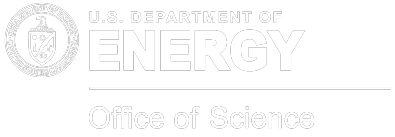
Multithreading

Andrea Dotti

April 19th, 2015

Geant4 tutorial @ M&C+SNA+MC 2015



Contents

SLAC

Basics of multithreading

Event-level parallelism

How to install/configure MT mode

Race condition

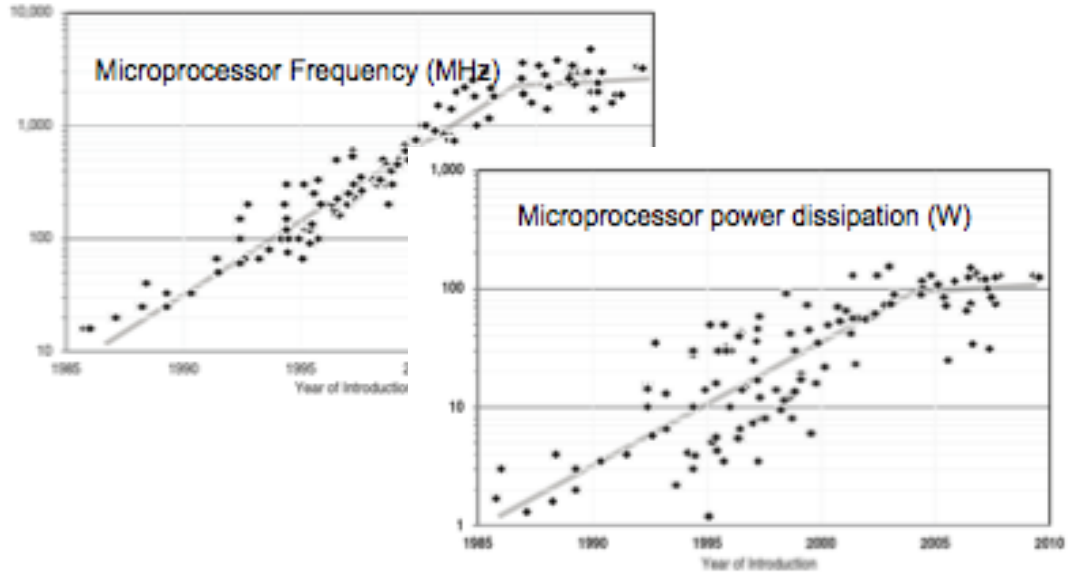
Mutex and thread local storage

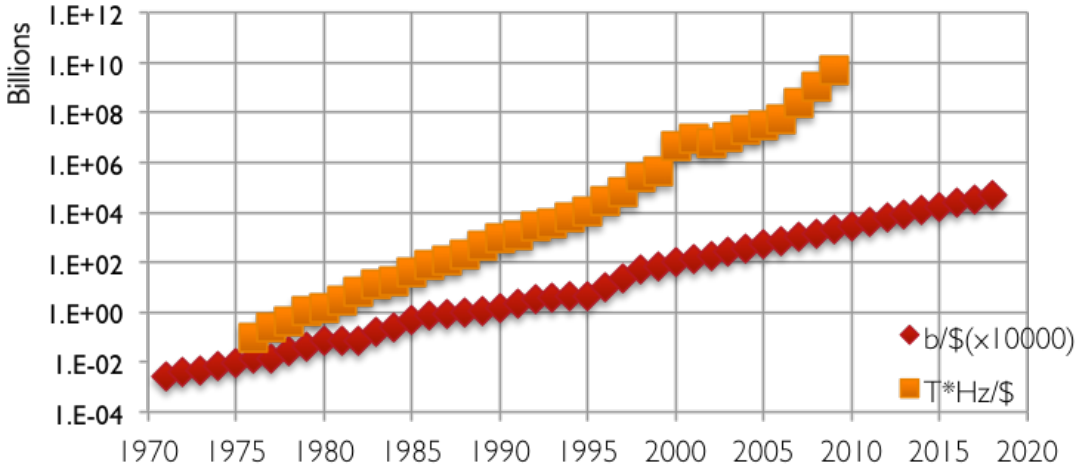
How to migrate user's code to MT

Intel Xeon Phi

The challenges of many-core era

SLAC





- Increase frequency of CPU causes **increase of power needs**
- Reached plateau around 2005
- No more increase in CPU frequency
- However number of transistors/$ you can buy continues to grow
- Multi/May-core era
- Note: quantity memory you can buy with same $ scales slower
- **Expect:**
- Many core (double/2yrs?)
- Single core performance will not increase as we were used to
- Less memory/core
- New software models need to take these into account: increase parallelism

CPU Clock Frequecy land usage: The Future of Computing Performance: Game Over or Next Level?
DRAM cost: Data from 1971-2000:VLSI Research Inc. Data from 2001-2002: ITRS, 2002 Update, Table 7a, Cost-Near-Term Years, p. 172. Data from 2003-2018: ITRS, 2004 Update, Tables 7a and 7b, Cost-Near-Term Years, pp. 20-21.
CPU cost: Data from 1976-1999: E. R. Berndt, E. R. Dulberger, and N. J. Rappaport, "Price and Quality of Desktop and Mobile Personal Computers: A Quarter Century of History," July 17, 2000, ;Data from 2001-2016: ITRS, 2002 Update, On-Chip Local Clock in Table 4c: Performance and Package Chips: Frequency On-Chip Wiring Levels -- Near-Term Years, p. 167. ;
Average transistor price: Intel and Dataquest reports (December 2002), see Gordon E. Moore, "Our Revolution,"

In Brief

SLAC

- Modern CPU architectures: need to introduce **parallelism**
- 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)
- **Events are independent**: each event can be simulated separately
- Multi-threading for event level parallelism is the natural choice

Geant4 Multi Threading capabilities

SLAC

- MT code integrated into G4
- Public release
- All functionalities ported to MT

- Proof of principle
- Identify objects to be shared
- First testing

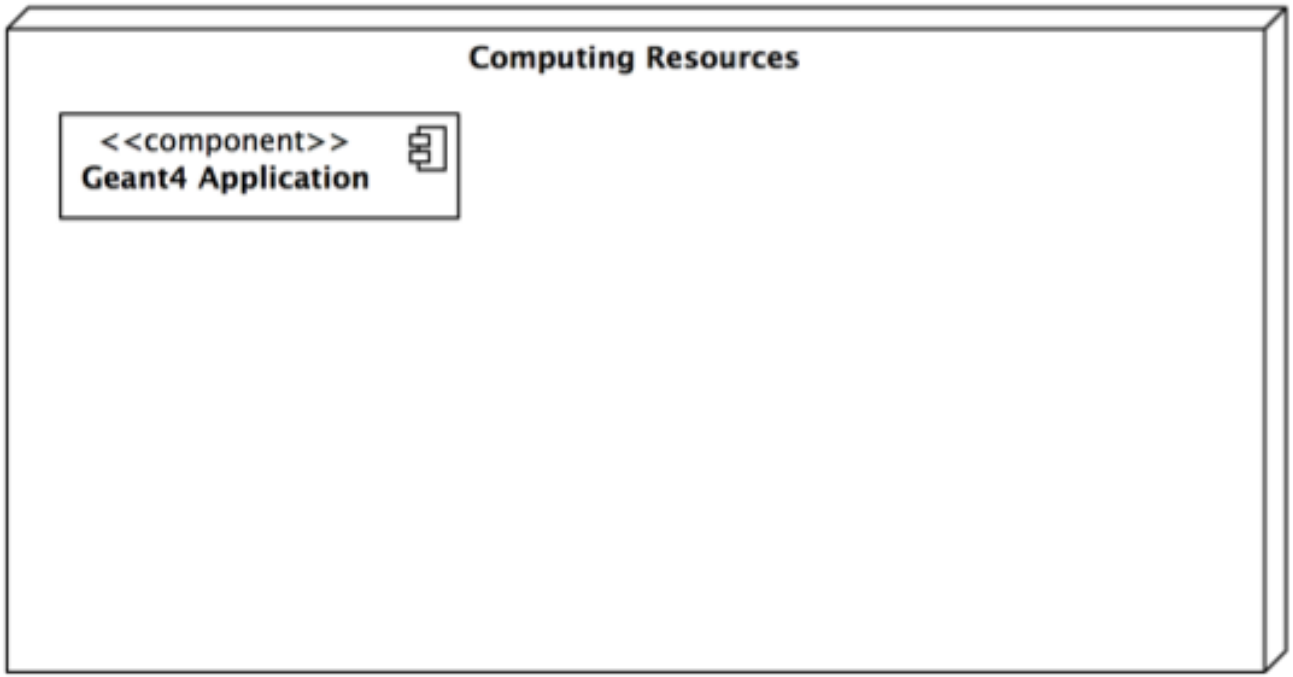
- API re-design
- Example migration
- Further testing
- First optimizations

- Further Refinements
- Focus on further performance improvements

What is a thread?

SLAC

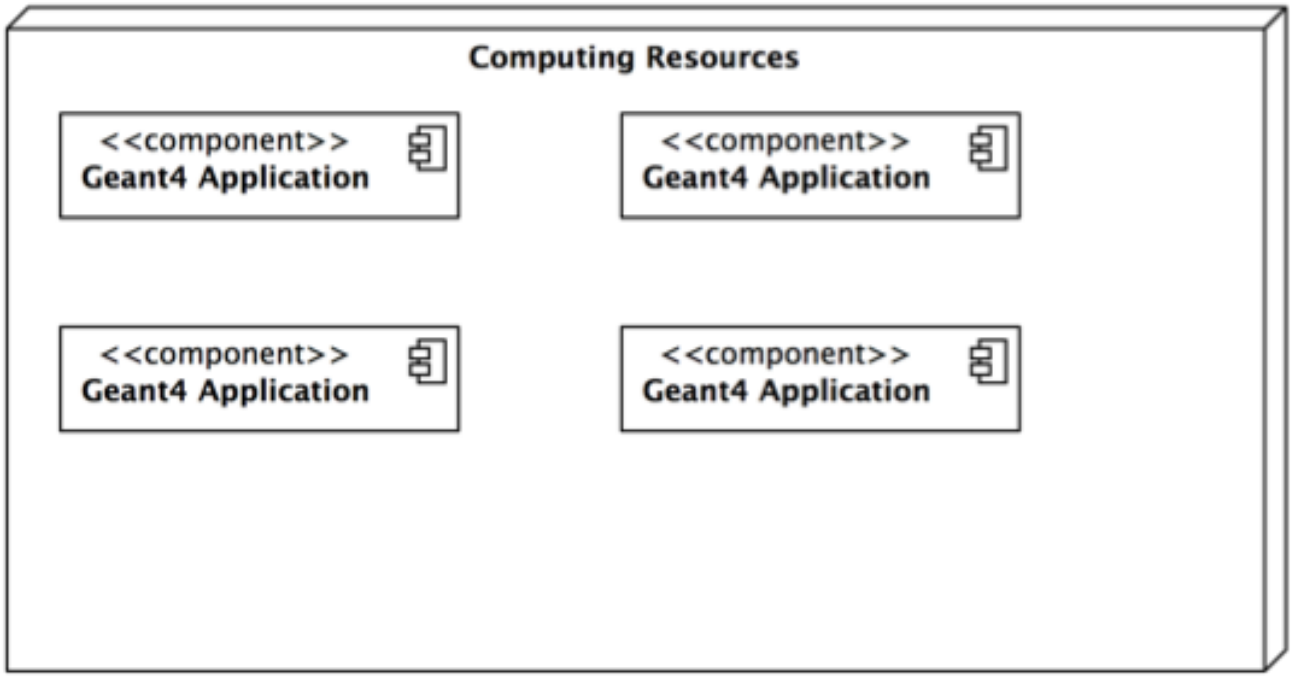
Sequential application



What is a thread?

