
# Geant4 Multithreading and Tasking

Soon Yung Jun, Makoto Asai (JLab), John Apostolakis (CERN)

LArSoft Multithreading and Acceleration Workshop

March 2. 2023

# Geant4 Multithreading and Tasking

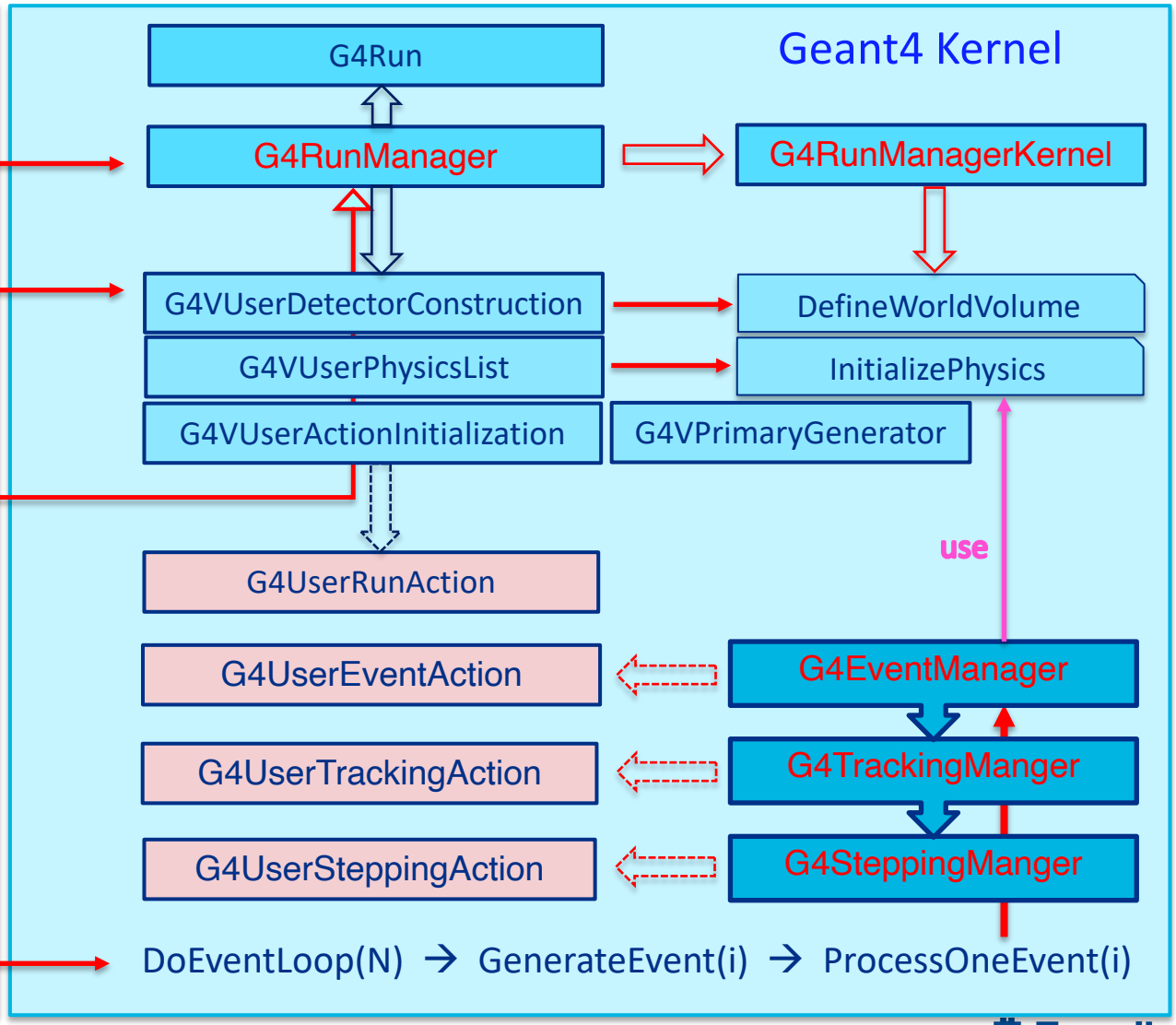
- Motivations and benefits of Geant4 multithreading
  - Take advantages of multi/many-core CPU architectures (FLOPS/Watt)
  - Maximize event throughput with resource (memory) sharing on a chip
  - Event-level parallelism is the natural choice as HEP events are independent and each event can be simulated separately
- Geant4 tasking extension to support
  - Sub-event level parallelism for better load balancing (Concurrency)
  - Task-level parallelism for efficient uses of resources (Heterogeneity)
- Introduction of Geant4 multithreading (2013) and tasking (2021)
  - POSIX pthread → Parallel tasking library (PTL, based on C++ thread)
  - Also support TBB (Intel® Thread Building Block) backend


Geant4 has been evolving continuously for more than two decades!

# Geant4 Serial (Sequential) Mode

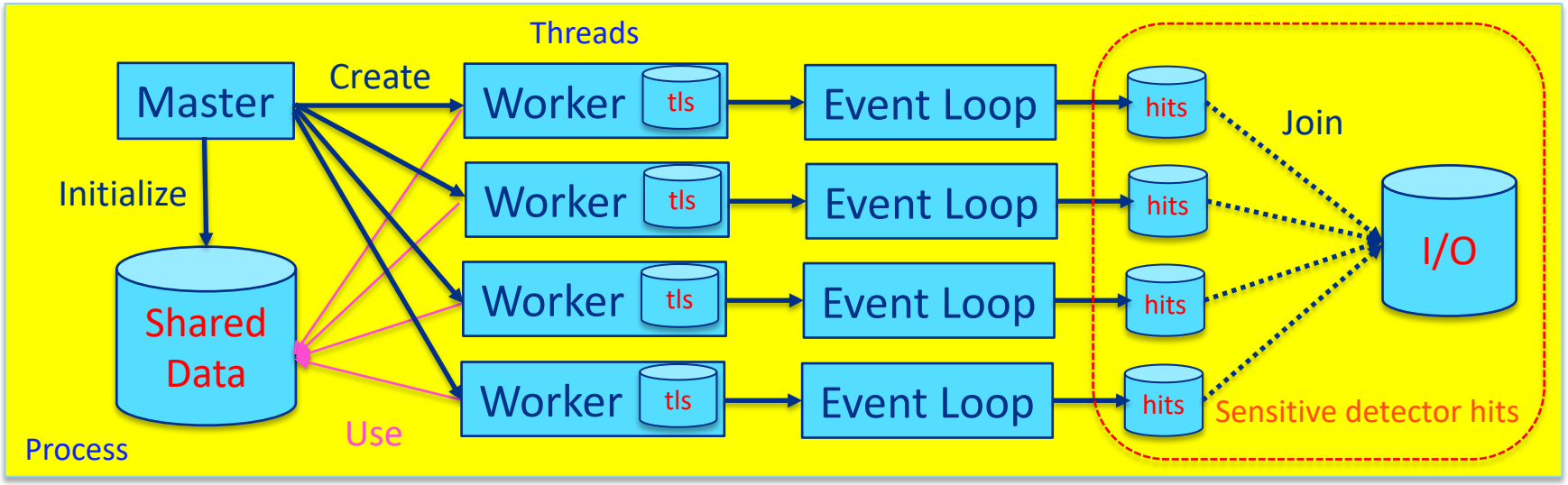
- Process one event at a time per CPU process

```
int main()
{

// Create the run manager

// Set mandatory classes
  DetectorConstruction
  PhysicsList
  ActionInitialization

// Initialize the run manager

//UI: BeamOn
  RunManager→BeamOn(N)
}
```



