Multithreading - 1

I. Hrivnacova, IJCLab Orsay

Credits: A. Dotti, M. Asai (SLAC), M. Verderi (LLR)

Geant4 IN2P3 and ED PHENIICS Tutorial,
22 – 26 May 2023, IJCLab

Outline

- Introduction
- What is a thread
- Why multithreading
- Multithreading in Geant4
- Multithreading Geant4 application

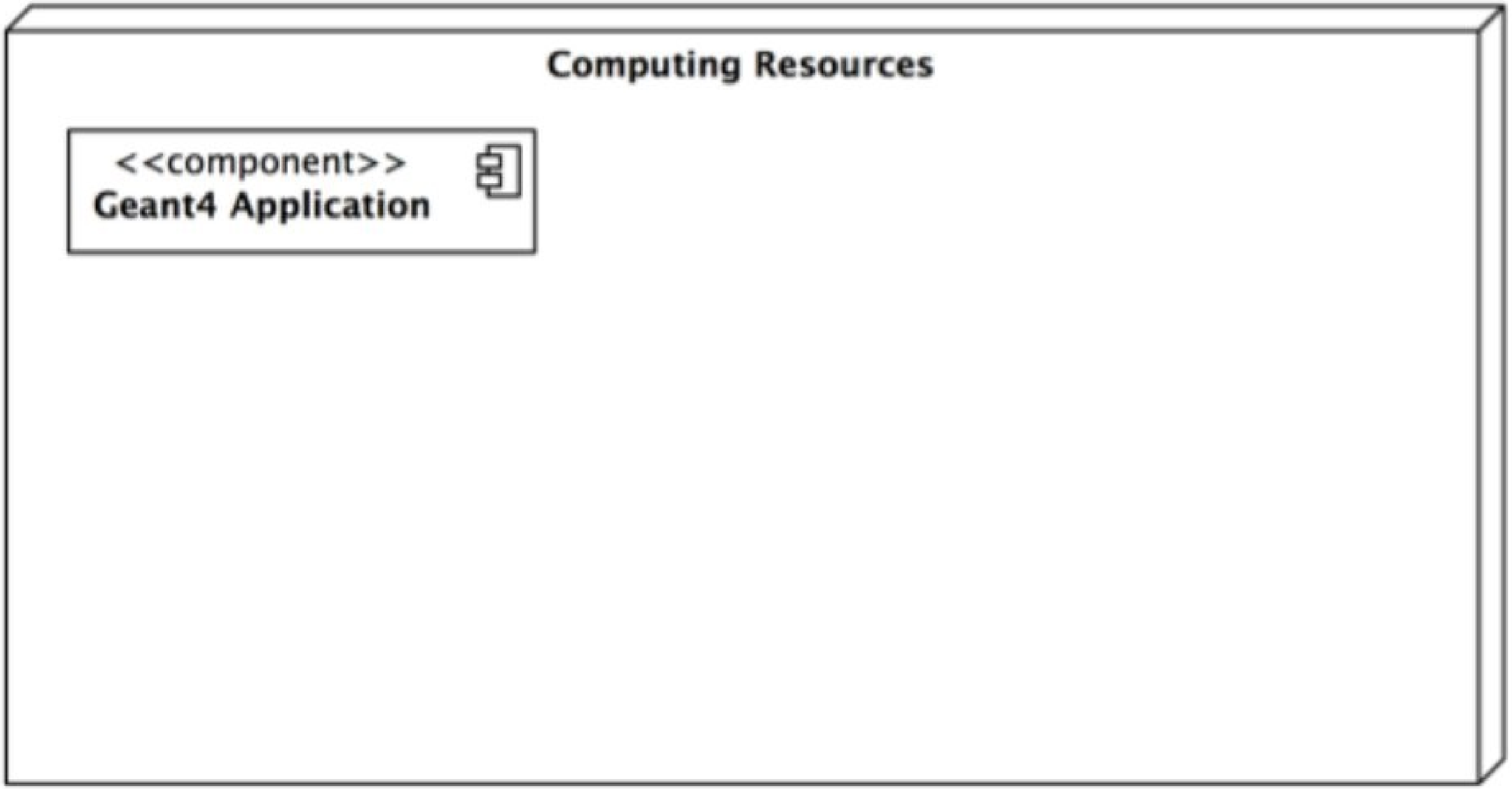
Introduction

- Modern CPU architectures:
  - Increasing number of processors & memory, but memory cost scales slower => Less memory/core
- Memory and its access will limit number of concurrent processes running on single chip
- Solution: add parallelism in the application code
- Geant4 needs back-compatibility with user code and simple approach (physicists != computer scientists)

What Is a Thread

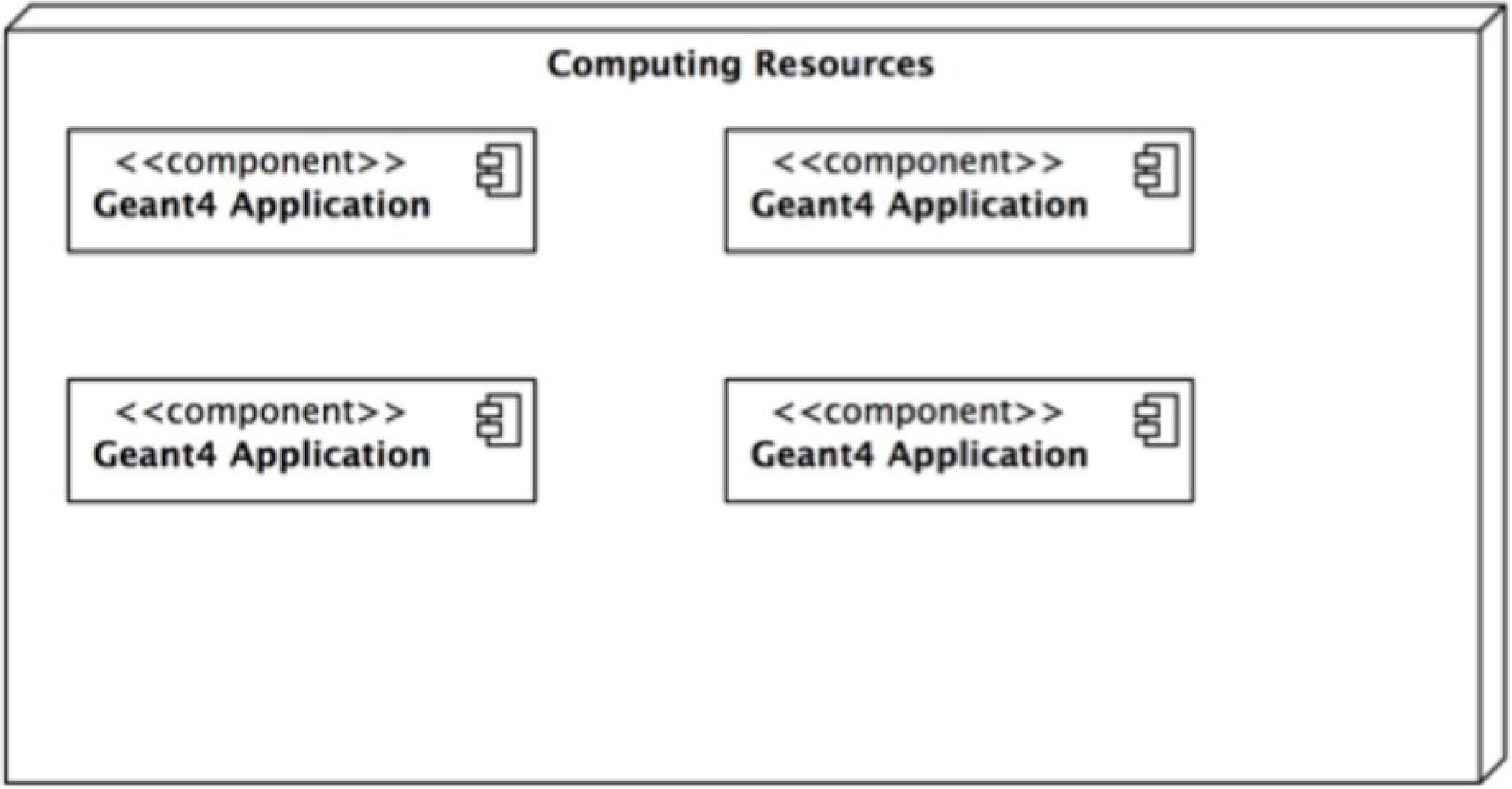
What Is a Thread ?