SLAC

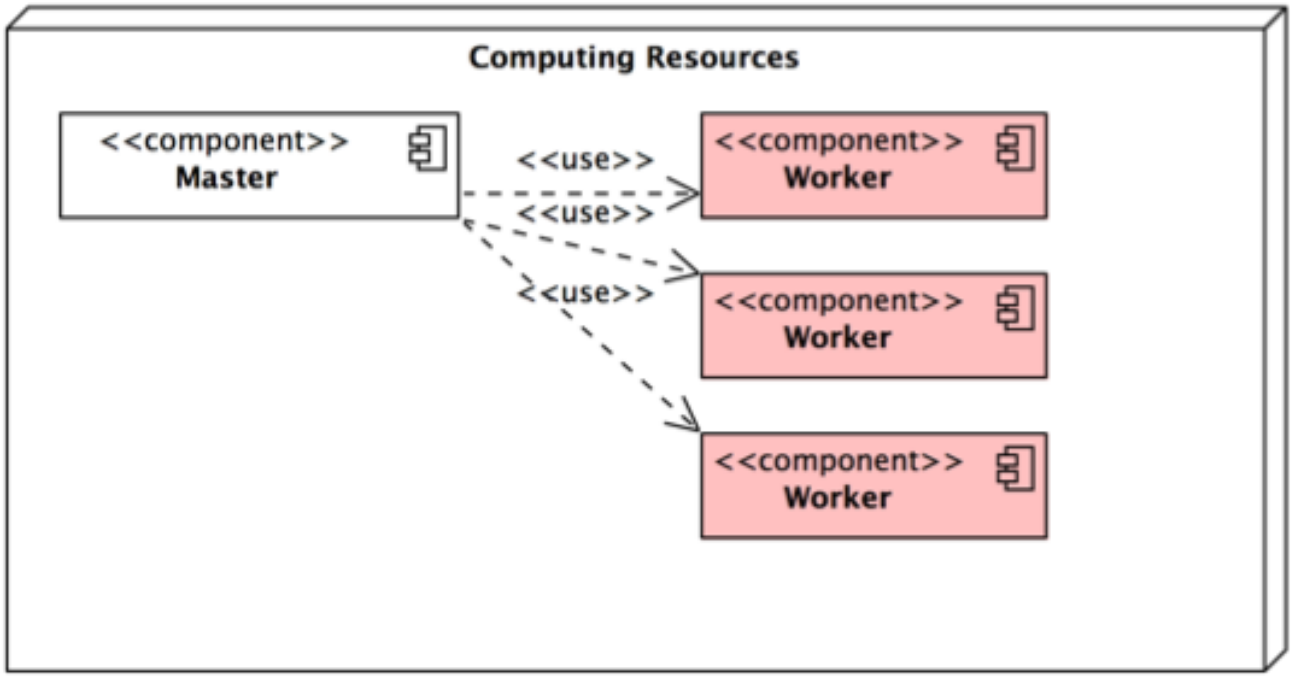
Sequential application: start N (cores/CPUs) copies of application if fits in memory



What is a thread?

SLAC

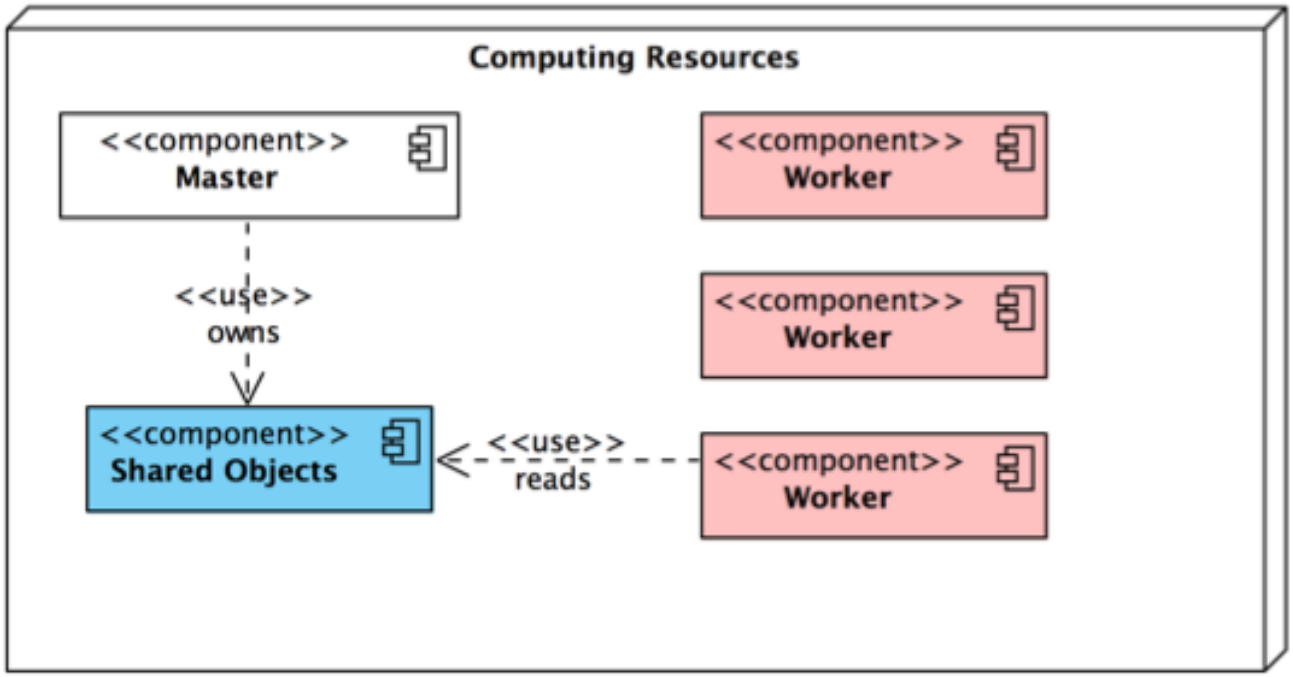
MT Application: single application starts threads. For G4: application (master) controls workers that do simulation, no memory sharing now, each worker is a copy of the application



What is a thread?

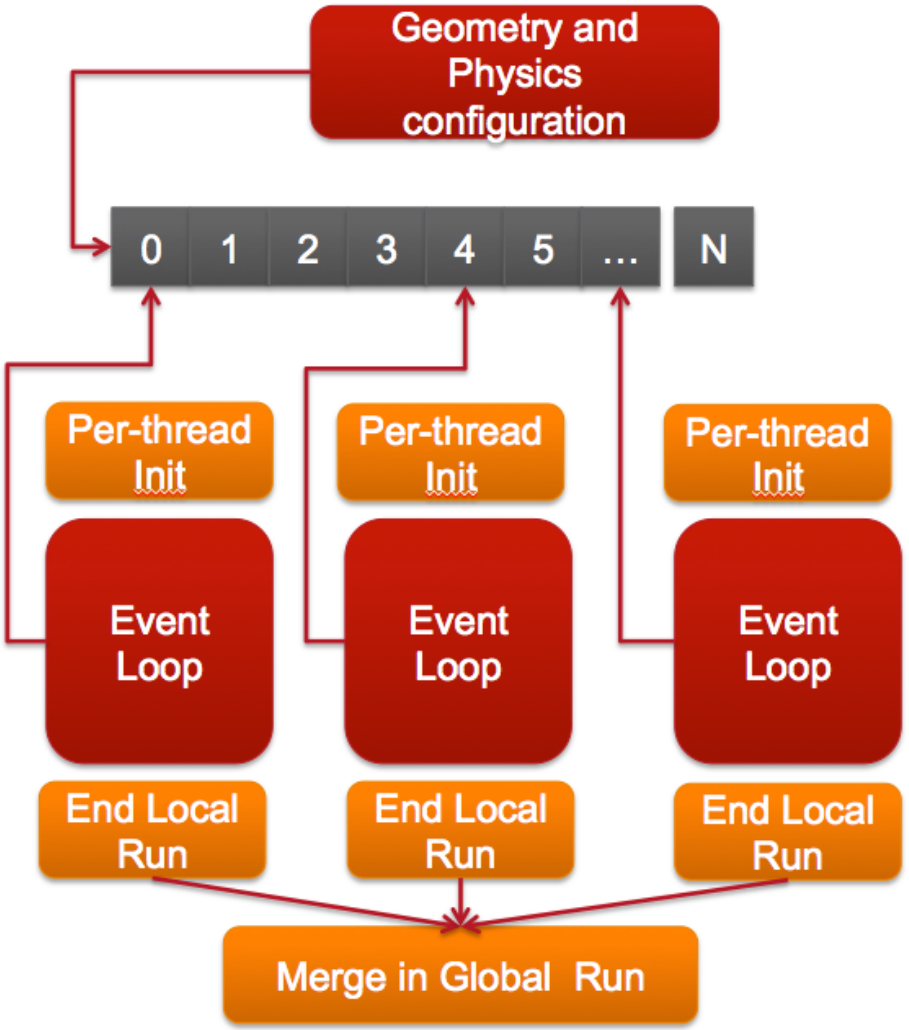
SLAC

Memory reduction: introduce shared objects, memory of N threads is less than memory used by N copies of application



General Design

SLAC



Per-event seeds pre-prepared in a "queue"

ete for next
cesses (new
in ref-08)