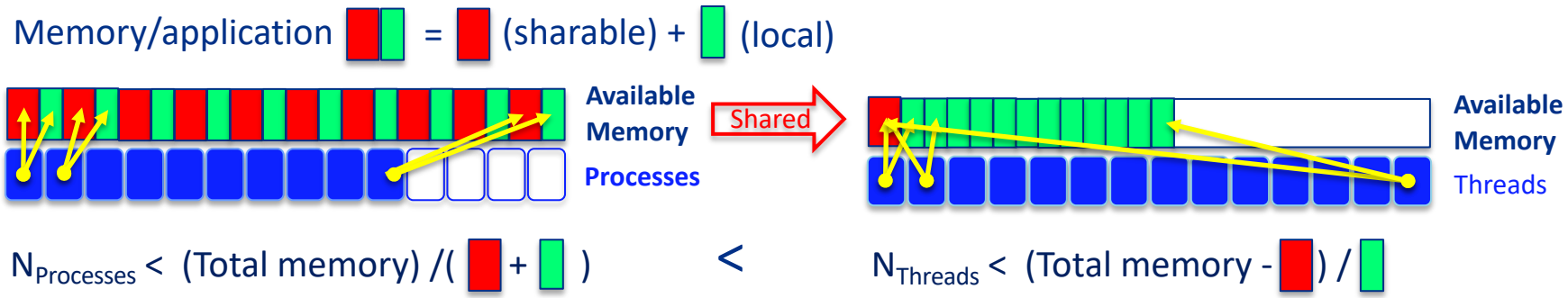
# Geant4 Multithreading Mode

- Design principles
  - Minimize changes in user-codes (maintain API changes at minimum)
  - Minimize mutex/lock (avoid deadlock and data race)
  - Build a simplified master/worker model
    - The master thread is responsible for initialization of global shared data (e.g., geometry and physics data) and configures worker threads, but does not process any event
    - Worker threads initialize thread local data and do actual work for the event processing (i.e., start the event loop and process events)



# Geant4 Event-level Multithreading

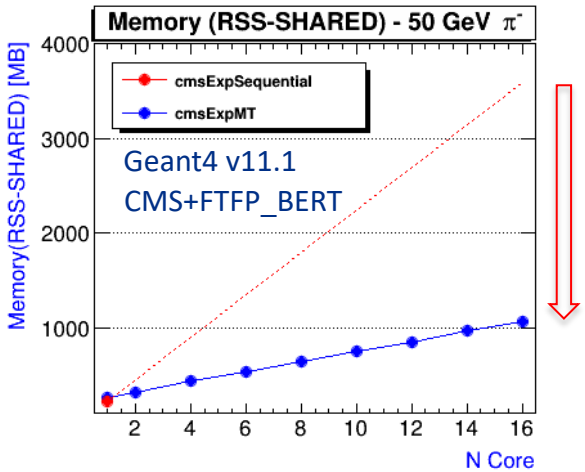
- Embarrassingly parallel application without any dependence and communication between parallel tasks
- Beneficial when forking N-processes of a Geant4 application requires more memory than resources provided by the system



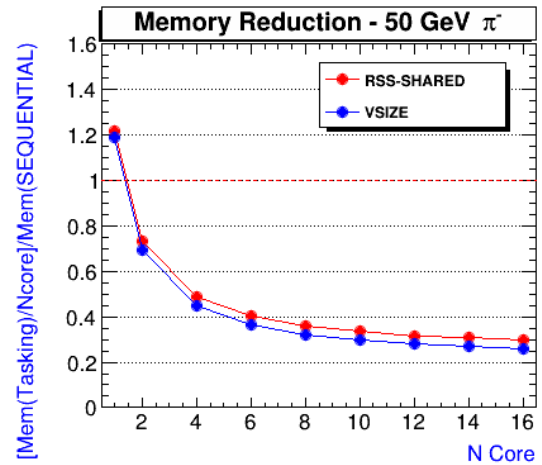

- A key consideration for multi-threading is a substantial amount of the memory shared by threads/tasks
  - data generated at initialization (geometry/material, EM processes)
  - data read in by multiple (hadronic) processes (lazy initialization)
- Data initialization is currently sequential – the full potential for sharing data is achieved after multiple events have been simulated

# Geant4 Multithreading Performance

- Memory reduction ($R_f$) by the fraction of shared memory ($f$) and the number of cores/threads ($N$): $Rf = (1 - f) + f/N$