- Sequential application - one core



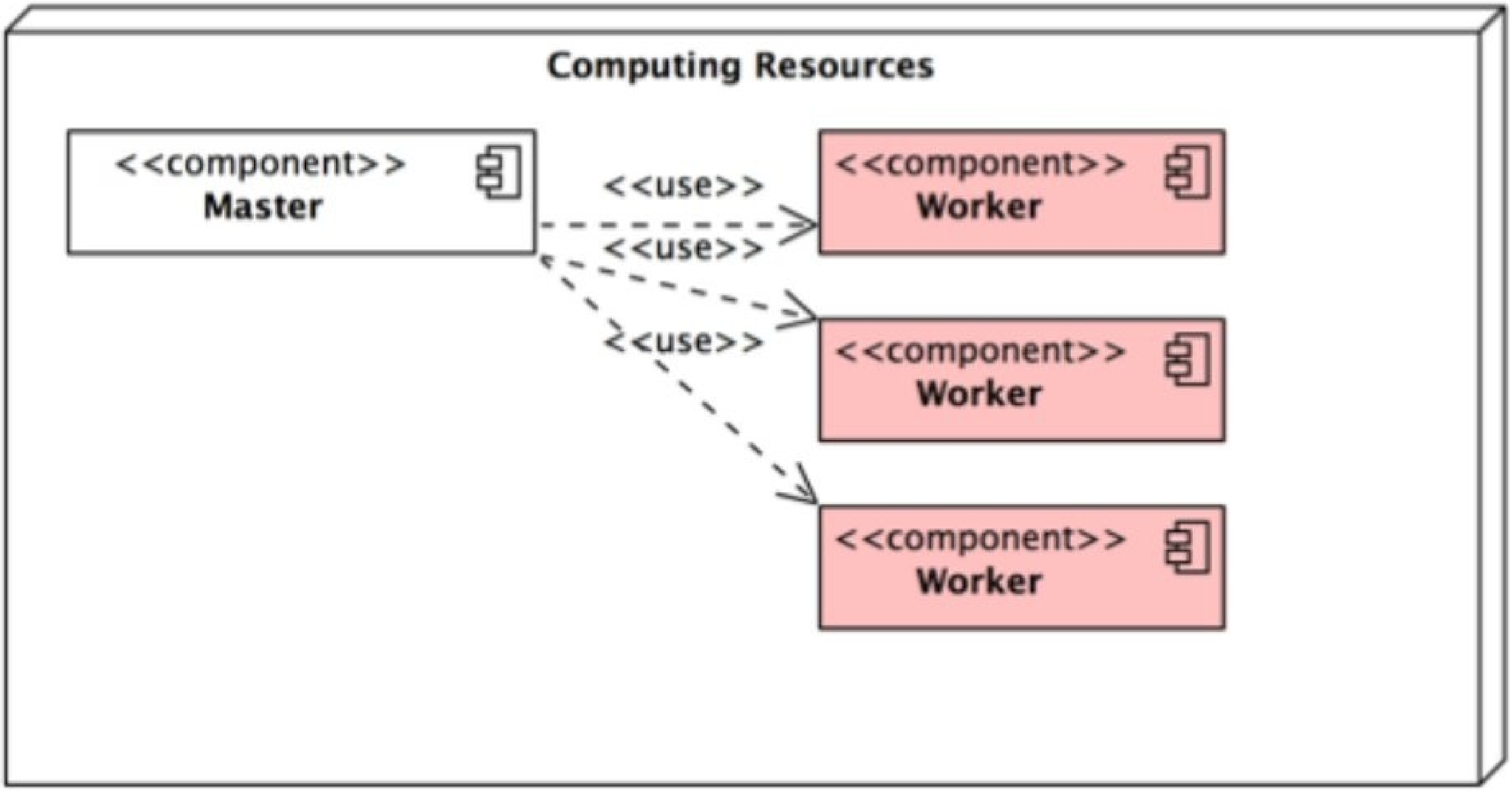
What Is a Thread ? (2)

- Sequential application – start N (cores/CPUs) copies of an application if it fits in memory



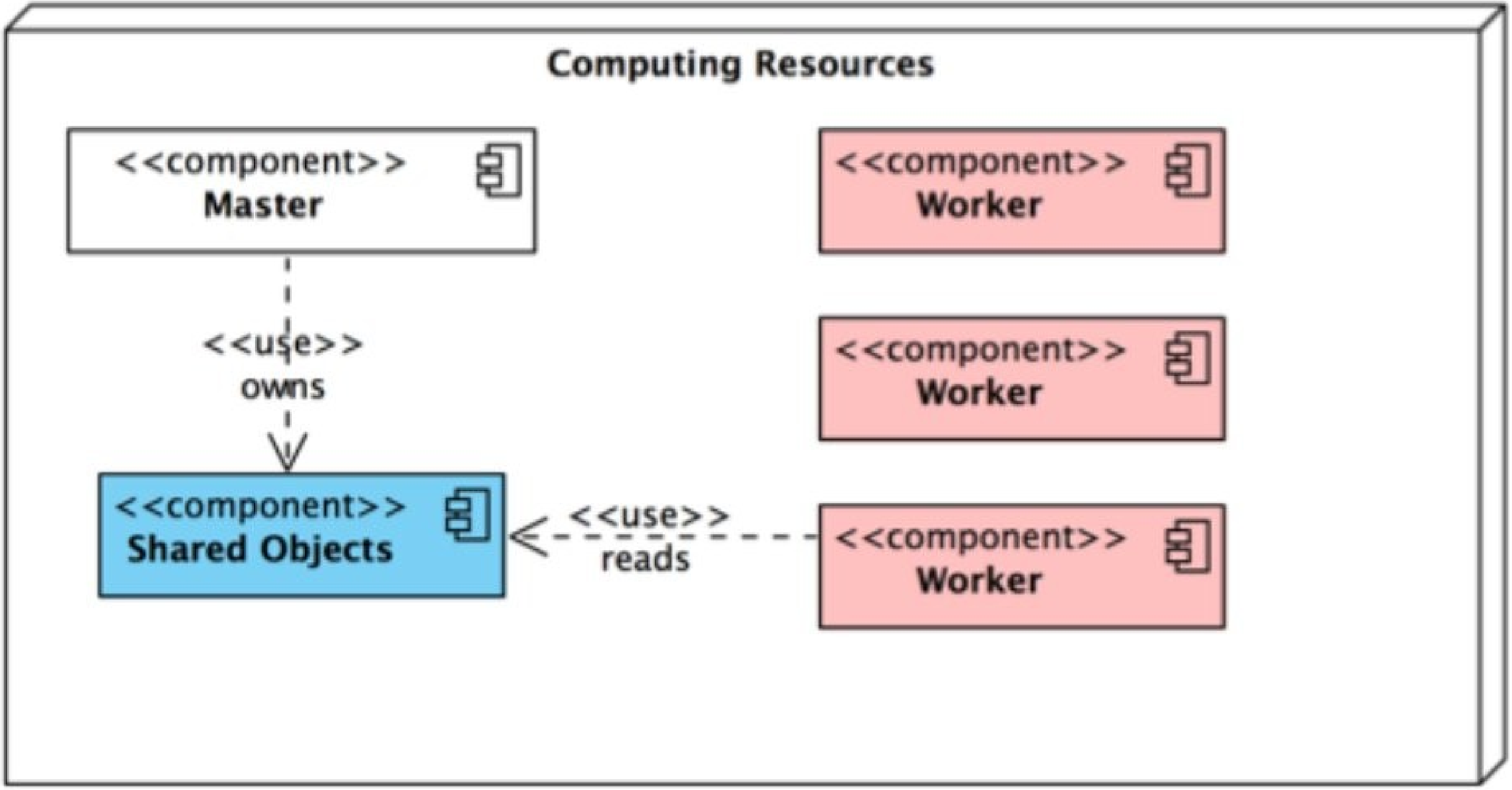
What Is a Thread ? (3)