Command line scoring and G4tools automatically merge results from threads

Simplified Master / Worker Model

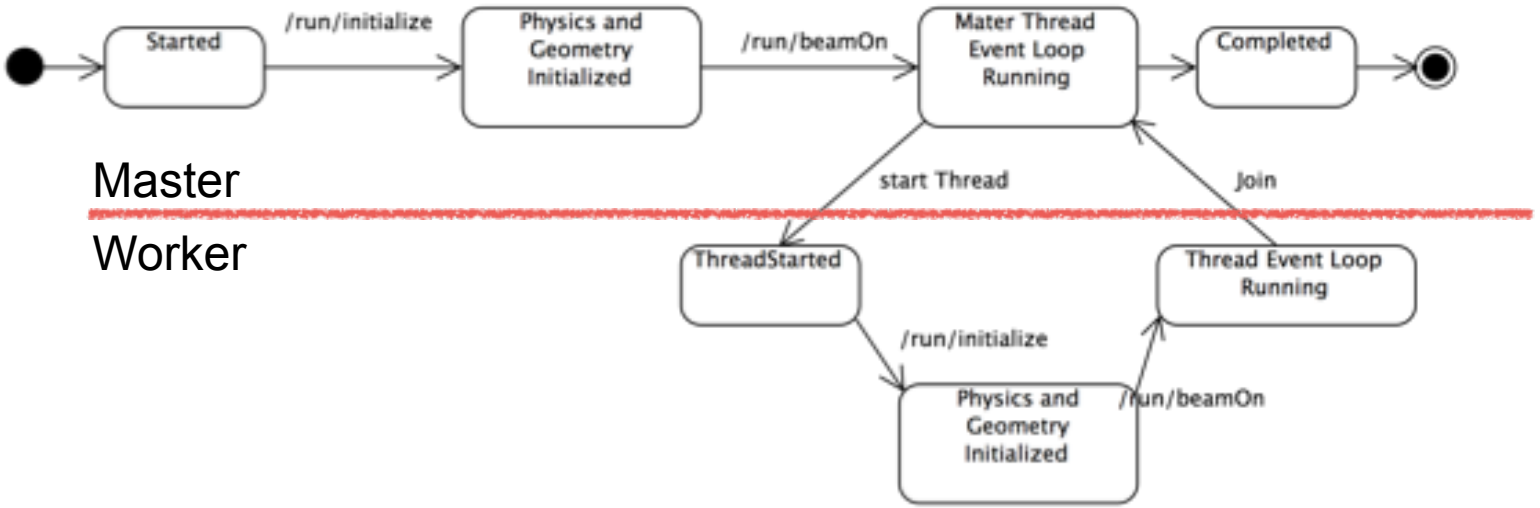
SLAC

- A G4 (with MT) application can be seen as simple finite state machine

Simplified Master / Worker Model

SLAC

- A G4 (with MT) application 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 ? Private?

SLAC

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

Shared by all threads

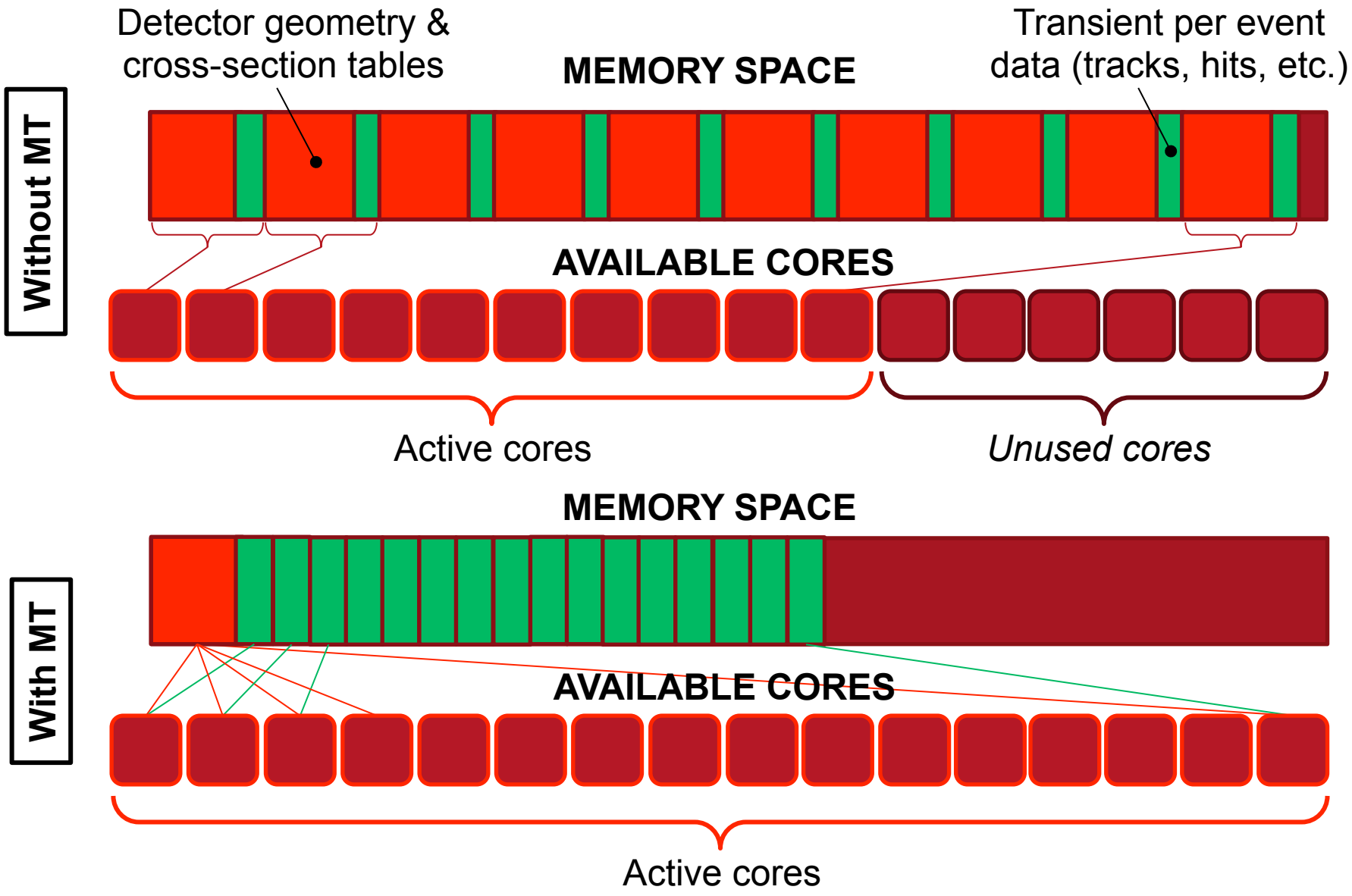
: stable during the event loop

- 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


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

Shared ? Thread-local?

SLAC

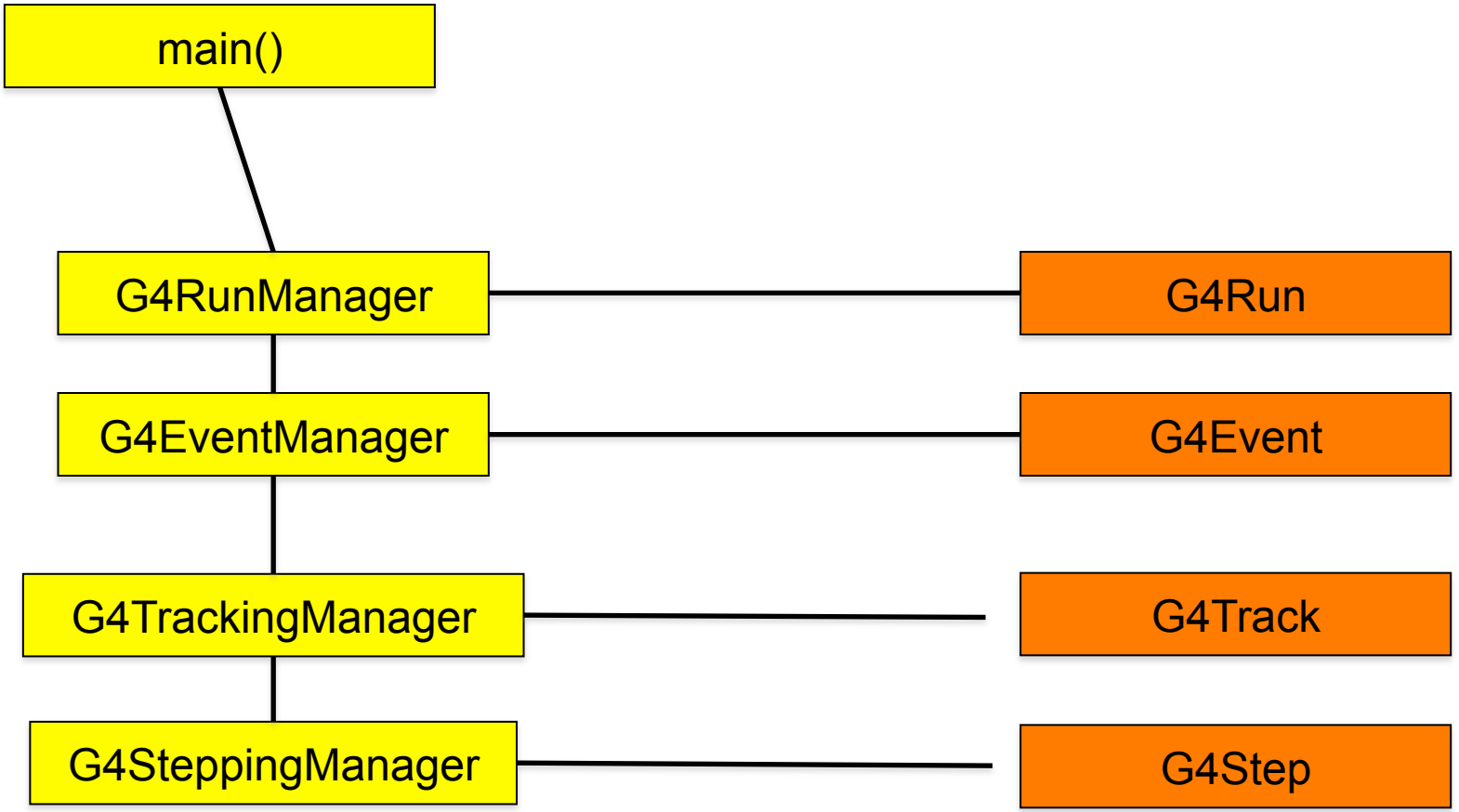
In general, geometry and physics tables are shared, while event, track, step, trajectory, hits, etc., as well as several Geant4 manager classes such as EevntManager, TrackingManager, SteppingManager, TransportationManager, FieldManager, Navigator, SensitiveDetectorManager, etc. are thread-local.

Among the user classes, user initialization classes (G4VUserDetectorConstruction, G4VUserPhysicsList and newly introduced G4VUserActionInitialization) are shared, while all user action classes and sensitive detector classes are thread-local.