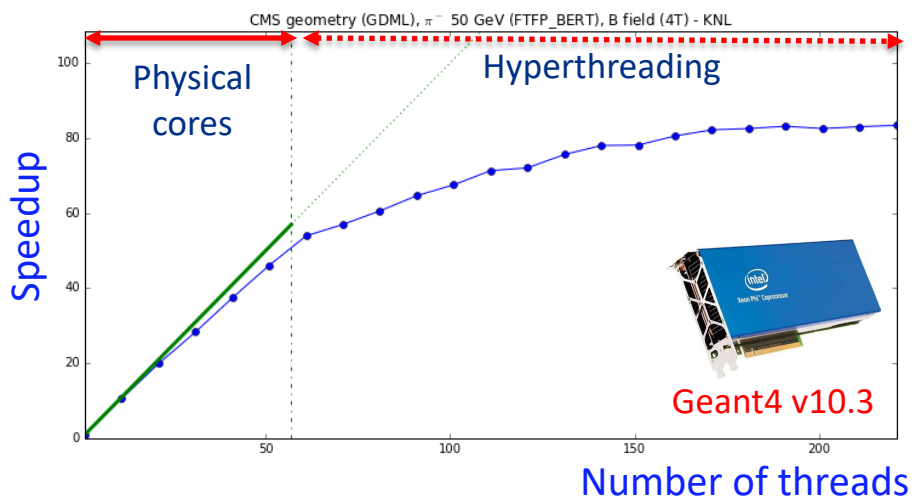



$R_f$



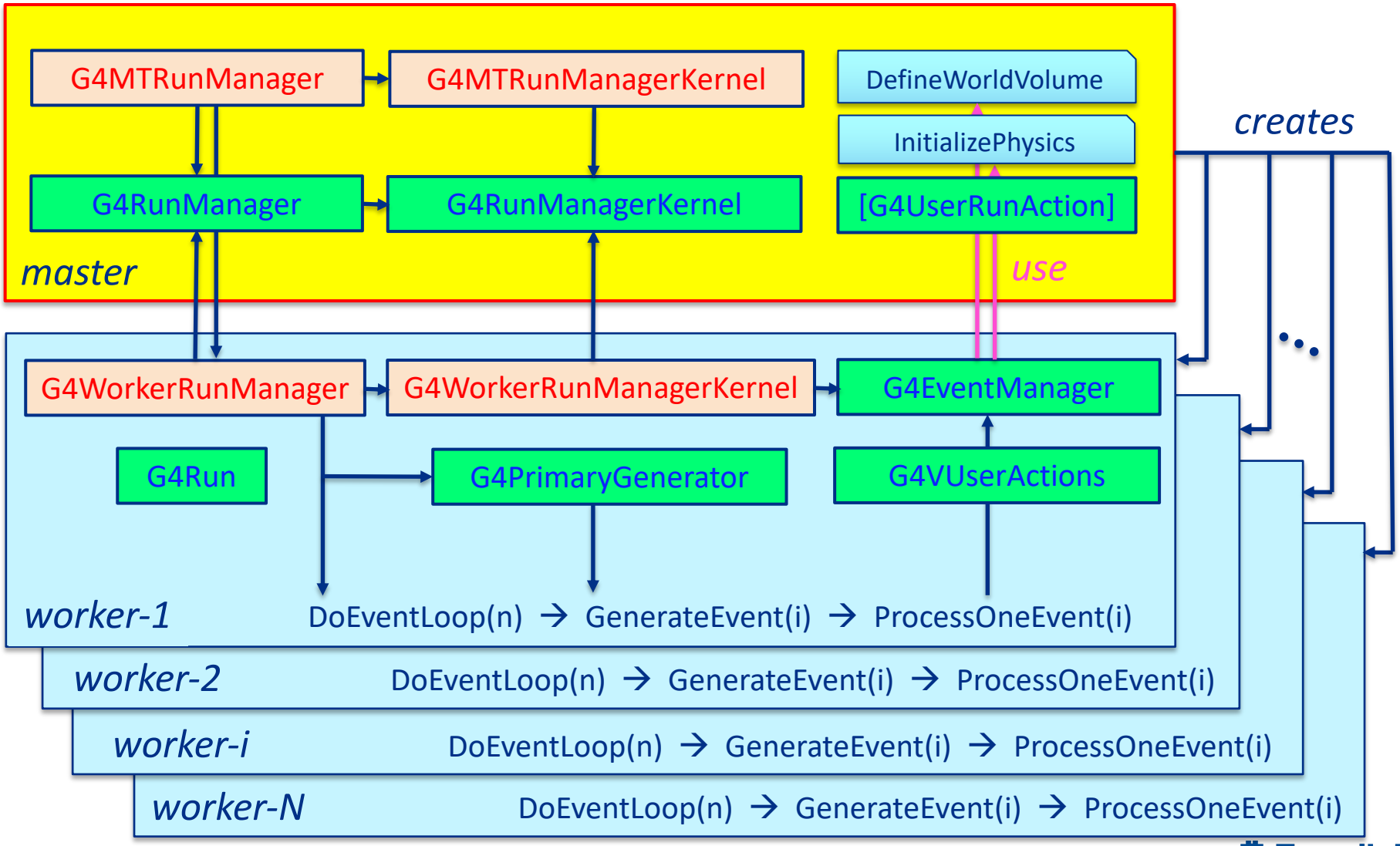

$f \sim 0.75$

- Scalability on Intel® Xeon Phi (KNL, speed up by the number of threads)

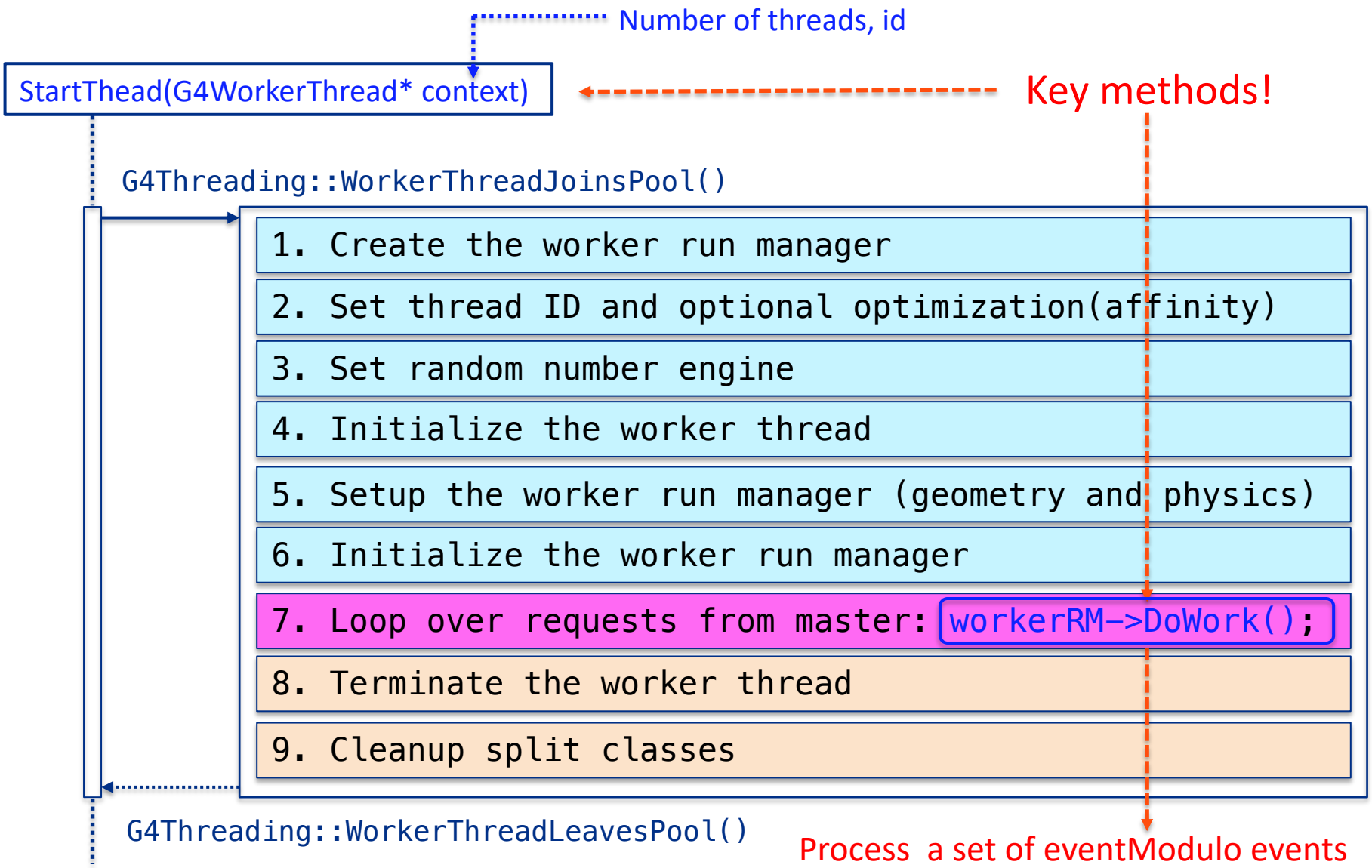


# Geant4 Multithreading Kernels

- Process n-eventModulo per thread (requested by the master)