- MT application – a single application starts threads.

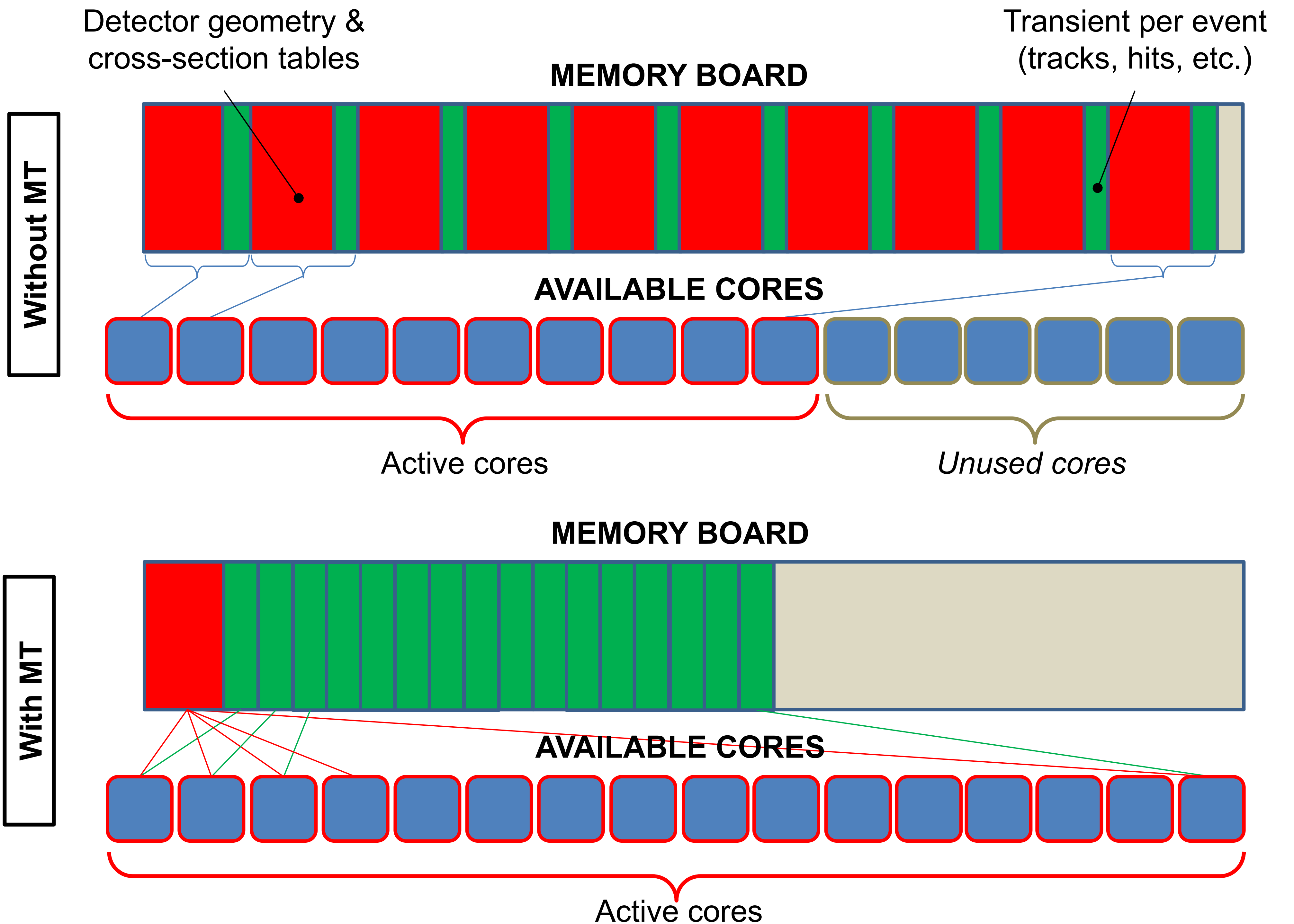


What Is a Thread ? (4)

- Memory reduction: when shared objects are introduced, memory of N threads is less than memory used by N copies of the application



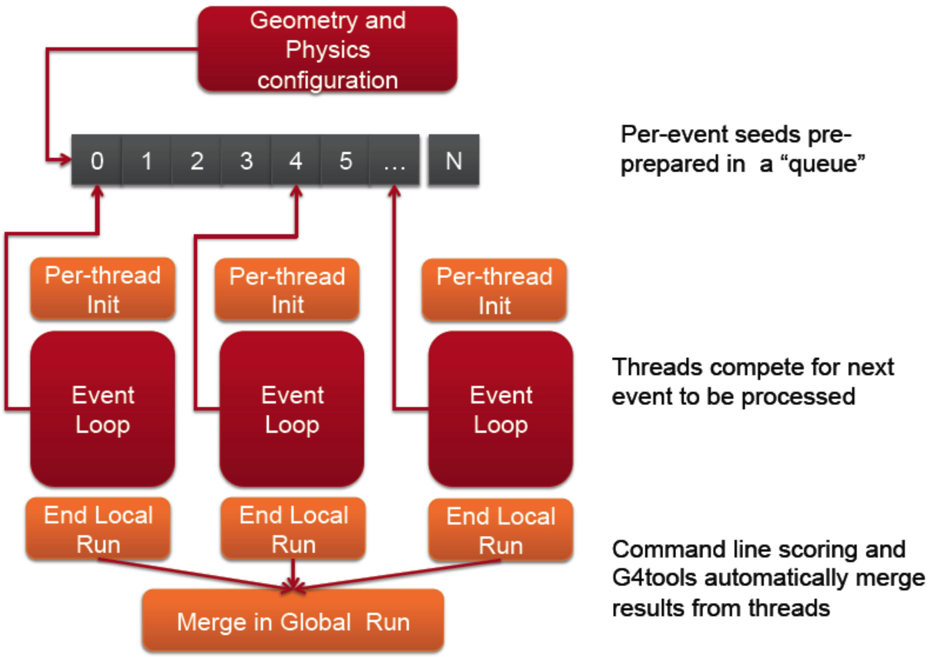
Why Multithreading


Detector geometry & cross-section tables
Transient per event (tracks, hits, etc.)
MEMORY BOARD
Without MT
AVAILABLE CORES
Active cores
Unused cores
MEMORY BOARD
With MT
AVAILABLE CORES
Active cores