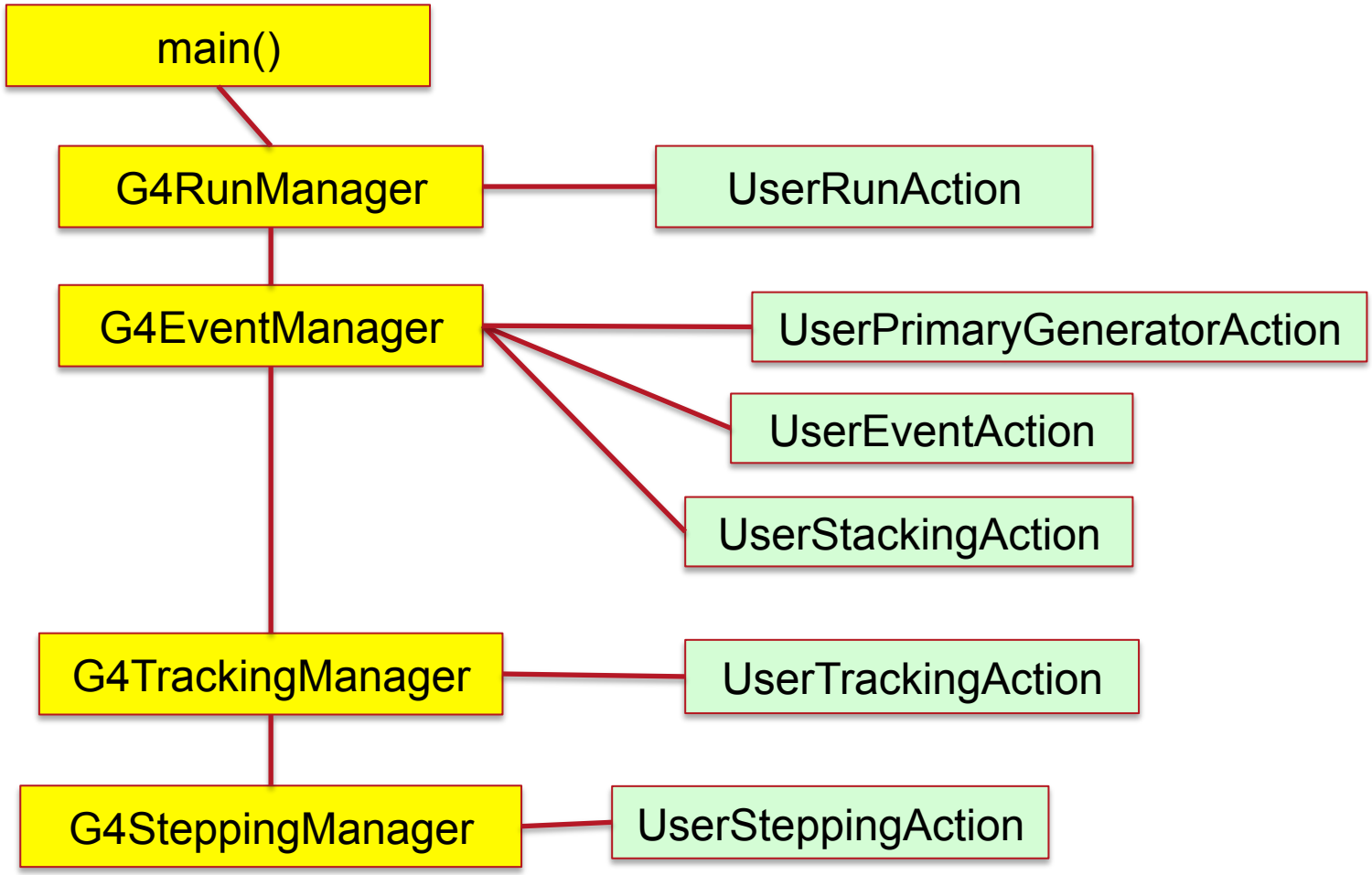
- It is not straightforward (and thus not recommended) to access from a shared class object to a thread-local object, e.g. from detector construction to stepping action.
- Please note that thread-local objects are instantiated and initialized at the first *BeamOn*.

To avoid potential errors, it is advised to always keep in mind which class is shared and which class is thread-local.