# G4MTRunManagerKernel::StartThread



# Processing Events and Reproducibility

- EventModulo is the number of events that each worker thread is tasked to simulate before coming back to the master RunManager for requesting the next sub-set of events
  - Set by UI: /run/eventModule <M> <seedOnce>
  - Default (M = 0) EventModulo = int( $\sqrt{\frac{N_{total\_events}}{N_{threads}}}$ )
  - <seedOnce> specifies how frequent each worker thread is seeded by the random number sequence centrally managed by the master thread
- Random number engine and reproducibility
  - Each worker clones the random engine of the master thread
  - seedOnce = 0 (default): seeds are set for every event of every worker thread. This option guarantees event reproducibility regardless of number of threads.
  - seedOnce = 1: seeds are set only once for the first event of each run of each worker thread. Event reproducibility is guaranteed only if the same number of worker threads are used.

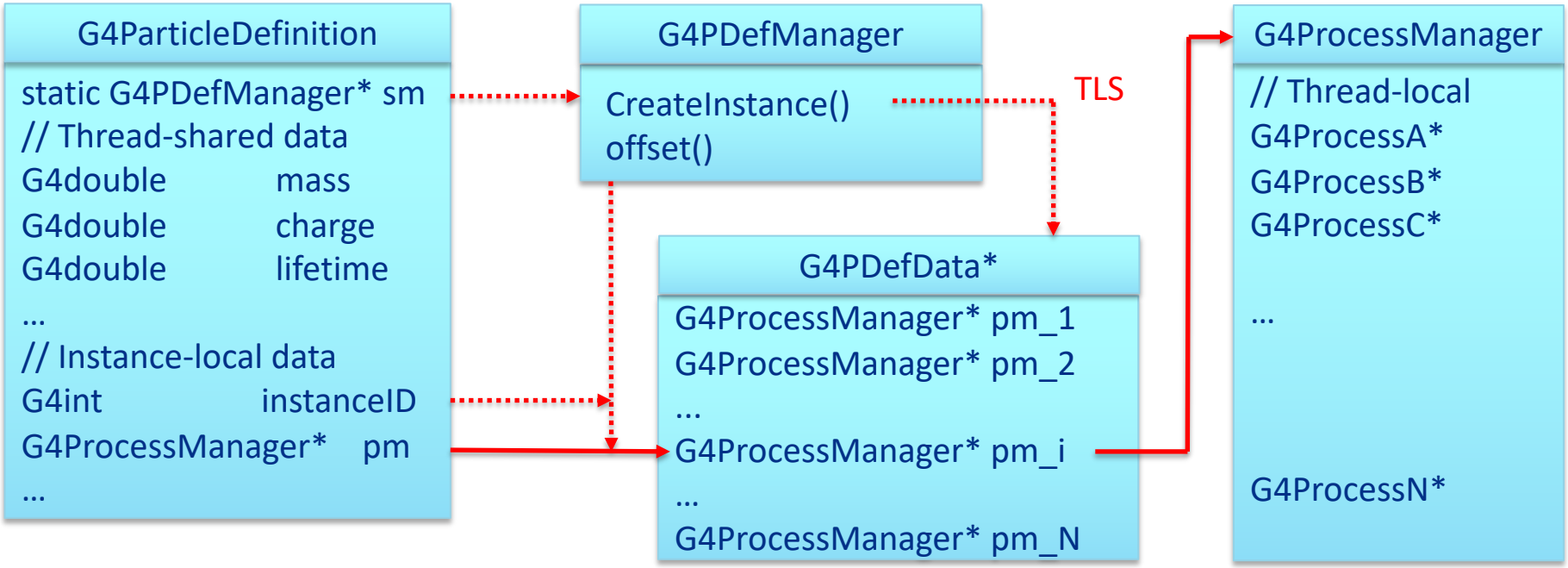
# Shared Data and Thread Safety

- General guideline in Geant4
  - Threads can share whatever is invariant (stable) during the event loop
  - Transient objects that are specific to each event are thread-local
- static class/data are shared among all class instances and threads
  - Geometry/Material (G4VUserDetectorConstruction)
  - Physics data (G4VPhysicsList)
- G4ThreadLocal instances (and objects that are created by them)
  - Classes designed to be per-thread, e.g., *G4WorkerRunManager*
  - *G4EventManager*, G4TrackingManager, G4SteppingManager, G4SDManager
  - *G4Event*, G4Track, G4Step, G4VHit, G4VSensitiveDetector
  - G4VUserAction, G4UserField, G4Transporation and G4Navigator, etc.
- The split-class allows sharing class instances among threads with a mixture of shared (read-only) and thread local storage data
  - Geometry related: G4LogicalVolume, G4PhysicalVolume, G4Region, ..
  - Physics related: G4ParticleDefinition, G4VPhysicsConstructors, etc.

# Split Class Mechanism: An Example

- Split class mechanism is to collect all per-thread objects into a separate helper class, instances of which are organized in an array, that is accessed via an index representing a unique identifier of a given class instance (thread-safety via TLS)
- *G4PaticleDefinition* (one of physics related split classes)

G4ParticleDefintion::GetProcessManager() → (sm.offset()[instanceID]).processManager → pm_i



# G4Mutex and G4AutoLock

- Support global mutex types and automatic unlocking mechanism
  - *G4Mutex* = std::mutex
  - *G4AutoLock* = std::unique_lock<G4Mutex>
- An example of *G4Mutex* for reading primary particles from a file
  - *PrimaryGeneratorAction*: G4ThreadLocal
  - *G4HEPEvtInterface*: static (shared by all the PrimaryGeneratorAction)
  - Without the mutex locking, each *GeneratePrimaryVertex(G4Event* event)* will open input file multiple times and read same events one by one

```
G4Mutex interface_mutex = G4MUTEX_INITIALIZER;
G4Mutex generator_mutex = G4MUTEX_INITIALIZER;
G4HEPEvtInterface* PrimaryGeneratorAction::hepEvt = nullptr;

PrimaryGeneratorAction::PrimaryGeneratorAction(G4String file_name)
{
    G4AutoLock scoped_lock(&interface_mutex);
    if( interface == nullptr ) interface = new G4HEPEvtInterface(file_name);
}

void PrimaryGeneratorAction::GeneratePrimaries(G4Event* event)
{
    G4AutoLock scoped_lock(&generator_mutex);
    interface->GeneratePrimaryVertex(event);
}
```