Multithreading in Geant4

Multi-threading in Geant4

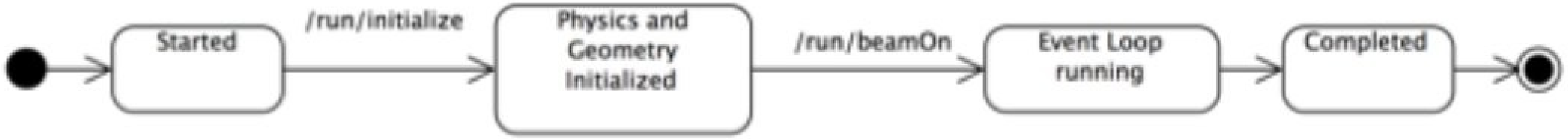
- General design choice: event level parallelism via multi-threading (POSIX based, in 10.5 migration from POSIX threading to C++11 threading)



- Each worker thread proceeds independently
  - Initializes its state from a master thread
  - Identifies its part of the work (events)
  - Generates hits in its own hits-collection
- Geant4 automatically performs reductions (accumulation) when using scorers, G4Run derived classes or g4tools

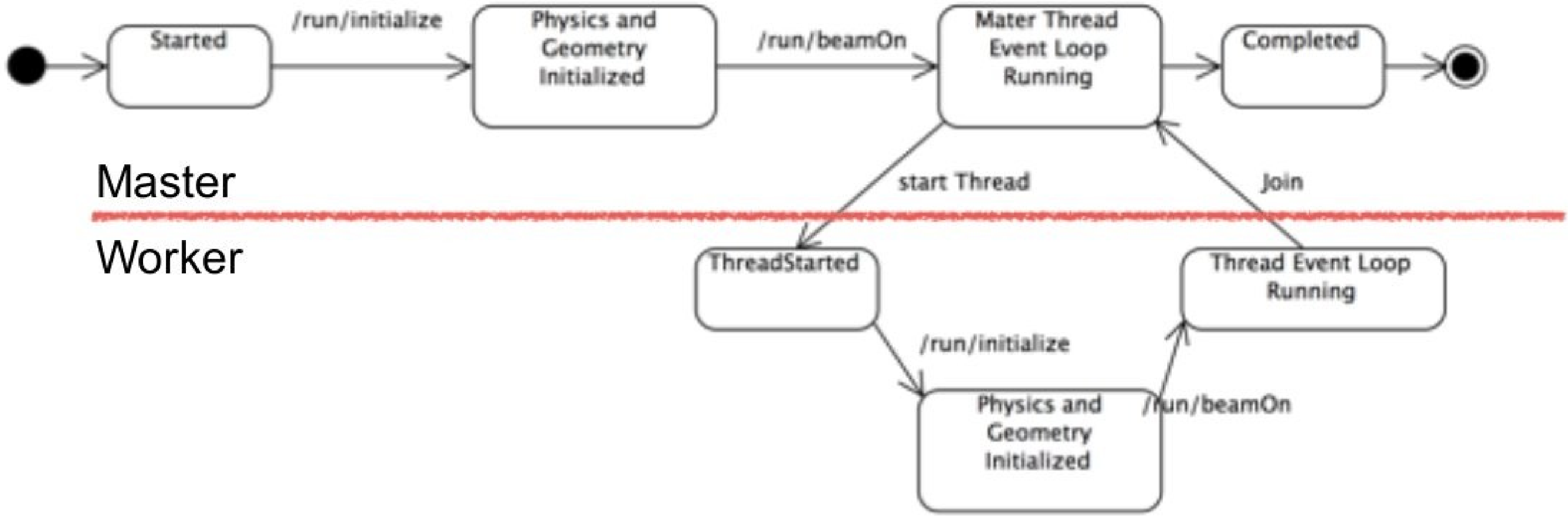
Simplified Master / Worker Model

- A Geant4 application (in MT mode) can be seen as simple finite state machine



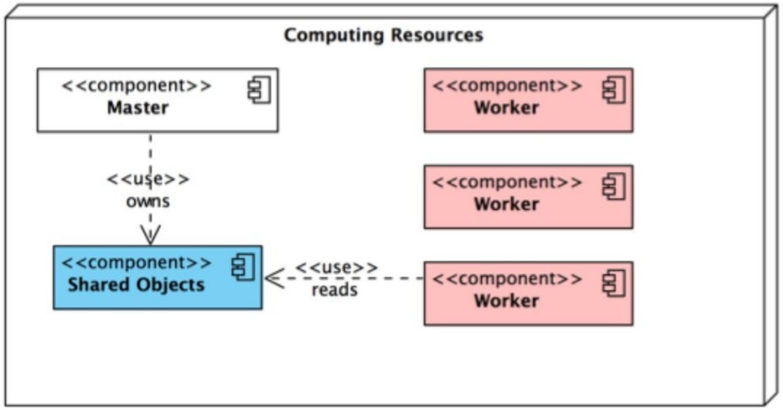
Simplified Master / Worker Model (2)

- A Geant4 application (in MT mode) can be seen as simple finite state machine
- Threads do not exists before first /run/beamOn
- When master starts the first run spawns threads and distribute work!



Shared Memory

- To reduce memory footprint threads must share at least part of the objects
- General rule in Geant4: threads can share whatever is invariant during the event loop (e.g. threads do not change these objects while processing events, these are used “read-only”)
  - Geometry definition
  - Electromagnetic physics tables



Shared ? Private?