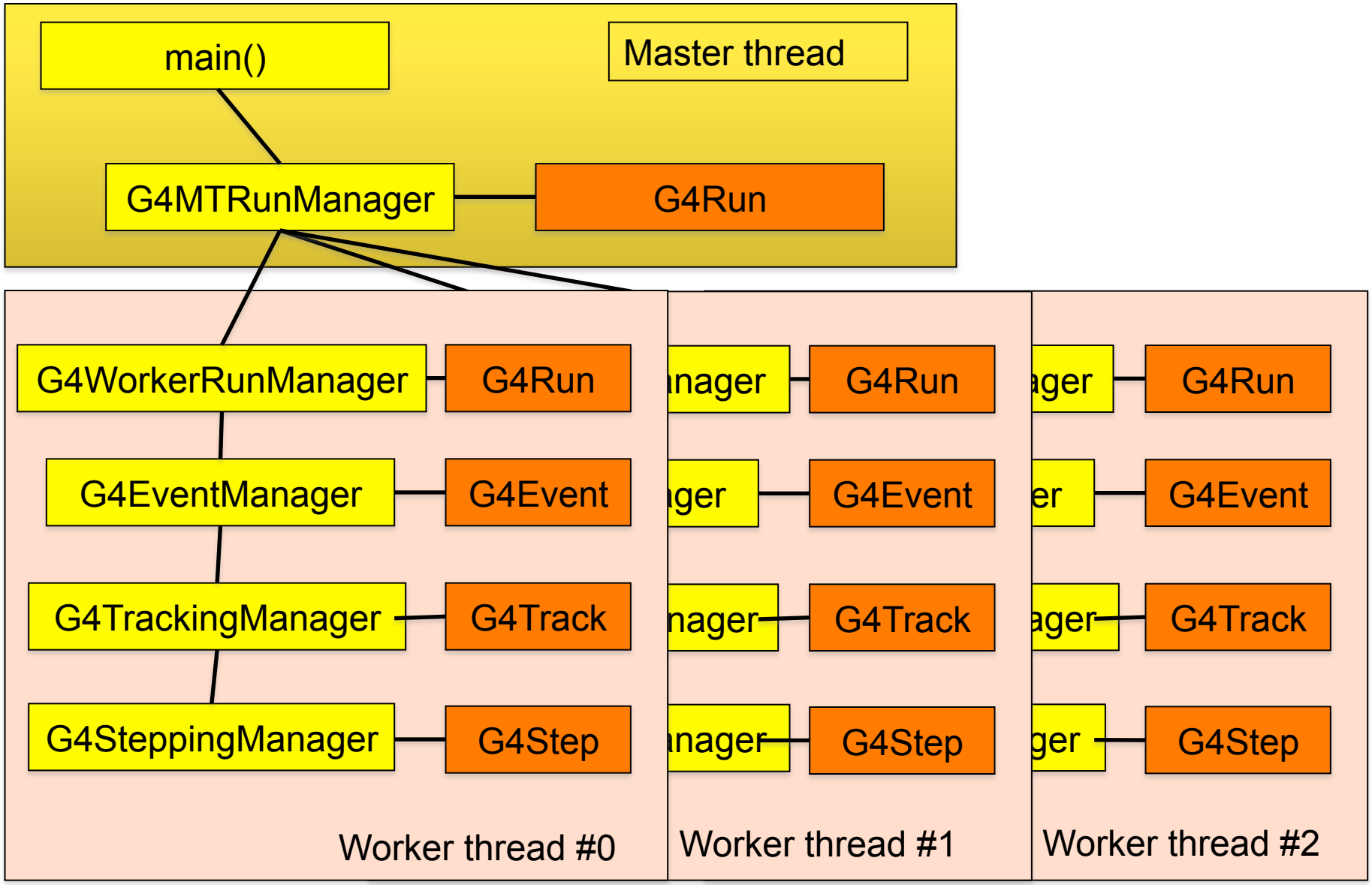
Sequential mode



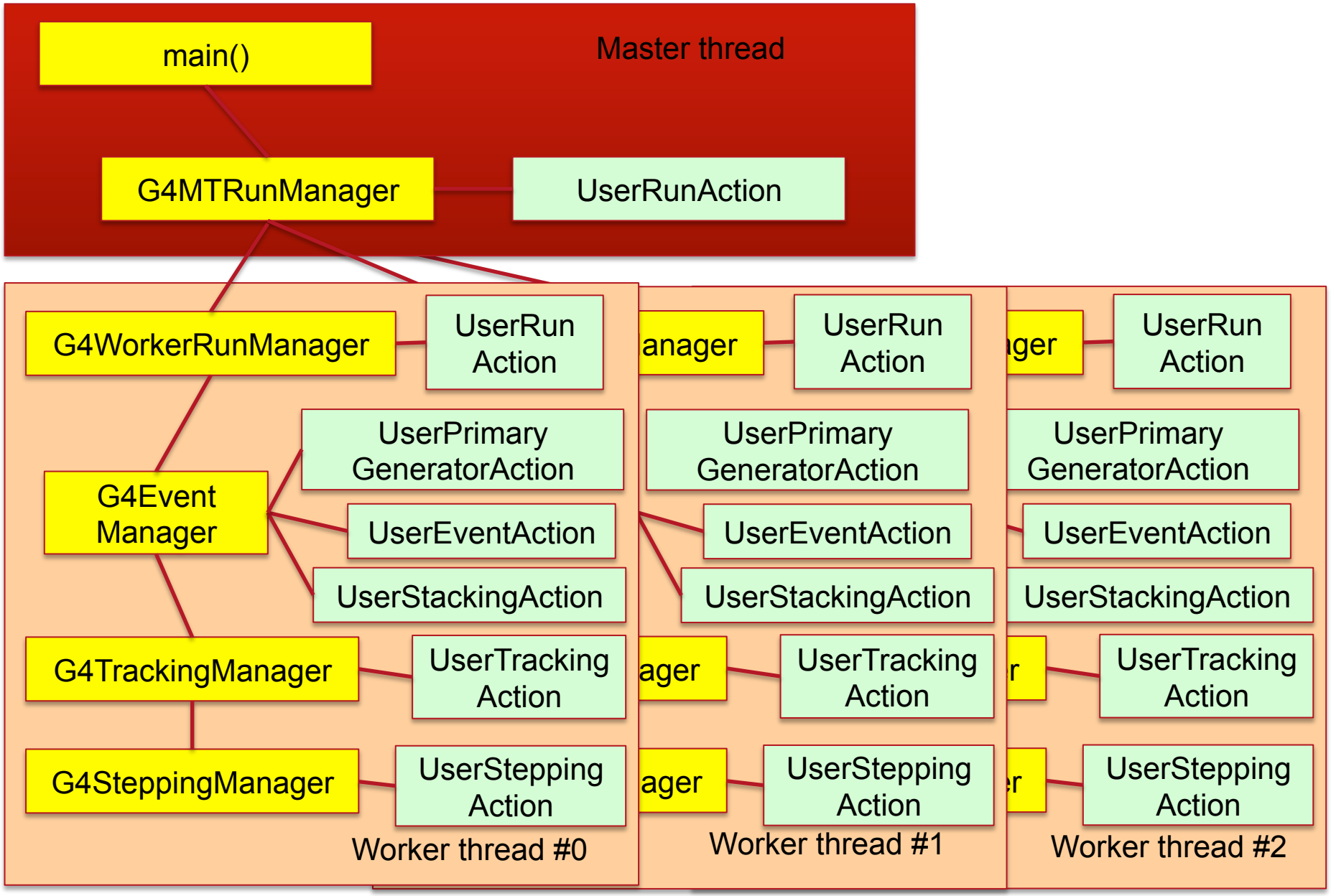
Sequential mode



Multi-threaded mode



Multi-threaded mode



How to configure Geant4 for MT

SLAC

- `cmake -DGEANT4_BUILD_MULTITHREADED=ON […]`
- Requires "recent" compiler that supports ThreadLocalStorage technology (to be discussed Thursday) and pthread library installed (usually pre-installed on POSIX systems)
- Check cmake output for:
  - `-- Performing Test HAVE_TLS`
  - `-- Performing Test HAVE_TLS - Success`
- If it complains then your compiler is too old, sorry…
- Mac OS X, you need to use clang>=3.0 (not gcc!). On Mac OS X 10.7:

```
cmake -DCMAKE_CXX_COMPILER=clang++ -DCMAKE_C_COMPILER=clang \
      --DGEANT4_BUILD_MULTITHREADED=ON […]
```

- Sorry no Windows support!
- Compile as usual

MT related UI commands

SLAC

`/run/numberOfThreads [n]` : Specify number of threads

or `/run/useMaximumLogicalCores` : Use the maximum number of cores

`/control/cout/setCoutFile [filename]` : Sends G4cout stream to a per-thread file. Use "***Screen***" to reset to screen

`/control/cout/setCerrFile [filename]` : As previous but for G4cerr

Advanced commands:

`/control/cout/useBuffer [true|false]` : Send G4cout/G4cerr to a per-thread buffer that will printed at the end of the job

`/control/cout/prefixString [string]` : Add an per-thread identifier to each output line from threads, the thread id is appended to this prefix (default: G4WTn)

`/control/cout/ignoreThreadsExcept [id]` : Show output only from thread "id"

`/run/pinAffinity [n]`: Set thread affinity (lock threads to core), may increase performances in some cases

`/run/eventModulo [n] [s]`: Set how many events to send to threads in one request and how often re-seed thread RNG engine, defaults work vast majority cases, in special cases (e.g. extremely small and fast events) tweaking this parameters can increase performances (warning: playing with seeding algorithm can break strong reproducibility)

Setting the number of threads

SLAC

- Default the number of threads: 2
  - Use `/run/numberOfThreads` or `G4MTRunManager::SetNumberOfThreads()`
- `G4Threading::G4GetNumberOfCores()` returns the number of logical cores of your machine
- Currently number of threads **cannot be changed** after `/run/initialize` (C++ call to: `G4RunManager::Initialize()`)
- You can overwrite your application behavior and UI commands setting the (shell) environment variables `G4FORCENUMBEROFTHREADS=...` before starting the application (the special keyword `max` can be used to use all system cores)

Thread safety a simple example

SLAC

Consider a function that reads and writes a shared resource (a global variable in this example).

```
int aShredVariable = 1;

int SomeFunction()
{
 int result = 0;
 if ( aShredVariable > 0 )
 {
   result = aSharedVariable;
   aSharedVariable = -1;
 } else {
   doSomethingElse();
   aSharedVariable = 1;
 }
 return result;
}
```

Thread safety a simple example

SLAC

Now consider two threads that execute at the same time the function. Concurrent access to the shared resource

```
double aSharedVariable = 1;
```

```
int SomeFunction()
{
 int result = 0;
 if ( aShredVariable > 0 )
 {
   result = aSharedVariable;
   aSharedVariable = -1;
 } else {
   doSomethingElse();
   aSharedVariable = 1;
 }
 return result;
}
```

T1

```
int SomeFunction()
{
 int result = 0;
 if ( aShredVariable > 0 )
 {
   result = aSharedVariable;
   aSharedVariable = -1;
 } else {
   doSomethingElse();
   aSharedVariable = 1;
 }
 return result;
}
```

T2

Thread safety a simple example

SLAC

Given *result* is a local variable, that exists in each thread separately, it is not a problem. Now T1 arrives **here**.

```
double aSharedVariable = 1;
```

```
int SomeFunction()
{
 int result = 0;
 if ( aShredVariable > 0 )
 {
   result = aSharedVariable;
   aSharedVariable = -1;
 } else {
   doSomethingElse();
   aSharedVariable = 1;
 }
 return result;
}
```

T1

```
int SomeFunction()
{
 int result = 0;
 if ( aShredVariable > 0 )
 {
   result = aSharedVariable;
   aSharedVariable = -1;
 } else {
   doSomethingElse();
   aSharedVariable = 1;
 }
 return result;
}
```

T2

Thread safety a simple example

SLAC

Now T2 starts and arrives **here**, given the value of aSharedVariable depends on whether T1 has already changed it or not, what is the expected behavior? what is happening?

```
double aSharedVariable = 1;
```

```
int SomeFunction()
{
 int result = 0;
 if ( aShredVariable > 0 )
 {
   result = aSharedVariable;
   aSharedVariable = -1;
 } else {
   doSomethingElse();
   aSharedVariable = 1;
 }
 return result;
}
```

T1

```
int SomeFunction()
{
 int result = 0;
 if ( aShredVariable > 0 )
 {
   result = aSharedVariable;
   aSharedVariable = -1;
 } else {
   doSomethingElse();
   aSharedVariable = 1;
 }
 return result;
}
```

T2

Thread safety a simple example

SLAC

Use mutex / locks to create a barrier. T2 will not start until T1 reaches UnLock

Significantly reduces performances (general rule in G4, not allowed in methods called during the event loop)

```
int aSharedVariable = 1;

int SomeFunction() {
 int result = 0;
 Lock(&mutex);
 if ( aShredVariable > 0 ) {
   result = aSharedVariable;
   aSharedVariable = -1;
 } else {
   doSomethingElse();
   aSharedVariable = 1;
 }
 Unlock(&mutex);
 return result;
}
```

T1

```
int SomeFunction() {
 int result = 0;
 Lock(&mutex);
 if ( aShredVariable > 0 ) {
   result = aSharedVariable;
   aSharedVariable = -1;
 } else {
   doSomethingElse();
   aSharedVariable = 1;
 }
 Unlock(&mutex);
 return result;
}
```

T2

http://en.wikipedia.org/wiki/Lock_(computer_science)

Thread safety a simple example

SLAC

- Do we really need to share *aSahredVariable*?
  - Simple way to "transform" your code (but very small cpu penalty, no memory usage reduction)
- General rule in G4: do not use Mutex lock unless really necessary

```
double __thread
aSharedVariable;

int SomeFunction() {
 int result = 0;
 if ( aShredVariable > 0 ) {
   result = aSharedVariable;
   aSharedVariable = -1;
 } else {
   doSomethingElse();
   aSharedVariable = 1;
 }
 return result;
}
```

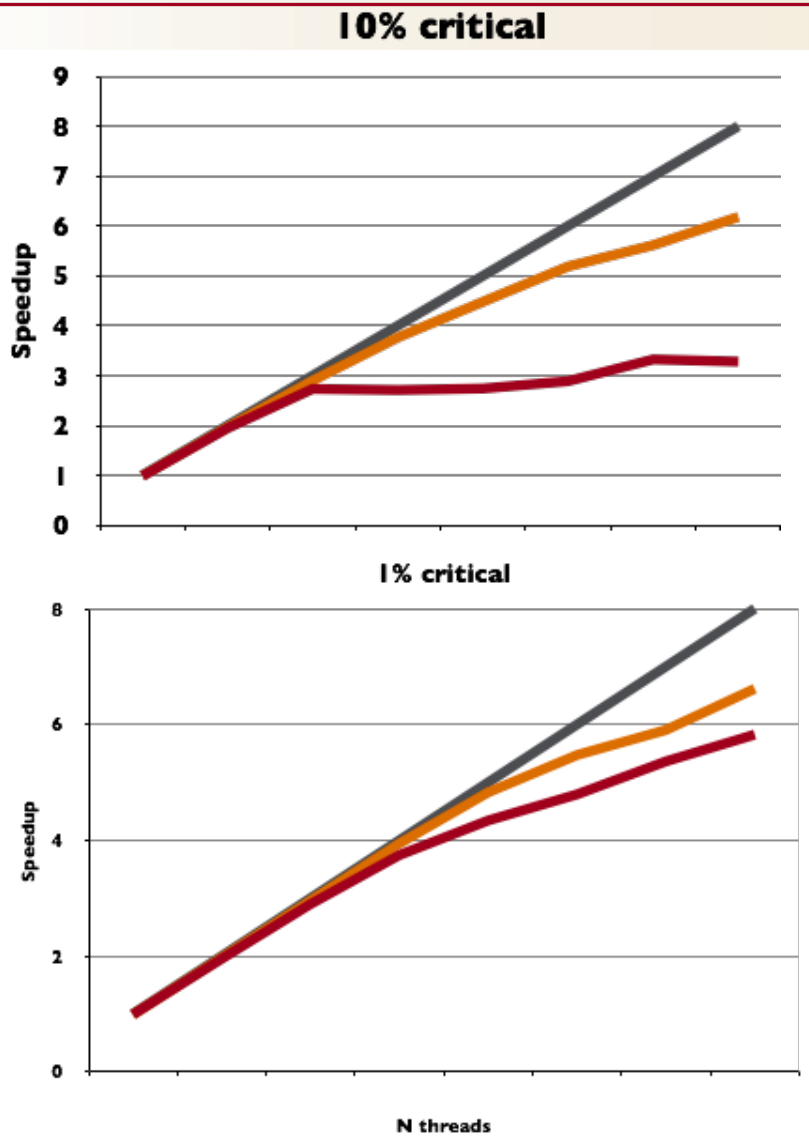
T1

```
double __thread
aSharedVariable;

int SomeFunction() {
 int result = 0;
 if ( aShredVariable > 0 ) {
   result = aSharedVariable;
   aSharedVariable = -1;
 } else {
   doSomethingElse();
   aSharedVariable = 1;
 }
 return result;
}
```