# Geant4 Tasking

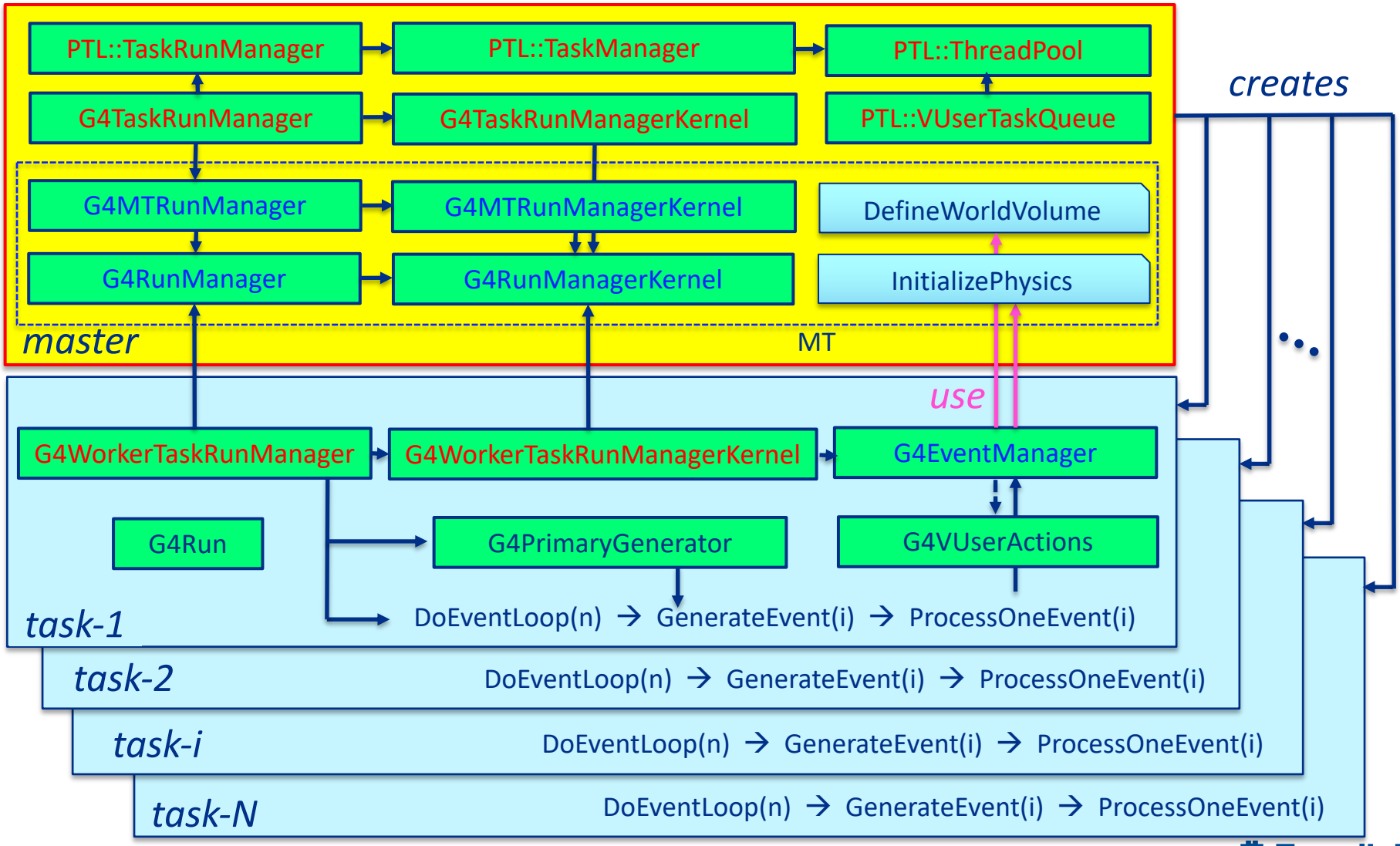
- Geant4 supports a task-based framework (*G4Tasking*) from v11.0
  - Based on PTL (parallel tasking library, developed by J. Madsen (AMD), lightweight tasking system featuring thread-pool, task-group, and task-queue using the C++ thread) or the TBB backend
  - Support *G4RunManagerType* = {Serial, MT, Tasking,TBB} using *G4RunManagerFactory* or environment variables

```
auto* rm = G4RunManagerFactory::CreateRunManager(G4RunManagerType::Tasking);
```

- G4Tasking opens opportunities for task-level parallelism
  - Sub-event level parallelism (from events to tracks)
    - A group of selected particles can be executed in a thread-pool
    - A group of special tasks can be a task-group
  - Heterogeneous computing with more diverse processing elements
    - Heavy CPU centric (HTC) → combination of CPUs + Accelerators (GPUs)
    - Workload decomposition and optimization for very specific domain tasks for the efficiency triplet (power, performance, cost)