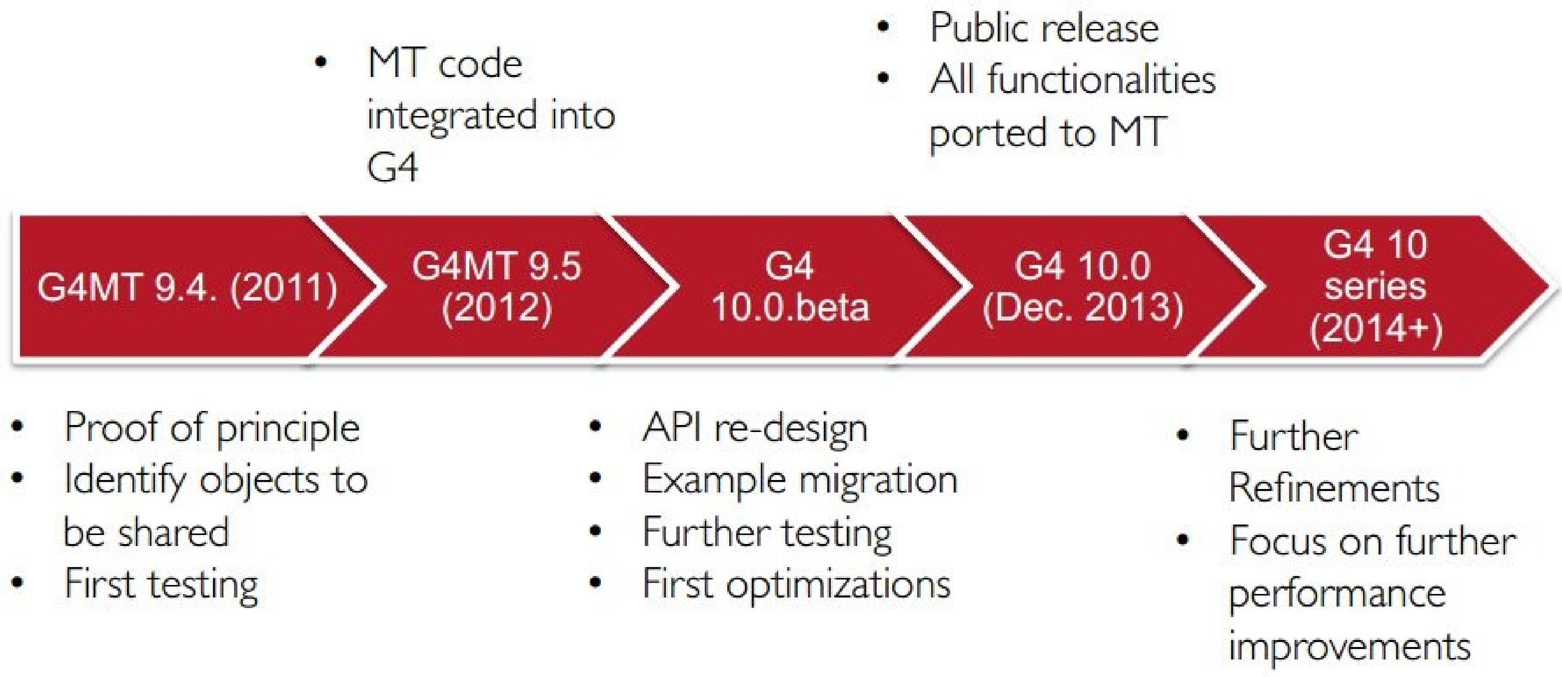
- In the multi-threaded mode
  - data that is stable during the event loop is shared among threads, while
  - data that is transient during the event loop is thread-local.

- **Shared** by all threads: stable during the event loop
  - Geometry
  - Particle definition
  - Cross-section tables
  - User-initialization classes

- **Thread-local**: dynamically changing for every event/track/step
  - All transient objects such as run, event, track, step, trajectory, hit, etc.
  - Physics processes
  - Sensitive detectors
  - User-action classes

Geant4 MT

- Event level parallelism via multithreading (POSIX based)
- Built on top of experience of G4MT prototypes
  - Capitalizing the work started back in 2009 by X.Dong and G.Cooperman, Northeastern University
- Main design driving goal: *minimize user-code changes*
- Integrated into Version 10.0 codebase



Geant4 10.00

- This is the first release (December 2013) with multithreading capability with event parallelism
  - Two build options: Multithreaded and Sequential mode, selection via a cmake configuration option -DGEANT4_BUILD_MULTITHREADED=ON
- Maximum back-compatibility with user code - however some API had to changed to enable MT (this is why this is a major release)
  - An application developed for Geant4 version 9.6 can be used without changing the code in sequential mode (except for other mandatory modifications not MT-related)
  - An MT-ready application, can also run in sequential mode without changing the code (but not vice-versa)

Multithreading Geant4 Application

Geant4 MT and User Application

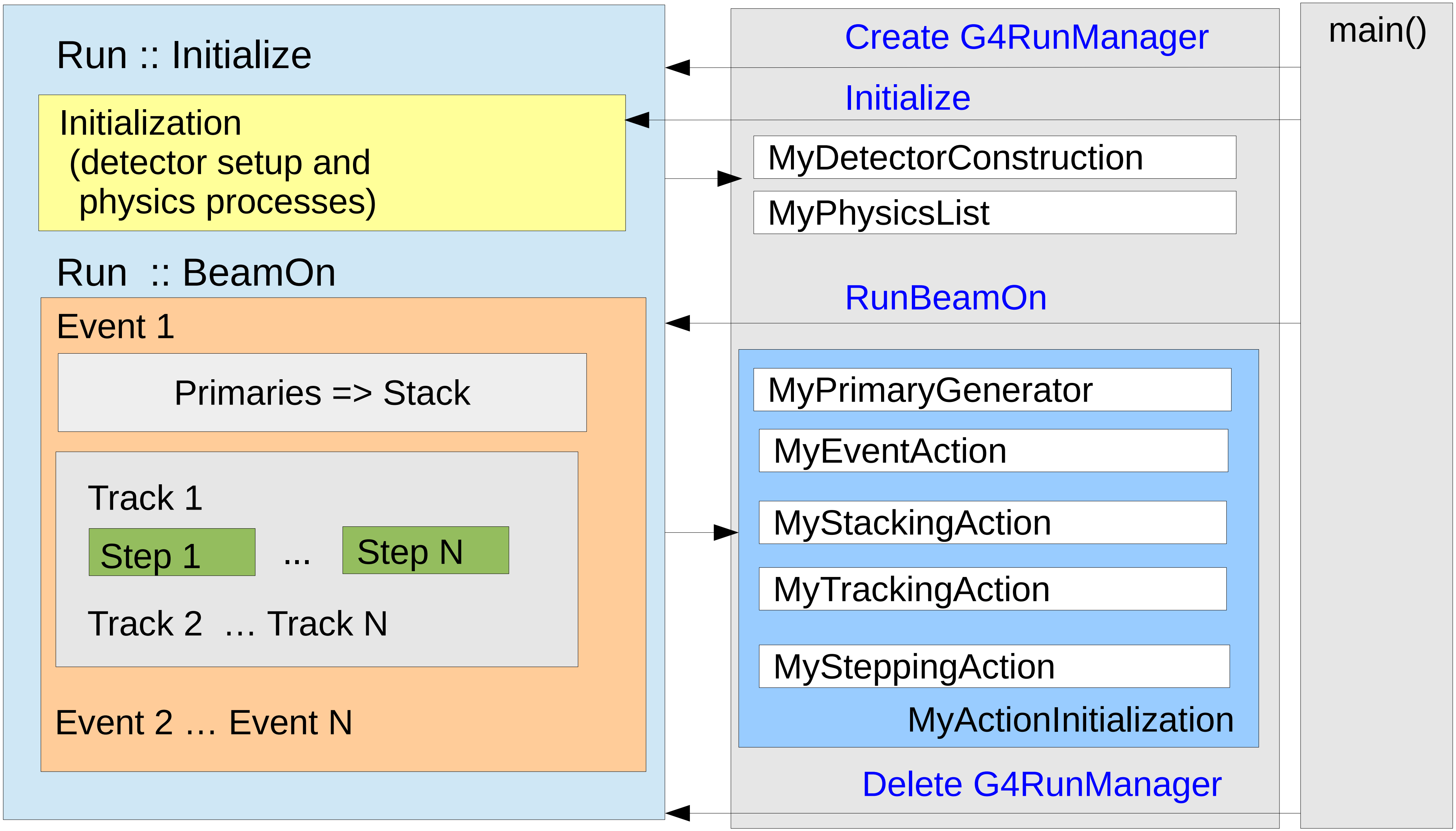
- Geant4 provides building blocks (bricks)

- Users have to assemble them to describe their scenario in their application program