T2

Thread Local Storage



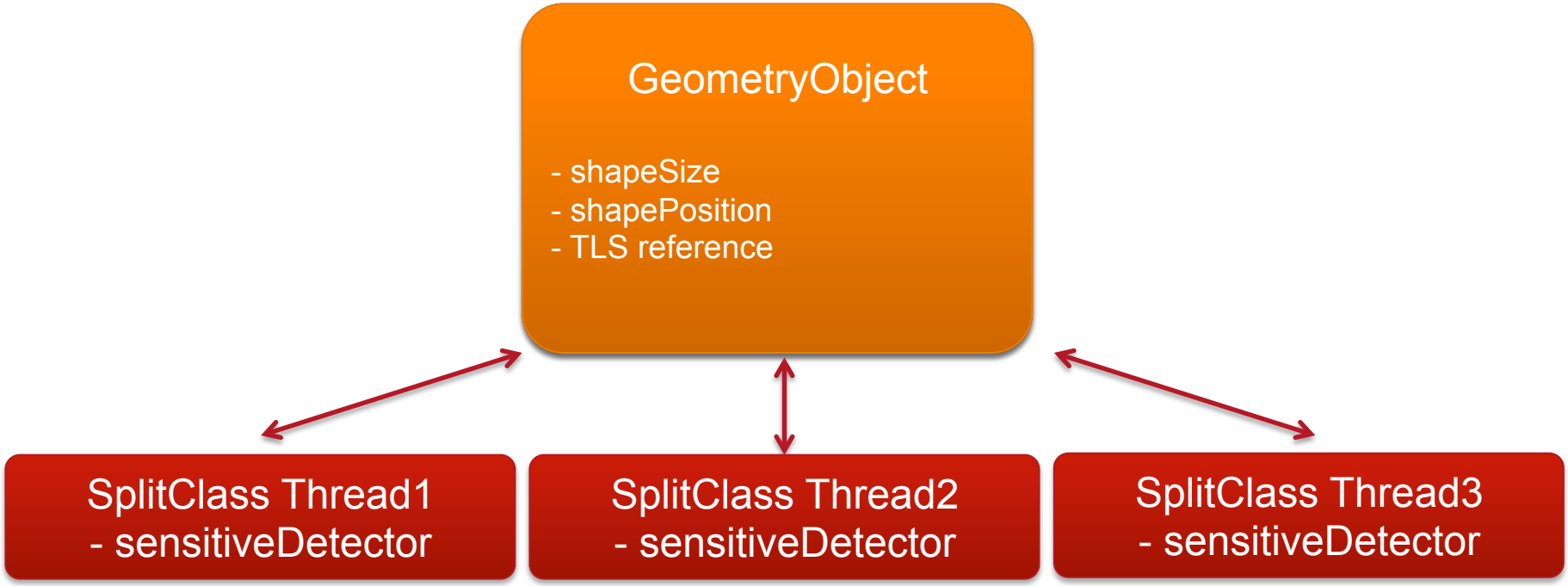
NB: results obtained on toy application, not real G4

- Each (parallel) program has sequential components
- **Protect access to concurrent resources**
- Simplest solution: use mutex/lock
- TLS: each thread has its own object (no need to lock)
  - **Supported by all modern compilers**
  - "just" add `__thread` to variables
  - `__thread int value = 1;`
  - Improved support in C++11 standard
- Drawback: increased memory usage and small cpu penalty (currently 1%), only simple data types for static/global variables can be made TLS

Basic design choice

SLAC

- Thread-safety implemented via **Thread Local Storage**
- "Split-class" mechanism: reduce memory consumption
  - Read-only part of most memory consuming objects shared between thread
  - Geometry, Physics Tables
  - Rest is thread-private

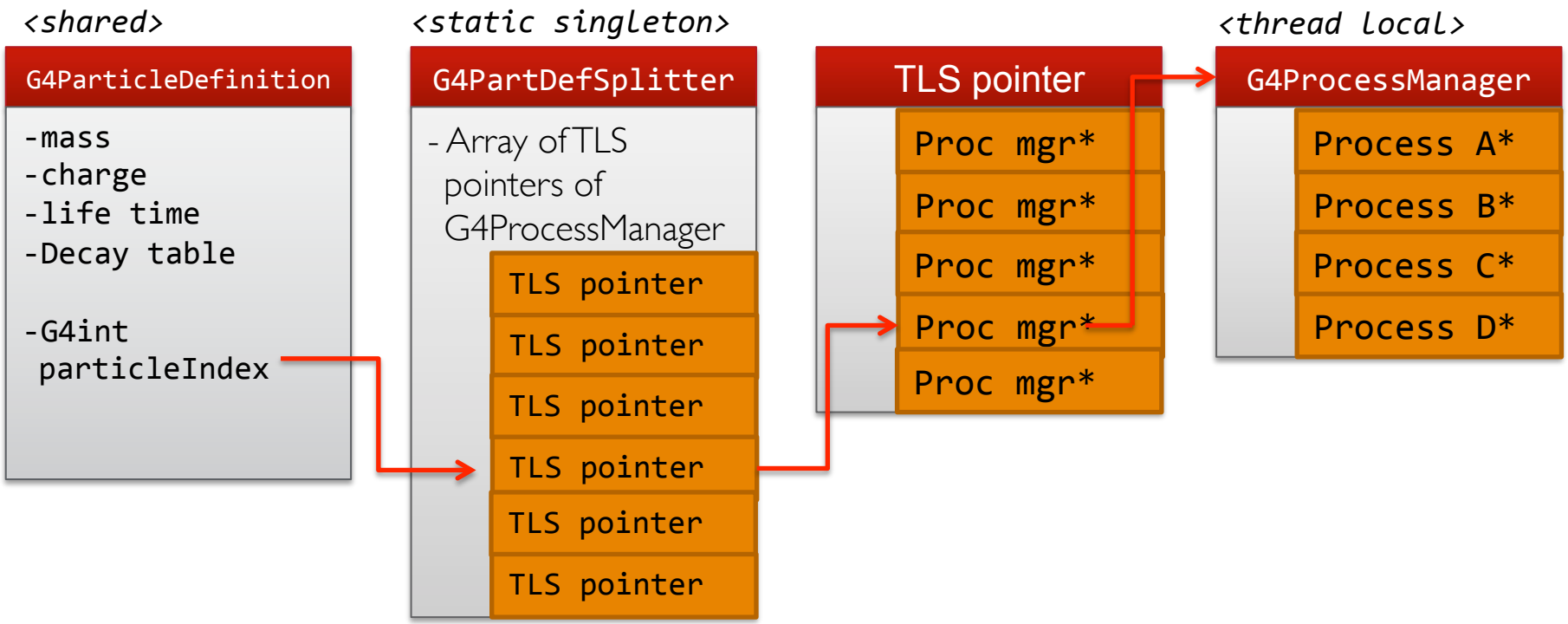


Split class – case of particle definition

SLAC

In Geant4, each particle type has its own dedicated object of G4ParticleDefinition class.

- Static quantities : mass, charge, life time, decay channels, etc.,
  - To be shared by all threads.
- Dedicated object of G4ProcessManager : list of physics processes this particular kind of particle undertakes.
  - Physics process object must be thread-local.



Locks and Mutex

SLAC

To add a lock mechanism (remember: will spoil performances but may be needed with non thread-safe code):

```
#include "G4AutoLock.hh"
namespace {
   G4Mutex aMutex = G4MUTEX_INITIALIZER;
}

void myfunction()
{
  //enter critical section
  G4AutoLock l(&aMutex);
  //will automatically unlock when out of scope
  return;
}
```

Memory handling

SLAC

Instead of using `__thread` keyword, use `G4ThreadLocal`. E.g.

```
static G4ThreadLocal G4double aValue = 0.;
```

Few classes/utilities have been created to help handling of objects.

Described in Chapter 2.14 of Users's Guide For Toolkit Developers. In brief:

- `G4Cache` : Allows to create a thread-local variable in shared class
- `G4ThreadLocalSingleton` : for thread-private "singleton" pattern
- `G4AutoDelete` : automatically delete thread objects at the end of the job

http://geant4.web.cern.ch/geant4/UserDocumentation/UsersGuides/ForToolkitDeveloper/html/ch02s14.html

User hooks

SLAC

- In special cases you may need to customize some aspects of the Thread behavior (only for experts)
- You can:
  - Build your class inheriting from `G4UserWorkerIntialization` allows to add user code during thread initialization stages (see .hh for details).
  - The threading model is handled in `G4UserWorkerThreadInitialization`, sub-class to customize (how threads start, how they join, etc). See .hh for details

- Instantiate in main (as all other initializations) and add them to kernel via `G4MTRunManager::SetUserInitialiation( … )`

User's code migration

SLAC

If you have a running code with version 9.6 and you want to stick to sequential mode, you do not need to migrate. It should run with version 10.0.

- Except for a few obsolete interfaces that you had already seen warning messages in v9.6.

Migration of user's code to multi-threading mode of Geant4 version 10.0 should be fairly easy and straightforward.

- Migration guide is available.
- Geant4 users guides are updated with multi-threading features.
- Many examples have been migrated to multi-threading.
- Geant4 tutorials based on version 10.0 has already started.

G4MTRunManager collects run objects from worker threads and "reduces".

Toughest part of the migration is making user's code thread-safe.

- It is always a good idea to clearly identify which class objects are thread-local.

Every file I/O for local thread is a challenge

- Input : primary events : examples are offered in the migration guide.
- Output : event-by-event hits, trajectories, histograms

Five steps to migrate

SLAC

Assuming that you have a running code built on Geant4 v9.6, there are only five simple steps to migrate your code to the multithreaded 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

https://twiki.cern.ch/twiki/bin/view/Geant4/QuickMigrationGuideForGeant4V10

Step 1 – G4UserActionInitialization

SLAC

G4VUserActionInitialization is a newly introduced class for the user to instantiate user action classes (both mandatory and optional).

As described in the previous slides, 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.