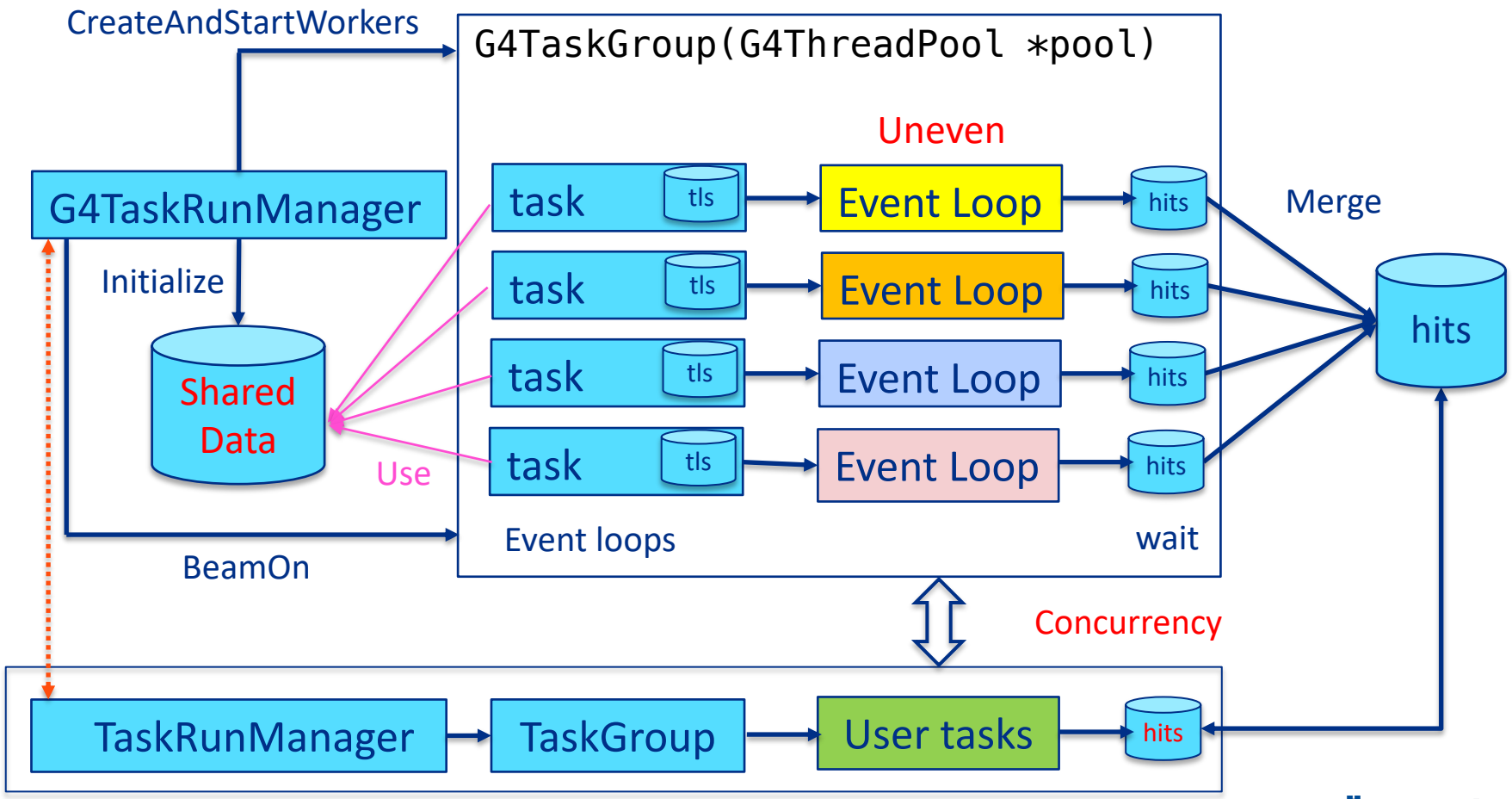
# Geant4 Task Mode

- Process a task per thread by *G4TaskRunManager*



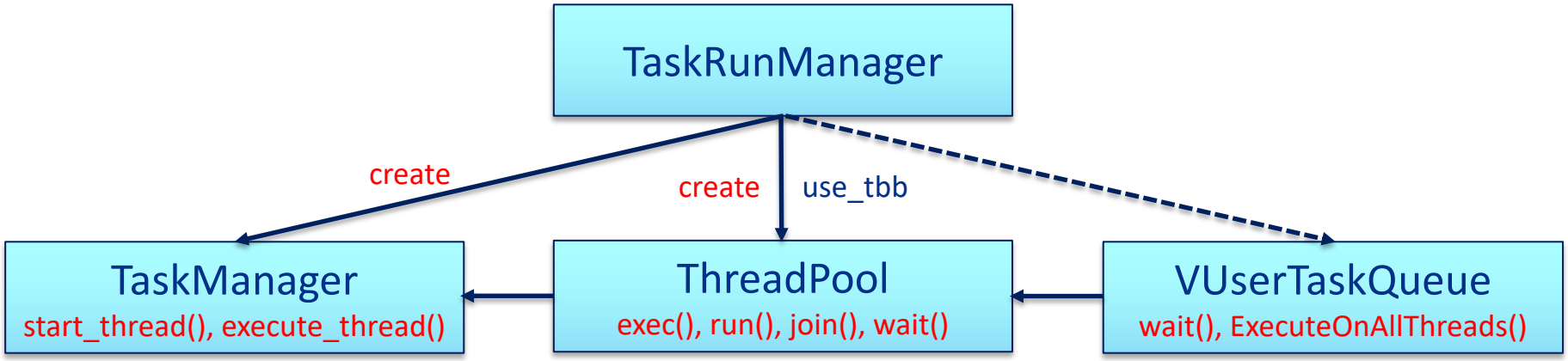
# Geant4 Multithread vs. Tasking

- What are main differences: *G4Tasking* facilitates
  - Better load balancing: one thread can process more events than others
  - User defined tasks with *PTL::TaskRunManager*



# Key Components of Geant4 Tasking

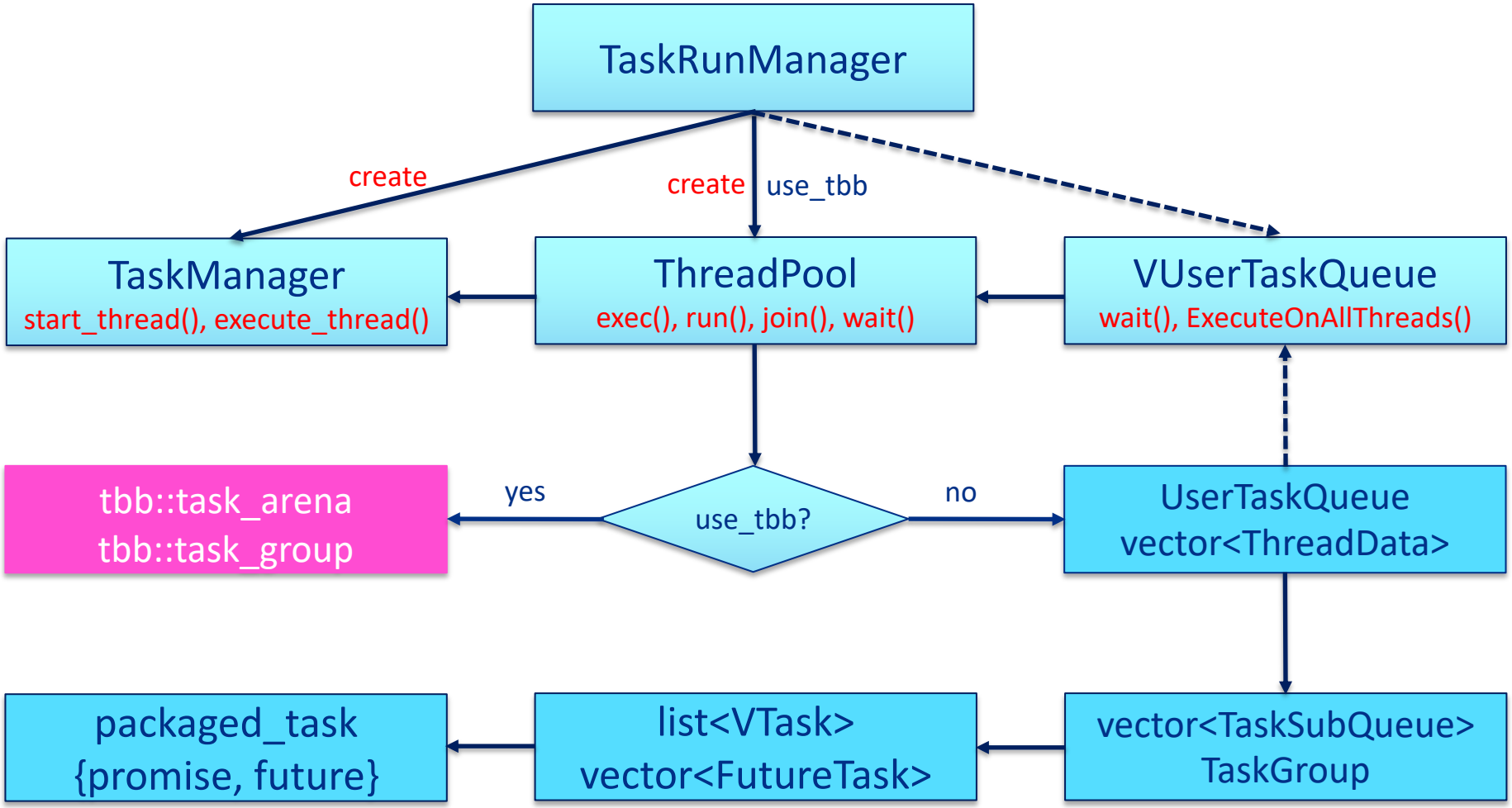
- TaskRunManager: a class for run control in tasking for multi-threaded runs which initializes ThreadPool and TaskManager