Towards MT Application

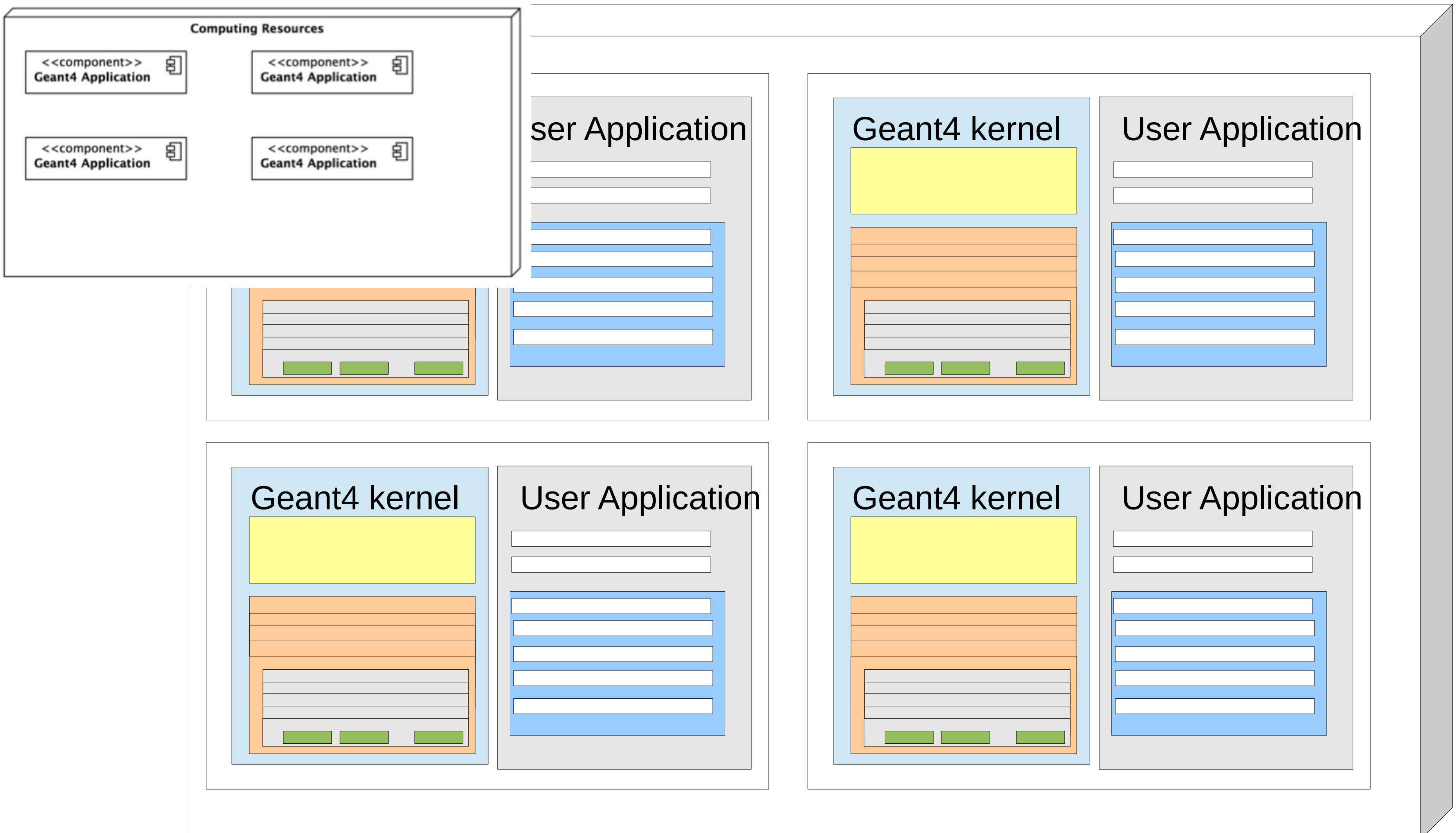
- Geant4 kernel takes care of steering event processing on workers
  - Use new G4RunManagerFactory class to create a G4RunManager derived class for steering MT run
- New Geant4 virtual methods/classes to be implemented in a user code
  - G4VUserActionInitialization – mandatory
  - G4VUserDetectorConstruction::ConstructSDandField() - for applications with field and/or sensitive detectors
  - G4UserWorkerThreadInitialization – optional, for applications which want/need to customize some aspects of thread behavior
- Make your application thread-safe