G4UserActionInitialization – a new user class

SLAC

Main() for v9.6

```
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()

SLAC

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;

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

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

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

Step 3 – UserDetectorConstruction

SLAC

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

SLAC

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 RenerateRun() method.

Step 5 – G4Allocator

SLAC

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

SLAC

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 file reading must be Mutex-ed to avoid accidental coincidence.

Each thread may create its own output file. Be careful to specify different file name for each thread.

- For example G4Threading::GetG4ThreadID() gives you the unique thread ID.

Be careful, ROOT is not thread-safe. If you need a ROOT file as an output for your histograms or n-tuples, use G4Tool.

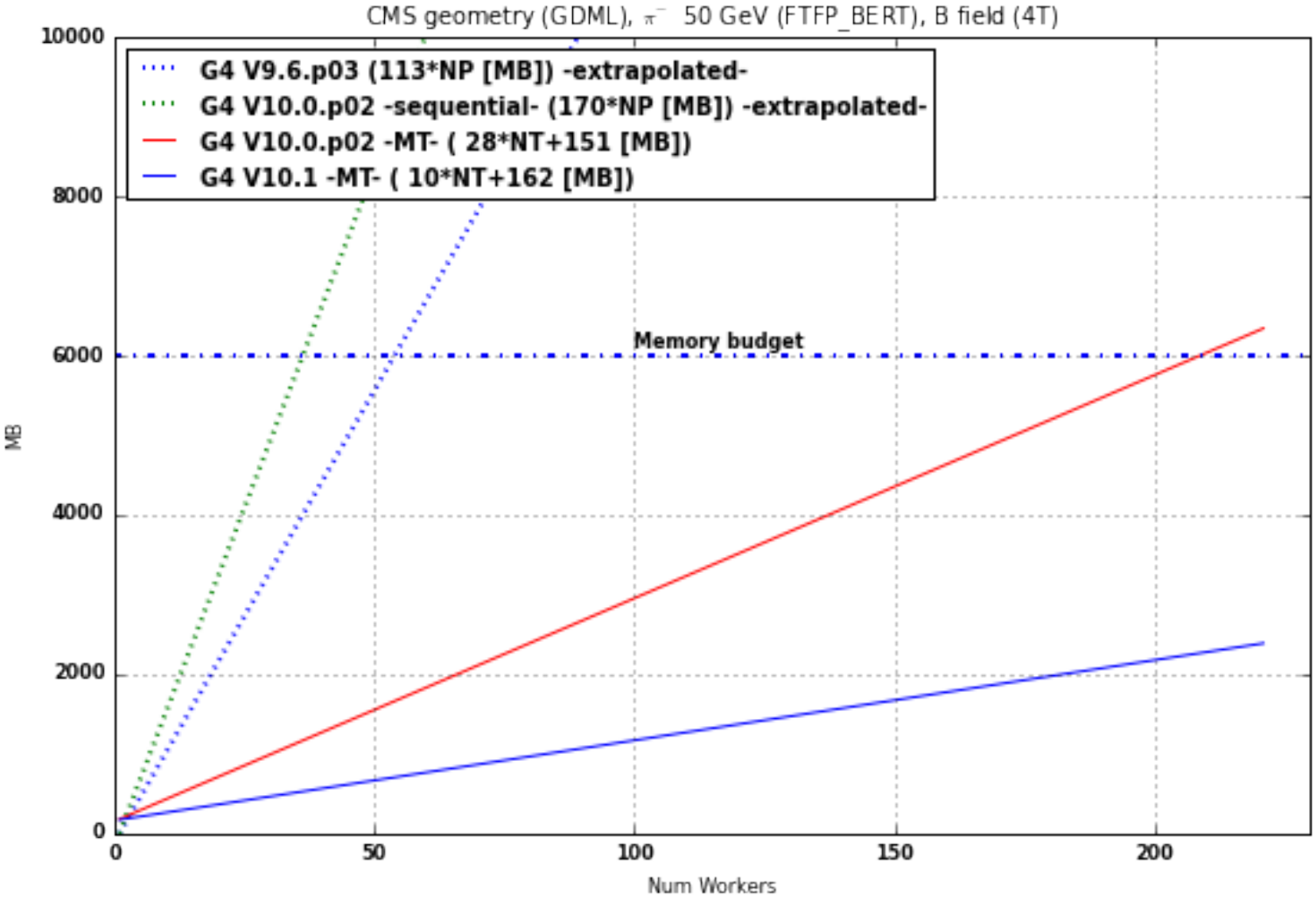
HPC Resources

SLAC

- Since few years Geant4 provides a MPI example to:
  - Show how to parallelize a G4 job using MPI
  - Steer from a single UI all MPI ranks
- With Geant4 Ver10.1 these examples have been extended to support MT:
  - Scale across nodes with MPI and use MT to scale across cores
- We are working a much improved version of G4-MPI library and examples (to be released for 10.2):
  - Merging of histograms across ranks
  - Better integration with cmake (easier to develop a MPI enabled application)
- We expect much easier integration of G4 jobs with HPC resources where MPI is de-facto standard

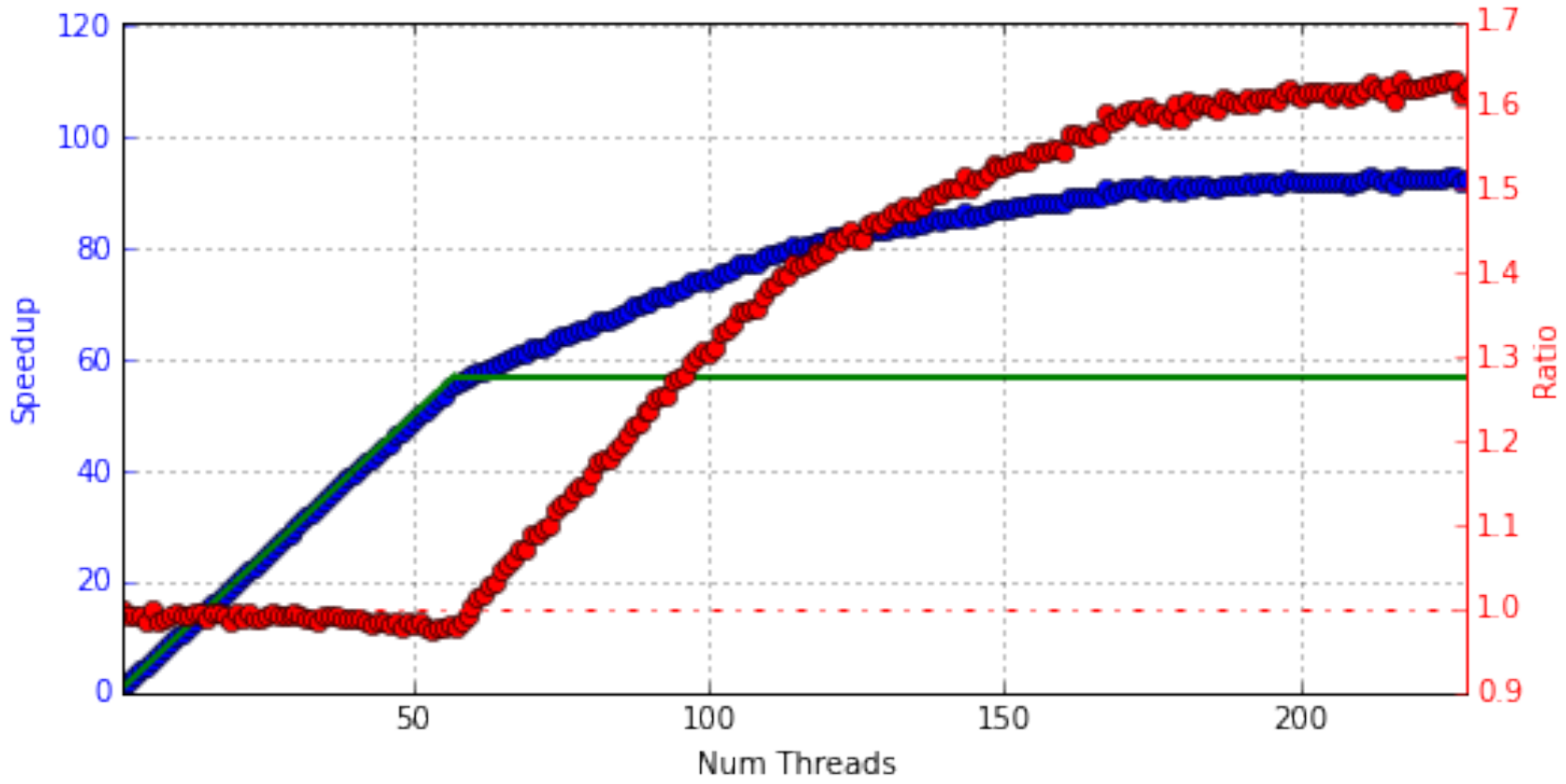
Some results

SLAC

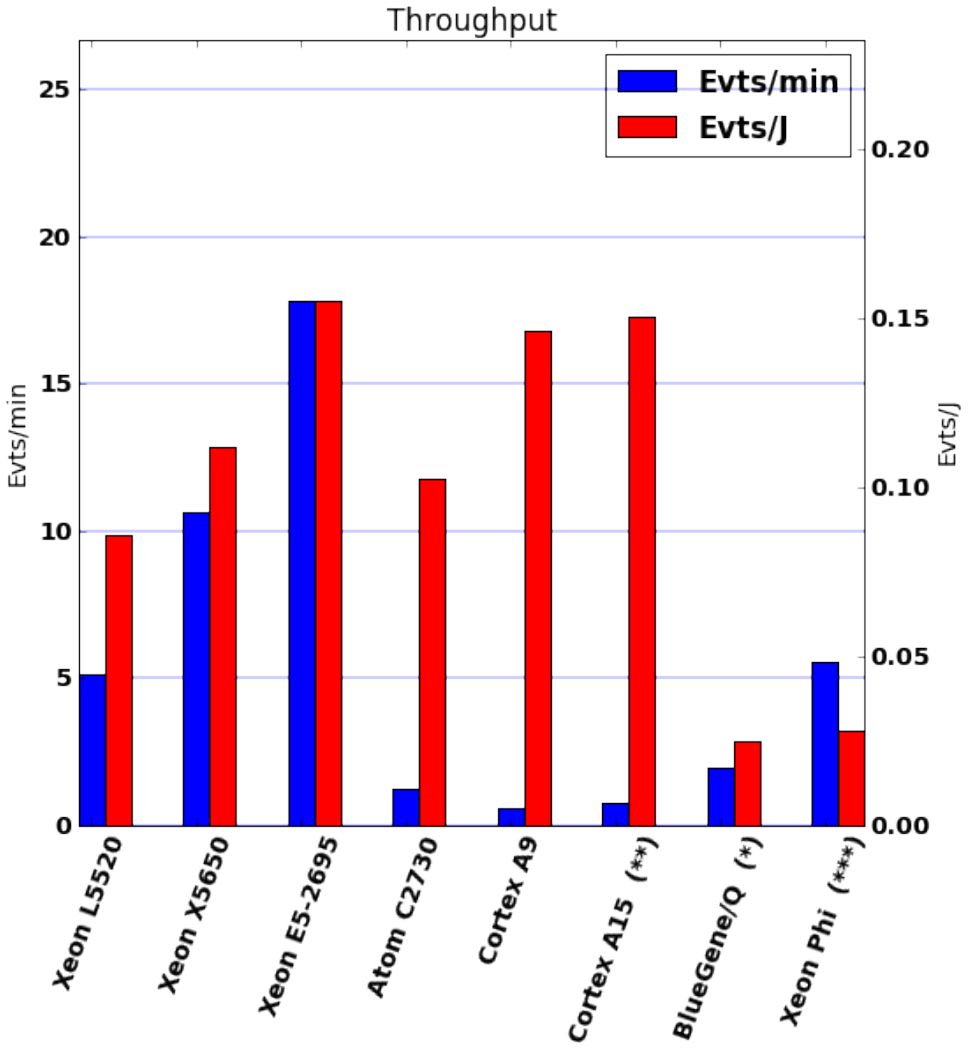


Speedup

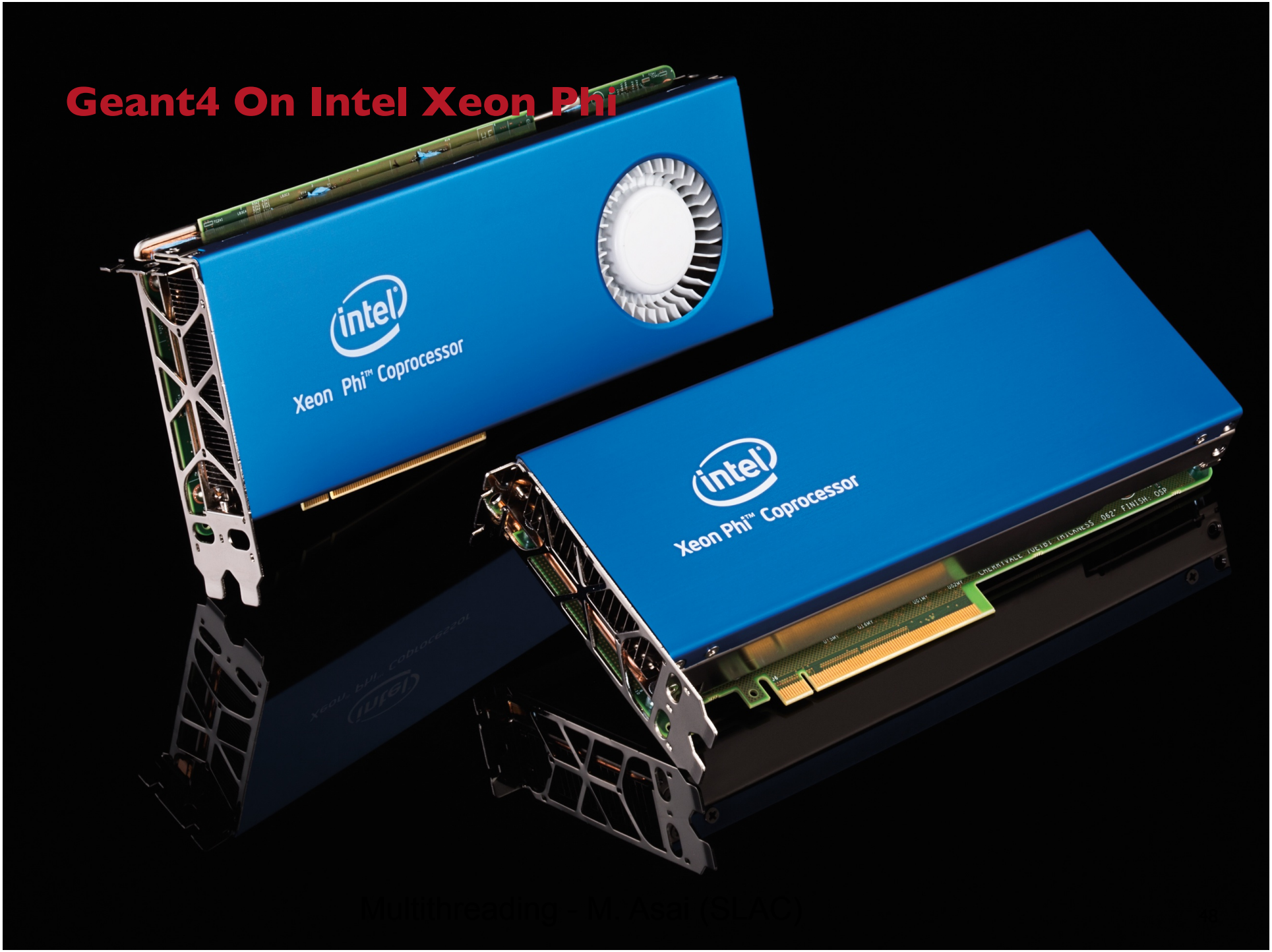
SLAC



Comparing architectures



Geant4 On Intel Xeon Phi

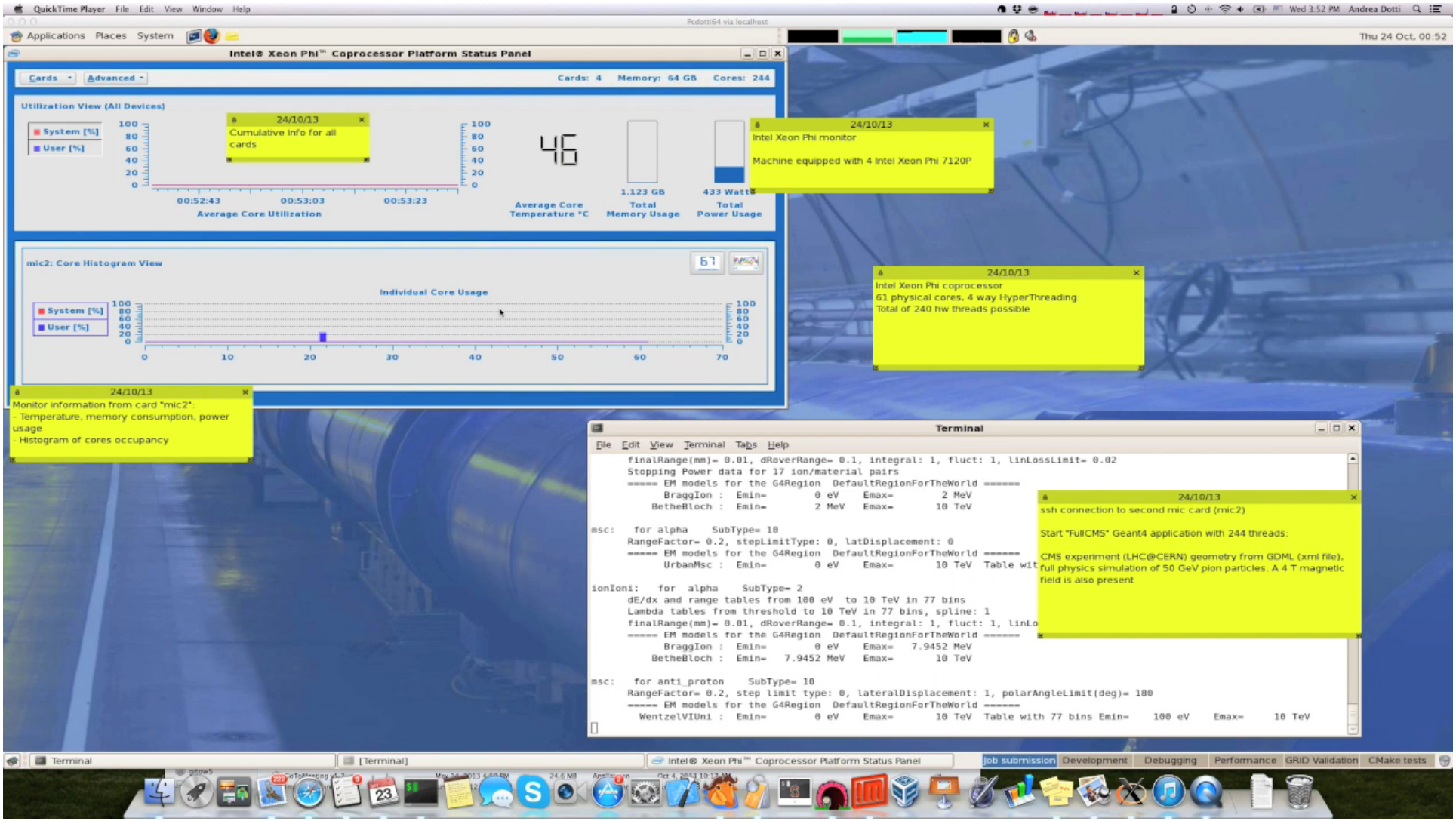


What is Intel Xeon Phi (aka MIC)?

SLAC

- A PCIe card that acts as a **"co-processor"**
  - In a certain sense similar to using a GPU for general computing (I know, I'm not precise here…)
