on thread type.
  - `Build()` : For Worker Threads
    - Register for PrimaryGeneratorAction, RunAction, EventAction, etc.
  - `BuildForMaster()` : For Master Thread
    - Register for RunAction to the master thread if necessary.

UserActionInitialization.hh

```
01 //---------------------------------------
02   class UserActionInitialization : public
03         G4VUserActionInitialization
04 //---------------------------------------
05 {
06   public:
07     UserActionInitialization();
08     virtual ~UserActionInitialization();
09
10     virtual void Build() const override;
11     virtual void BuildForMaster() const override;
12 };
```

UserActionInitialization.cc

Register for “user actions” to ***worker threads***

```
01 //-------------------------------------------------------
02 void UserActionInitialization::Build() const
03 {
04   SetUserAction(new PrimaryGeneratorAction);
05   SetUserAction(new RunAction);
06   SetUserAction(new EventAction);
07   SetUserAction(new SteppingAction);
08 }
09
10 //-------------------------------------------------------
11 void UserActionInitialization::BuildForMaster() const
12 {
13   SetUserAction(new RunActionForMaster);
14 }
```

Register for “run action” to ***the master thread***

STEP3: Detector Construction

- **Geant4 multi-thread mode handles two types of data objects:**
  - **Shared Objects**
    - Stable during event processing → ***Read-only*** objects
    - Shared among master and worker threads
  - **Thread-local Objects**
    - Each worker thread has its own local object that never interferes with the other threads.

  → Users should know which types each data is categorized.

- **Geometry data consists of two types of objects above. Users have to switch two functions to define them properly.**
  - **Shared Objects**:
    - Volumes, materials, and visualization attributes
    - G4VUserDetectorConstruction::Construct()
  - **Thread-local Objects**:
    - Sensitive detector, electromagnetic field, etc.
    - G4VUserDetectorConstruction::ConstructSDandField()

Geometry.hh

```
01 //-------------------------------------------------
02 class Geometry : public G4VUserDetectorConstruction
03 //-------------------------------------------------
04 {
05   public:
06     DetectorConstruction();
07    ~DetectorConstruction() override;
08
09     G4VPhysicalVolume* Construct() override;
10     void ConstructSDandField() override;
11 };
```

Geometry.cc

```
01 //-------------------------------------------------
02 G4VPhysicalVolume* Geometry::Constuct()
03 {
04   // setup for simulation geometry (->Shared Object)
05   ...
06
07   return world_pv;
08 }
09
10 //-------------------------------------------------
11 void Geometry::ConstructionSDandField()
12 {
13   // setup for sensitive detector (->Thread-local)
14   auto sd = new SensitiveDetector();
15   SetSensitiveDetector("LogVol_SD", sd);
16 };
```

STEP4: Scoring

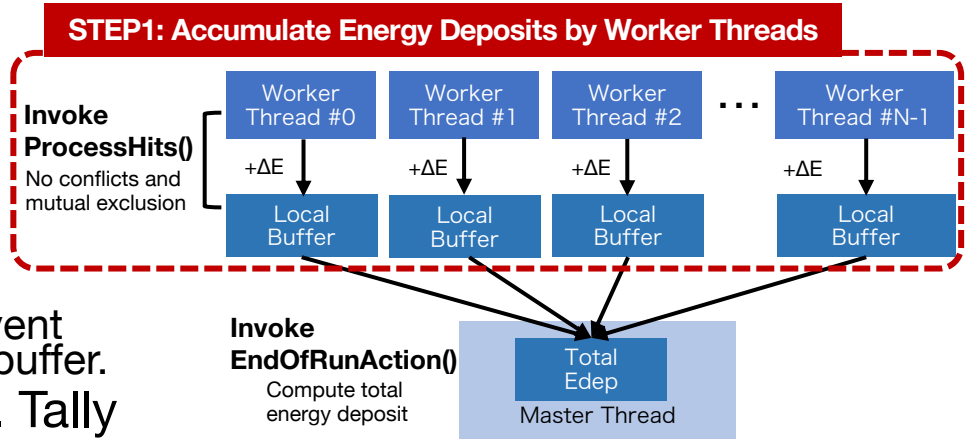
Geant4 Scoring Tools

- **Command-line Scorer:** For medicine
  - Record and marge “tally” (ex. energy deposits) automatically
- **G4Accumulable:** For HEP experiments
  - Based on Historgam/Ntuple for the ROOT Analysis Tool
  - See: https://onl.sc/Rp5dSJ5