Geant4 Kernel & User Application In Sequential Mode

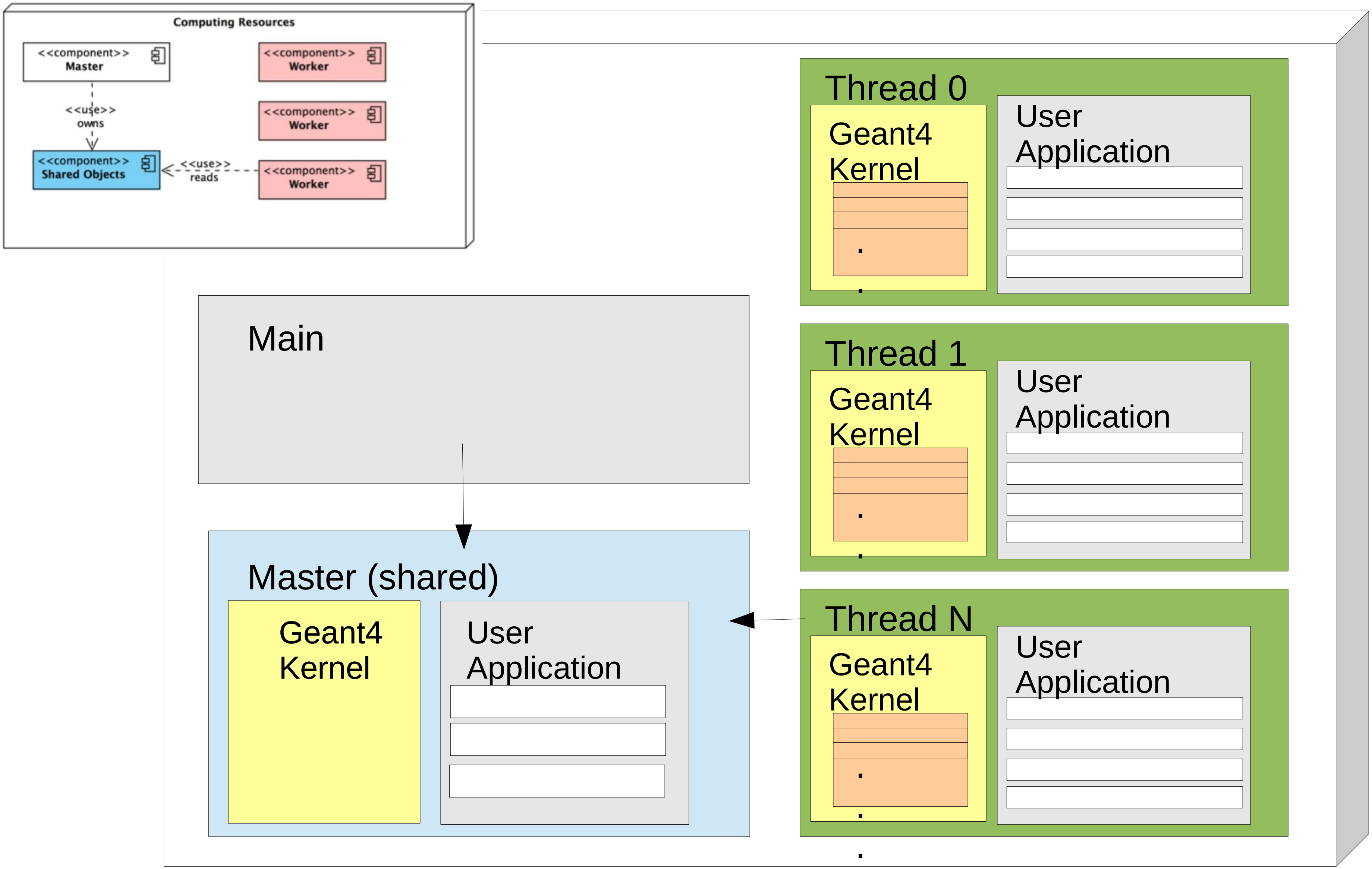


User Application and Geant4 Kernel In Sequential Mode

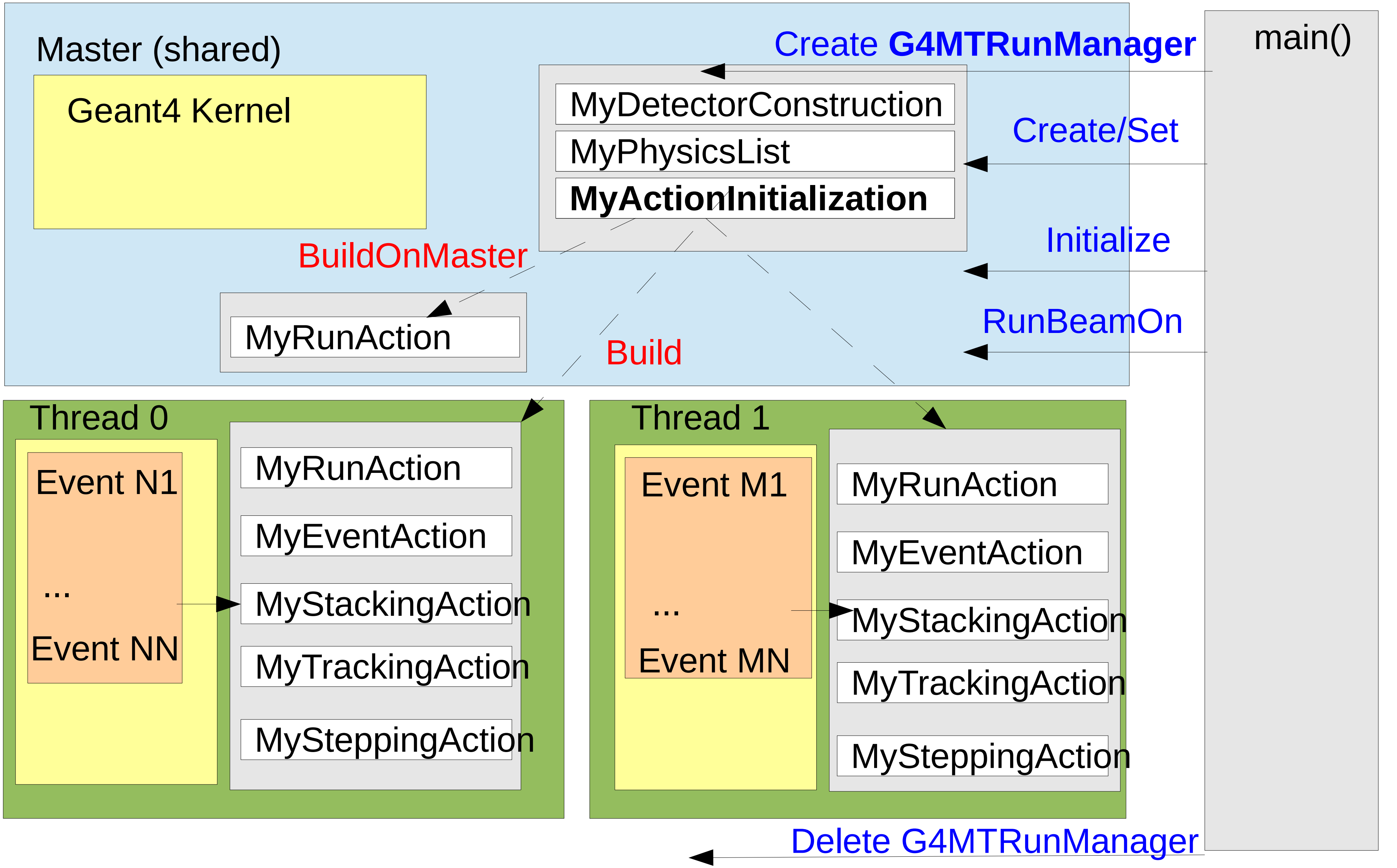
- Sequential application – start N (cores/CPUs) copies of an application if it fits in memory



User Application and Geant4 Kernel In MT Mode



User Application and Geant4 Kernel In MT Mode



main()

- Geant4 does not provide the main().
- In your main(), you have to
  - Construct G4RunManager derived class using G4RunManagerFactory
  - Define your initialization classes: MyDetectorConstruction and MyPhysicsList and set them to G4RunManager
  - Define your primary generator class (MyPrimaryGenerator) using your MyActionInitialization class and set it to G4RunManager
- You can also
  - Define optional user action classes and set them to G4RunManager using your ActionInitialization class
  - Define Geant4 visualization and (G)UI session via G4VisExecutive and G4UIExecutive and/or your persistency manager

main() - sequential

exampleED.cc

```
#include "DetectorConstruction.hh"
#include "ActionInitialization.hh"

#include "G4RunManager.hh"
#include "FTFP_BERT.hh"

int main(int argc,char** argv)
{
  // Create User Interface and enter in interactive session (1)

  // Construct the default run manager
  G4RunManager* runManager = new G4RunManager;

  // Detector construction
  runManager->SetUserInitialization(new ED::DetectorConstruction());

  // Physics list
  G4VModularPhysicsList* physicsList = new FTFP_BERT;
  runManager->SetUserInitialization(physicsList);

  // User action initialization
  runManager->SetUserInitialization(new ED::ActionInitialization());

  // Create User Interface and enter in interactive session
  ...
}
```

main() - MT

exampleED.cc

```
#include "DetectorConstruction.hh"
#include "ActionInitialization.hh"

#include "G4RunManagerFactory.hh"
#include "FTFP_BERT.hh"

int main(int argc,char** argv)
{
  // Create User Interface and enter in interactive session (1)
  ...
  // Construct the default run manager
  auto* runManager =
    G4RunManagerFactory::CreateRunManager(G4RunManagerType::Default);

  // Detector construction
  runManager->SetUserInitialization(new ED::DetectorConstruction());

  // Physics list
  G4VModularPhysicsList* physicsList = new FTFP_BERT;
  runManager->SetUserInitialization(physicsList);

  // User action initialization
  runManager->SetUserInitialization(new ED::ActionInitialization());

  // Create User Interface and enter in interactive session
  ...
}
```

The default run manager type supports MT

User Action Initialization

- The initialization and action classes which are called during event processing **MUST** be defined all together in the user action initialization class derived from G4VUserActionInitialization abstract base class.
  - Note that use of this class is mandatory for multithreading processing
- Implement the virtual method Build(), where you
  - Instantiate all initialization and action **classes called during event processing**
  - This method is called in MT mode on the workers
- Optionally, implement the virtual method BuildForMaster(), where you
  - Instantiate all initialization and action classes **called during event processing** which should be **build on master**
  - Typically, RunAction is created both on master and workers