  - Up to 8 cards per host
- Based on **x86 instruction sets**
  - You do not need to rewrite your code, "just" recompile
- It requires Intel compiler (**not free**) and RTE
- **61 cores (x4 ways hyper-threading),** w/ max 16GB of RAM
  - Each core is much less powerful than a core of your host
  - In our experience: if your G4 code scales well 1 full card ~ 1 host
- Two ways of running code on the card:
  - Offload (a-la GPGPU)
  - Native: start a cross-compiled application on the card
- Geant4 has been ported to compile and run on MIC cards in **Native mode**
  - **Xeon Phi will be one of the officially tested platforms for version 10.1.**

Running Geant4 on Xeon Phi



Conclusions

SLAC

Parallelism is a tricky business:

- User code has to be thread-safe
- Race conditions may appear (better: they will very probably appear)
- Bugs may often seem "random" and difficult to reproduce
- Experience is needed for complex applications, but we believe for simple ones following these instructions is enough
- A new hyper news user forum has been created (Multithreading) to address all possible questions
- Ask an expert!

Vectorization

SLAC

Local vectorization is surely beneficial, but full vectorization far beyond each individual physics interaction may not be.

- Switches (e.g. if-statements) are almost unavoidable for complicated physics / geometry.

Recent Intel press release

- To cope with the huge difference between the power consumption of Integer and AVX code, Intel is introducing new base and Turbo Boost frequencies for all their SKUs; these are called AVX base/Turbo. For example, the E5-2693 v3 will start from a base frequency of 2.3GHz and turbo up to 3.3GHz when running non-AVX code. When it encounters AVX code however, it will not able to boost its clock to more than 3GHz during a 1 ms window of time. If the CPU comes close to thermal and TDP limits, clock speed will drop down to 1.9GHz, the "AVX base clock".
  - Note: Advanced Vector Extensions (AVX) are extensions to the x86 instruction set architecture for microprocessors from Intel and AMD proposed by Intel.