User-defined Scorer

- No “prescriptions” for a user-defined scorer that fits all use cases
  - Need to write a program based on your purpose
  - Should avoid mutual exclusion, which may become a time penalty, as far as possible
- In the next two slides, show an example of a user-defined scorer using a Sensitive Detector class

User-Defined Scorer (1)



STEP1: Data Filling by Worker Threads

- **Sensitive Detector is a thread-local object.**
  - `ProcessHits()` will be invoked by each worker thread in event processing, which gets an energy deposit and fills it in the buffer.
- Worker threads have their own buffer to store tally (cf. Tally class below), avoiding conflicts and mutual exclusions.

SensitiveDetector.hh

```
01 //-------------------------------------------------------------
02 class SensitiveDetector : public G4VSensitiveDetector
03 //-------------------------------------------------------------
04 {
05 public:
06   SensitiveDetector(G4String);
07   ~SensitiveDetector() override;
08
09   void   Initialize(G4HCofThisEvent*) override;
10   G4bool ProcessHits(G4Step*, G4TouchableHistory*) override;
11   void   EndOfEvent(G4HCofThisEvent*) override;
12 private:
13   Tally* tally;
14 };
```

SensitiveDetector.cc

```
01 //-------------------------------------------------------------
02 G4bool SensitiveDetector::ProcessHits(
03   G4Step* step, G4TouchableHisttory*)
04 {
05   // get worker thread id
06   auto id = G4Threading::G4GetThreadId();
07   // get energy deposit and fill it
08   auto edep = step->GetTotalEnergyDeposit();
09   tally->AccumEdep(id, edep);
10 }
```

Tally.hh

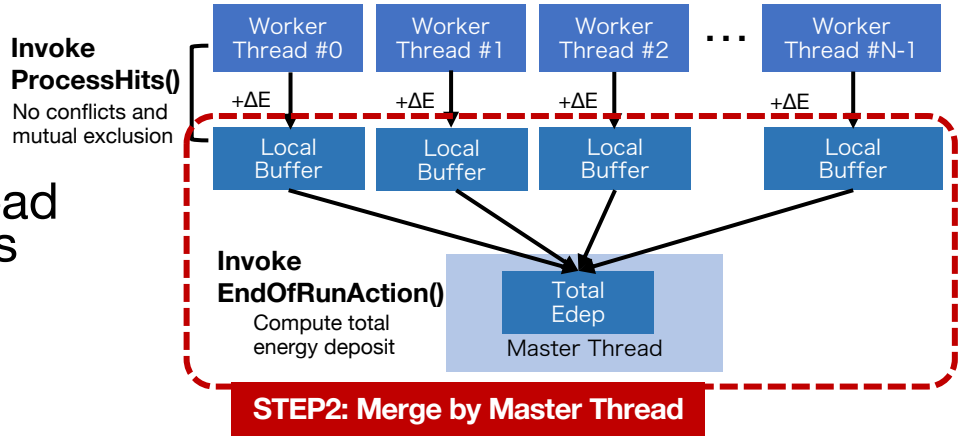
```
01 //-----------------------------------------
02 class Tally
03 //-----------------------------------------
04 {
05 public:
06   Tally(G4int num_threads)
07   {
08     edep_buffer.resize(num_threads, 0.0);
09   }
10   ~Tally() = default;
11
12   void AccumEdep(G4int id, G4double edep)
13   {
14     edep_buffer[id] += edep;
15   }
16
17   G4double GetEdep(G4int id) const
18   {
19     return edep_buffer[id];
20   }
21
22 private:
23   std::vector<G4double> edep_buffer;
24 };
```

To avoid data conflict, each worker thread has its own buffer to score tally (energy deposits).