ActionInitialization.hh

```
#include "G4VUserActionInitialization.hh"

namespace ED
{
class ActionInitialization : public G4VUserActionInitialization
{
  public:
    ActionInitialization();
    virtual ~ActionInitialization();

    virtual void Build() const;
};
}
```

Sequential

ActionInitialization.cc

```
#include "ActionInitialization.hh"
#include "PrimaryGeneratorAction.hh"
#include "EventAction.hh"

namespace ED
{
ActionInitialization::ActionInitialization()
{}

void ActionInitialization::Build() const
{
  SetUserAction(new PrimaryGeneratorAction);
  SetUserAction(new EventAction);
}
}
```

Action Initialization - .hh - MT-ready

ActionInitialization.hh

```
#include "G4VUserActionInitialization.hh"

namespace ED
{

class ActionInitialization : public G4VUserActionInitialization
{
  public:
    ActionInitialization();
    virtual ~ActionInitialization();

    virtual void Build() const;
    virtual void BuildForMaster() const;
};

}
```

Function called only in MT mode

Action Initialization - .cc - MT-ready

ActionInitialization.cc

```
#include "ActionInitialization.hh"
#include "PrimaryGeneratorAction.hh"
#include "EventAction.hh"

namespace ED
{

// ...

void ActionInitialization::Build() const
{
  SetUserAction(new PrimaryGeneratorAction);
  SetUserAction(new EventAction);
  SetUserAction(new RunAction);
}

void ActionInitialization::BuildForMaster() const
{
  SetUserAction(new RunAction);
}

}
```

Function called only in MT mode

Geometry

- To describe your detector you have to derive your own concrete class from G4VUserDetectorConstruction abstract base class.
- Implement the virtual method Construct(), where you
  - Instantiate all necessary materials
  - Instantiate volumes of your detector geometry
  - Optionally, create regions, visualization attributes
  - All these geometry objects (materials, volumes, ...) are created **in shared memory** (on master)
- Optionally, implement the virtual method ConstructSDandField(), where you
  - Instantiate your sensitive detector classes and set them to the corresponding logical volumes
  - Instantiate magnetic (or other) field
  - Using ConstructSDanField() is **mandatory** with multi-threading
  - Sensitive detectors and field are created **on workers**

Physics

- Physics list is instantiated in main()
  - Its is created in **shared memory** (on master)
- Physics lists provided in Geant4 are MT-ready
  - Nothing to be done on the user side in this case
  - Particles are constructed via call to ConstructParticle() **in shared memory** (on master)
  - Physics processes are constructed via call to ConstructProcess() **on workers**
- If you define your own physics list
  - Make sure that all process objects are instantiated in the ConstructProcess() method and NOT in the physics list constructor
  - If it includes ions, add G4GenericIon::GenericIonDefinition() into ConstructParticle() method. This ensures that all ions (including light ions such as deuteron, alpha) work properly.

Scoring

- Geant4 sensitive, hits collections are MT ready
  - Hits objects, as well as sensitive detectors, are instantiated on workers, that's why the G4Allocator declared with hit class need to be defined thread-local - add G4ThreadLocal keyword

MyHit.hh sequential

```
extern G4Allocator<MyHit>*  MyHitAllocator;
```

MyHit.cc sequential

```
G4Allocator<MyHit>*  MyHitAllocator = 0;
```

MyHit.hh MT-ready

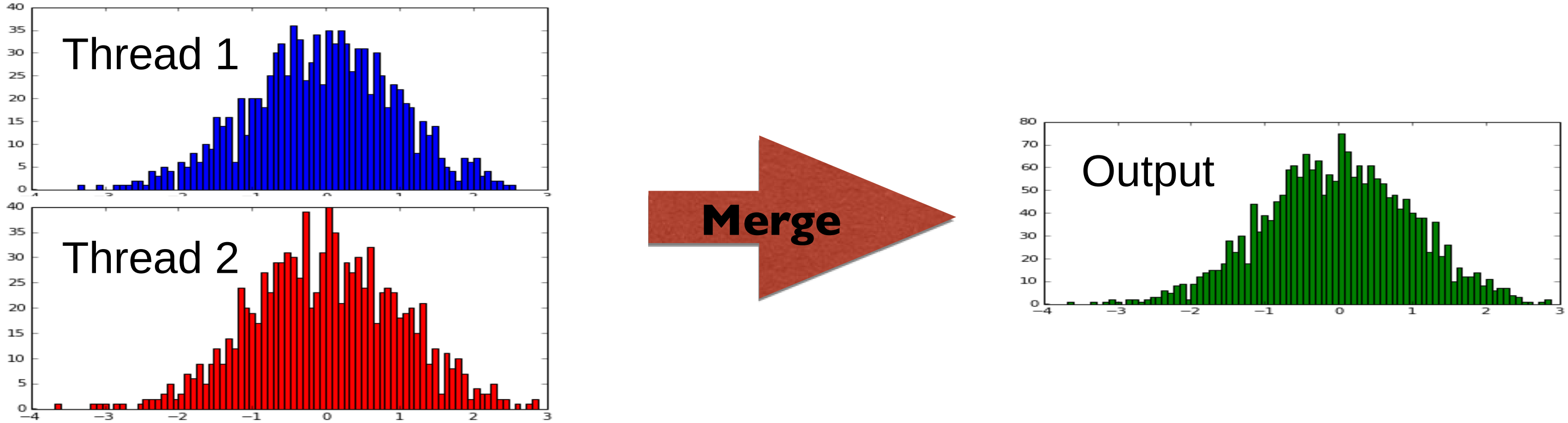
```
extern G4ThreadLocal G4Allocator<MyHit>*  MyHitAllocator;
```

MyHit.cc MT-ready

```
G4ThreadLocal G4Allocator<MyHit>* MyHitAllocator = 0;
```

Analysis

- Geant4 analysis tools are MT-ready
- Histograms & profiles:
  - Each thread owns its own copy of given histograms & profiles
  - At the end of the run workers objects are “merged” into a single one on master
  - A single file with merged histograms and profiles will be produced
- When using G4AnalysisManager with histograms, the UserRunAction class must be instantiated **both on master and workers**



Analysis (2)

- Ntuples
  - Each thread owns a copy of ntuple
  - **Not merged** by default
- Output files
  - Each thread will write out a separate file, file names are generated automatically:
    - fileName[_ntupleName]_tid.ext
      - where tid = thread Identifier (0,1,2, ...), ext = root, xml, csv, hbook
- When using Root output merging can be activated using
  - `analysisManager->SetNtupleMerging(true);`

Visualization

- Geant4 visualization is MT-ready
- Visualization is done by master thread based on event keeping
- Events are drawn directly from worker threads as soon as any are ready

User Interface

- User interacts with application typing UI commands
  - Master thread “accumulates” the commands and passes the commands stack to all the threads at the beginning of a run
  - Threads execute the same commands sequence as master thread
- However some commands make sense only in master thread (e.g. the one modifying the geometry)
  - UI commands can be marked as “not to be broadcasted” via G4UIcommand::SetToBeBroadcasted(false);
- Do not forget this step if you implement user-defined UI commands

Conclusions

- Geant4 collaboration made a big effort to make writing Geant4 multi-threading application easy
  - We believe that just following the instructions is enough – for simple applications
- Parallelism is however a tricky business:
  - We will speak about race conditions in the second part of this presentation