- ThreadPool: a class for an efficient thread-pool that accepts work in the form of tasks
- TaskManager: a class for handling the wrapping of functions into task objects and submitting them to a thread pool
- VUserTaskQueue: an abstract base class for creating a task queue used by ThreadPool

# Tasking Types: Native (PTL) vs. TBB

- Supports both native TaskGroup/UserTaskQueue and Intel® TBB

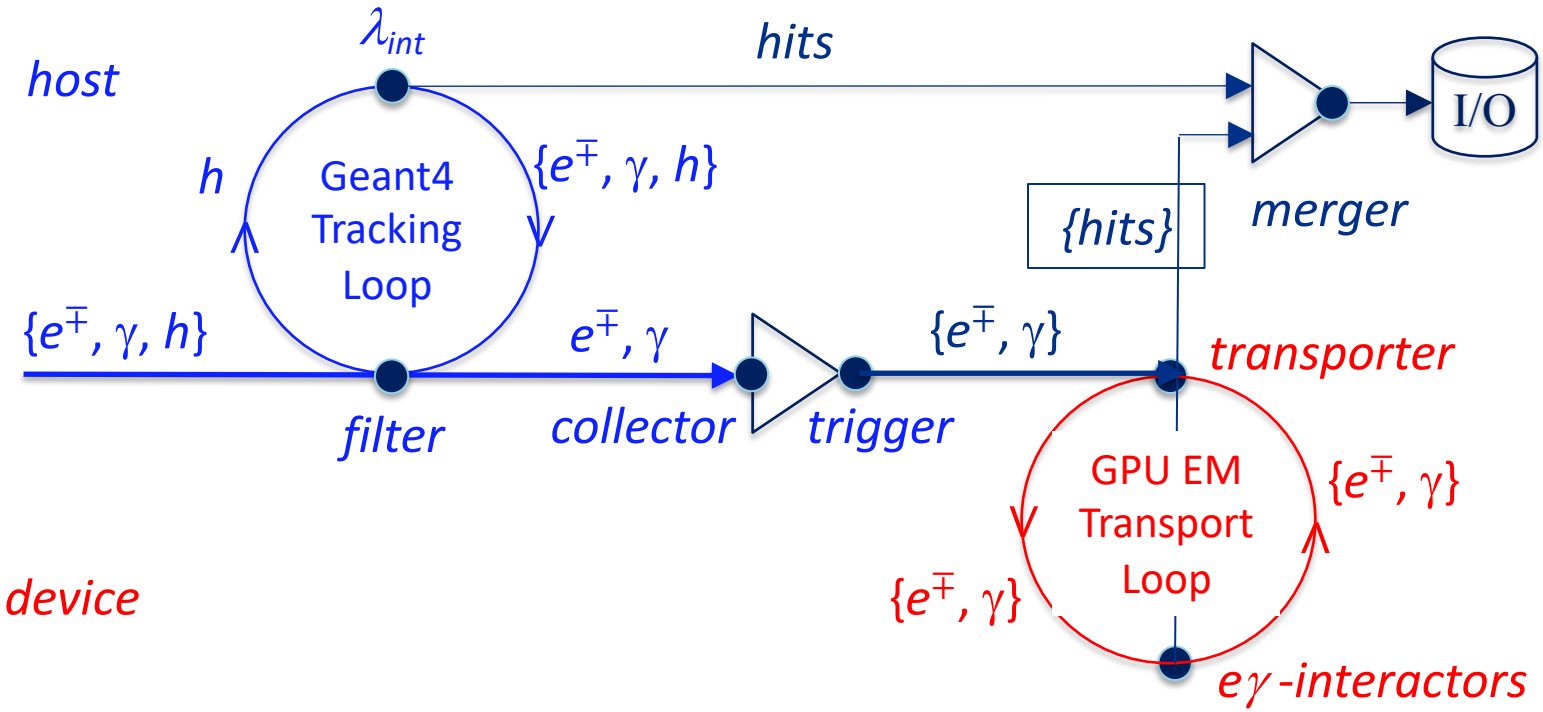


# Toward Sub-event Parallelism in Geant4

- Phase-I: Geant4 Kernel extension
  - Use *G4TaskRunManager* for sub-event parallelism
  - Implement *UserStackingAction* to sort tracks into sub-events
  - Implement *Merge()* method in *G4Event* if special merging treatment (efficient I/O for hit collection or scoring) is required
- Phase-II: Interfaces or integrations to task-oriented packages
  - Support specialized physics lists and/or detector construction dedicated to sub-tasks if needed: examples
    - G4HepEM (EM physics for HEP, designed to be compatible with GPUs)
    - VecGeom/Cuda or surface-oriented geometry models and navigators
  - Support generic interfaces to integrate external packages, e.g.,
    - Opticks/NVIDIA OptiX™ (Optical photon simulation)
    - Adept(CERN), Celeritas (US), etc. (EM particle transport)
  - Facilitates heterogenous computing on HPC facilities equipped with more diverse processing elements
    - Offloading a variety of single-purpose, optimized sub-tasks to accelerators

# Example: EM Particle Transport on GPU

- A hybrid workflow with selected tasks executed on GPUs
  - Hadronic particle simulation on the CPU host (event-level tasks)
  - EM particle ($e^{\mp}$, $\gamma$) transport on co-processors (offloading task)
  - Asynchronous I/O streams and Concurrent hit merging (I/O task)




  - Applicable workflow for many experiments: @LHC, LArTPC, EIC, etc.

# Tasking Code Example

- Direct submission to a task group with non-void return types

```
//-----------------------------------------------------------------------//
/*!
 * A tasking example with a non-void return type from user tasks.
 */
void DeviceManager::DoIt(id_type event_id, const G4Track& track)
{
    // Store this track to the stack
    collector->AddTrack(event_id, track);

    // Trigger offloading
    if(trigger())
    {
        // Submit work to a GPU task group
        TaskGroup<HitCollection> gpu_tasks(Merge, manager->GetThreadPool());
        gpu_tasks.exec(Propagate, collector->GetStack(), 1);
        gpu_tasks.exec(Propagate, collector->GetStack(), 2);

        // Merge hit collections from tasks
        auto result = gpu_tasks.join();

        // Clear the stack
        collector->Clear();
    }
}
```

  - Merge: a functor for merging HitCollection when join
  - Propagate: a kernel function (task) for propagating tracks on GPUs