User-Defined Scorer (2)

STEP2: Data Gathering by Master Thread

- After completion of all worker threads, the master thread invokes `EndOfRunAction()` to gather energy deposits stored in buffers of the worker threads
- Accumulate the tally from worker threads sequentially
  - No data conflicts



RunAction.cc

```
01 //---------------------------------------------------
02 void RunAction::EndOfRunAction(const G4Run*)
03 {
04   // Master thread executes data gathering
05   if (IsMaster()) {
06
07     // get the number of worker threads
08     auto num_threads =
09       G4RunManager::GetRunManager()->GetNumberOfThreads();
10
11     // calculate total energy deposit
12     double tot_edep{0.0};
13     for (auto id = 0; id < num_threads; id++) {
14       tot_edep += tally->GetEdep(id);
15     }
16
17   }
18 }
```

Tally.hh

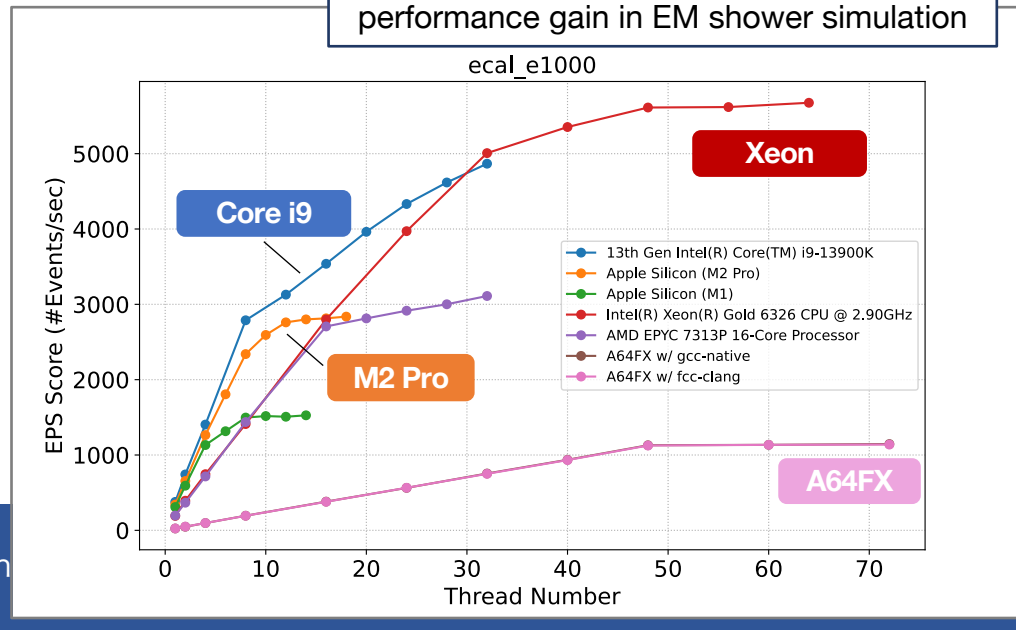
```
01 //-----------------------------------------
02 class Tally
03 //-----------------------------------------
04 {
05 public:
06   Tally(G4int num_threads)
07   {
08     edep_buffer.resize(num_threads, 0.0);
09   }
10   ~Tally() = default;
11
12   void AccumEdep(G4int id, G4double edep)
13   {
14     edep_buffer[id] += edep;
15   }
16
17   G4double GetEdep(G4int id) const
18   {
19     return edep_buffer[id];
20   }
21
22 private:
23   std::vector<G4double> edep_buffer;
24 };
```

G4Bench

Benchmark Application for Geant4

- https://github.com/koichi-murakami/g4bench2
  - Support Geant4 version 11
- Three types of benchmarks
  - ecal/hcal: electromagnetic/hadronic shower simulation
  - vgeo: water phantom simulation for voxel geometry
- Learn about *user-defined scorers* in multi-thread mode
- **RIGHT FIG:** Confirm linear performance gain as increasing thread numbers
  - EPS Score = Event Numbers per Second

Correlation between thread numbers and performance gain in EM shower simulation



References for Multi-Threading

- **Geant4 User's Guide**
  - For Toolkit Developers: https://onl.sc/kA1DWg2
    - Detailed information for event processing in parallel
  - Geant4 Examples: https://onl.sc/AALuain
    - All basic examples B1-B5 have been migrated to multi-threading

- **Geant4 IN2P3 and ED PHENIICS Tutorial 2023 @ IJCLab**
  - https://geant4-ed-project.pages.in2p3.fr/geant4-ed-web/presentations/

- **Geant4 Advanced Course 2022 @ CERN**
  - https://indico.cern.ch/event/1172490/contributions/4924351/

- **Geant4 Japanese Tutorial 2022 @ Kyushu University**
  - https://wiki.kek.jp/x/TQIZDg
  - In Japanese