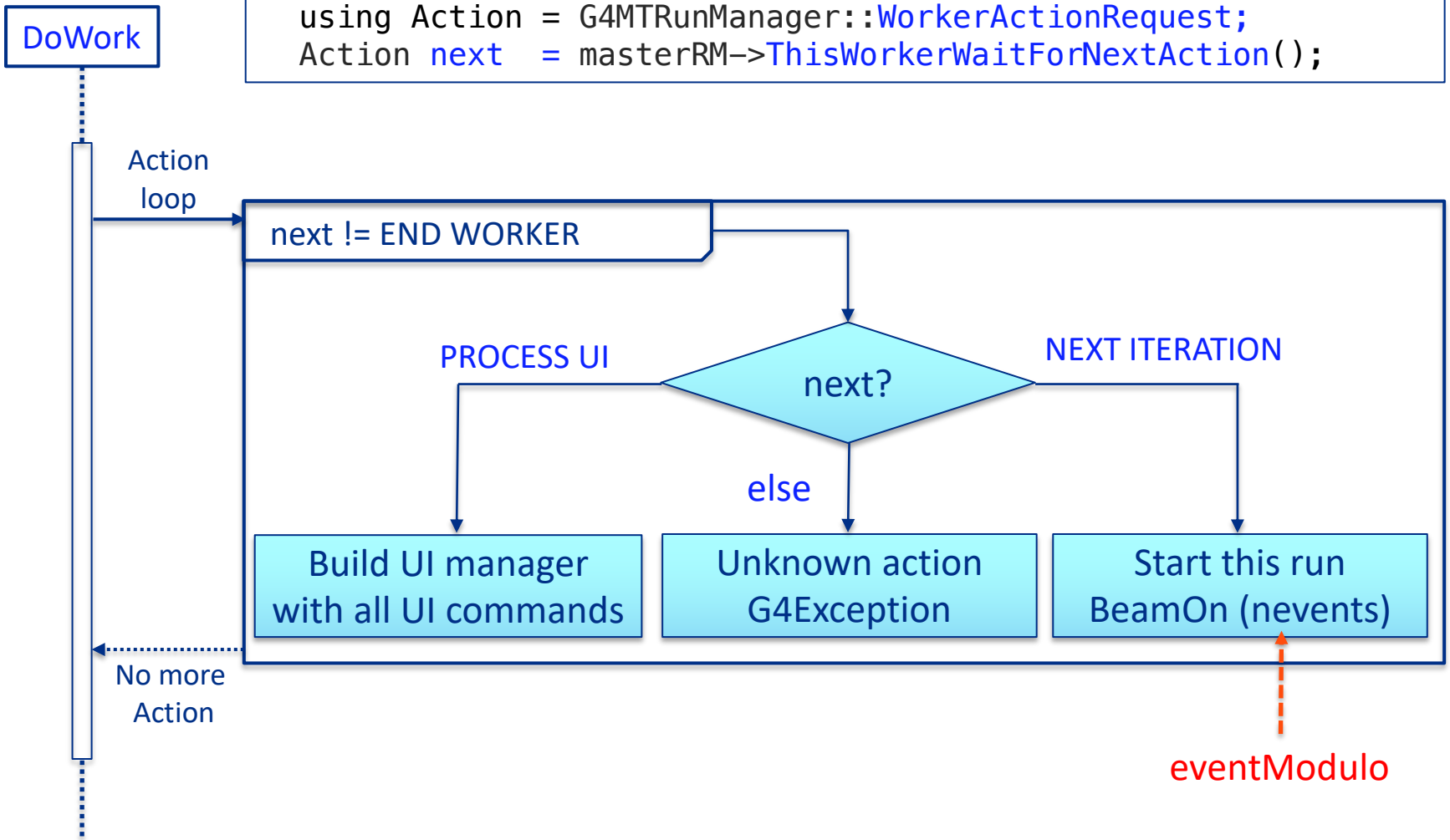
# Summary

- Geant4 has been evolving to satisfy HEP computing needs and will continue to evolve in future
- Geant4 supports a task-based framework (*G4Tasking*) suitable for multithreading, concurrent tasking and heterogeneous workflows
  - Event level parallelism
  - Sub-event level parallelism
  - (Track-level parallelism or user defined tasks)
- Some of examples of on-going HEP detection simulation projects using GPUs as parts of efforts within the Geant4 R&D task force
  - Offloading EM particle transport
    - G4HepEM/Adept (https://github.com/apt-sim/AdePT)
    - Celeritas/acceleritas (https://github.com/celeritas-project)
  - Optical photon simulation (Geant4/Opticks/NVIDIA OptiX™) (See Hans Wenzel's talk)

# Backup

# G4WorkerRunManager::DoWork

```
using Action = G4MTRunManager::WorkerActionRequest;
Action next  = masterRM->ThisWorkerWaitForNextAction();
```



# Thread Safety for Multithreading Applications

- Sharing of objects: a class (member methods) and data members
  - Thread-local or thread-shared class instances
  - Thread-shared or thread-local data field
  - Instance-shared or instance-local data field
- To be thread-safe for 8 different possible combinations

| thread-local class instances | thread-shared data field | thread-local data field |
|---|---|---|
| instance-shared data field | A<br>`static`<br>-safe only if const- | B<br>`static G4ThreadLocal`<br>-safe- |
| instance-local data field | C<br>probably nothing to do | D<br>nothing to do<br>-safe- |

| thread-shared class instances | thread-shared data field | thread-local data field |
|---|---|---|
| instance-shared data field | E<br>`static`<br>-safe only if const- | F<br>`static G4ThreadLocal`<br>-safe- |
| instance-local data field | G<br>nothing to do<br>-safe only if const- | H<br>split-class mechanism or G4Cache<br>-safe- |

  - A, E → static const
  - B, F → static G4ThreadLocal
  - C, D→ nothing to do (and G → const)
  - H → split class or G4Cache

# G4ThreadLocal

- Objects that are specific to each event and belong to each thread 'by design': G4Event, Hits Collections,  container of Primary Tracks
- Instances of key classes designed to be per-thread, e.g., G4WorkerRunManager
- Instance of others created in above classes remains thread-local: G4EventManager, G4TrackingManager, G4SteppingManager, G4SDManager, etc.
- Some classes must have a separate object (instance) for each thread, as they are not thread-aware: Processes, G4Navigator, etc
- For many classes, each thread has an instance (copy) of the relevant object as it was created by a per-thread manager: G4Track, G4Step, G4VSensitiveDetector
- Thread-local objects are instantiated and initialized at the first *BeamOn*

# Split Class

- Some classes must have a duality: they must maintain some ‘constant’ attributes which are shared by all threads, but also provide data which is thread-specific: pointers to per-thread objects, information which can be updated when a track is propagated by a thread.
- These are the “split” classes, such as physical volume which can have a different replica number (replicas, parameterized volumes, divisions), a logical volume that has a different G4FieldManager and can have a different material (in nested parameterization), etc.
- To enable collaboration with these, we require that user code follows strict guidelines:
  - UserActions must be initialized by the G4VUserActionInitialization (static - one instance)
    - Creating separate objects for a thread when it is called by that thread
  - UserDetectorConstruction must have a method that creates Sensitive Detectors and Field objects for a thread