MultiThreading in Geant4

Luciano Pandola

INFN – Laboratori Nazionali del Sud

pandola@lns.infn.it

A lot of material by J. Pipek

The 2nd Geant4 School in China, Shandong University, Qingdao, March 25th-29th, 2019
Part I: Motivation
Motivation: performance/$

- Multi-core CPUs
- Expensive memory

⇒ Memory optimization is more and more important!
Processes vs. threads

- **Processes** are *separate instances* of running computer programs that have their exclusive execution context, memory and other system resources.

- **Threads** are parallel “independent” executions *within a process*. They *share* the same memory space and system resources (of the process).
Concept for multi-thread ...
... vs. parallelisation

- Each node hosts a complete simulation
- Many copies of geometry and physics tables
- More memory-thirsty
Situation of Monte Carlo sims

- Single-particle simulation is **trivially parallelizable**!
- Each event can be simulated **independently**
  - not too much per-event state
  - not too much memory necessary for computation
- A lot of "**static**" data
  - complicated **geometries** (+ their optimization)
  - physics tables (**cross-section** data)
  - **electromagnetic** fields (if present)

⇒ We can benefit a lot from efficient memory sharing!
Solutions: threads

Advantages:
- memory & resource effectivity (sharing)
- in-process synchronization

Disadvantages:
- difficult to write properly
- difficult to debug (indeterministic behaviour)
- race conditions / dead-locks
- thread synchronization costs
Memory in MT applications

Without MT:
- Detector geometry & cross-section tables
- MEMORY SPACE
- AVAILABLE CORES
  - Active cores
  - Unused cores

With MT:
- MEMORY SPACE
- AVAILABLE CORES
  - Active cores

Transient per event data (tracks, hits, etc.)
Performance in MT mode

4-core CPU with hyper-threading (8 logical cores)

- Hypothetical
- Actual

- Real physical cores
- Hyper-threading
- No further gain 😞
Part II: Multi-threading in Geant4
Execution modes in Geant4

- **Sequential mode**
  - everything run in one thread only
  - accepts both user actions and action initialization to support old code (Geant4 < 10.0)

- **Multithreaded mode**
  - “master” thread for the application
  - events simulated in multiple “worker” threads
  - accepts only action initialization
  - Recently supported also in **Windows OS**
    - Starting from Geant4 10.5

**Good news:** The same code may support both modes!
Multithreading in Geant4

Main thread ("master")
- initialization of geometry and physics
- user interface
- start worker threads
- distribute events
- merge results

Worker threads
- event simulation
- partial results
- user actions
Event processing

Note: The diagram is simplifying a bit: events are distributed in seemingly random (though optimized) chunks, not split among workers one-by-one or in equal parts.
G4MTRunManager

- Substitute for sequential **G4RunManager**
  - inherits from it
  - disables the **SetUserAction()** methods

- **Additional** responsibilities
  - start worker threads
  - **distribute** events among the workers
  - take care about **merging** of runs
Run manager relations

- **User Initializations**
  - G4VUserPhysicsList
  - G4VUserDetectorConstruction
  - G4VUserActionInitialization
  - G4UserRunAction

- **User Actions**
  - G4VUserPrimaryGeneratorAction
  - G4UserRunAction
  - G4UserSteppingAction
  - G4UserTrackingAction

- **G4MTRunManager**
  - G4UserWorkerInitialization
  - G4WorkerRunManager
  - G4WorkerThread

Special case: run action for master to allow for reductions of partial results
G4UserRunAction in MT mode

This action (unlike the rest) can apply in both worker and master threads:

- To distinguish where you are, use `IsMaster()` method
- If you have behaviour for master, register the instance in `G4VUserActionInitialization::BuildForMaster()`

```cpp
void MyActionInitialization::Build() const {
    SetUserAction(new MyRunAction());
    // ...other actions
}

void MyActionInitialization::BuildForMaster() const {
    SetUserAction(new MyRunAction());
    // Only run action
}
```

Note: This, in principle, can be a different class
Merging of runs

- Usually tricky
- Geant4-native tools automatically
  - command-based scoring
  - \texttt{g4analysis} (histograms summed, trees in separate files)
- Custom data require manual approach
  - in \texttt{G4Run::Merge()} (of your custom "MyRun")
  - in \texttt{G4RunEventAction::EndOfRunAction()}

```c
void MyRunAction::EndOfRunAction(const G4Run* run) {
    // ...
    // Merge accumulables
    G4AccumulableManager* accumulableManager = G4AccumulableManager::Instance();
    accumulableManager->Merge();
    // ...
```
main() for both modes

- CMake setting
  
  \texttt{-DGEANT4\_BUILD\_MULTITHREADED=ON/OFF}

- Preprocessor macro \texttt{G4MULTITHREADED}

```cpp
#include <G4MTRunManager.hh>
#include <G4RunManager.hh>

int main() {
    #ifdef G4MULTITHREADED
        G4MTRunManager* runManager = new G4MTRunManager;
    #else
        G4RunManager* runManager = new G4RunManager;
    #endif
    //..
}
```
Set the number of threads

- Default number of threads: 2
- Change this using
  - UI command:
    - /run/numberOfThreads 6
    - /run/useMaximumLogicalCores
  - C++ code:
    - `runManager->SetNumberOfThreads(4)`
  - Environment variable (highest priority):
    - `G4FORCENUMBEROFTHREADS=4`
  - `G4Threading::G4GetNumberOfCores()` tells the actual number of logical cores
- Further tweaking options available (advanced)

Note: Must be done in pre-initialize stage
Multithreaded G4cout

- If you use **G4cout** for output, it’s relatively synchronized and each message is prepended with the thread number
  - **Note:** this does not work with **std::cout** (another reason not to use it!)

```plaintext
### Run 0 starts.
G4WT1 > EventAction: absorber energy/time scorer ID: 0
G4WT1 > EventAction: scintillator energy/time scorer ID: 1
G4WT0 > EventAction: absorber energy/time scorer ID: 0
G4WT0 > EventAction: scintillator energy/time scorer ID: 1
  Run terminated.
Run Summary
  Number of events processed : 10000
  User=21s Real=11.36s Sys=1.59s
```
Multithreaded G4cout

- To **buffer** the output from each thread at a time, so that the output of each thread is grouped and printed at the end of the job:
  /control/cout/useBuffer true|false

- To **limit** the output from threads to one selected thread only:
  /control/cout/ignoreThreadsExcept 0

- To **redirect** the output from threads in a file:
  /control/cout/setCoutFile coutFileName
  /control/cout/setCerrFile cerrFileName
Part III: Thread-aware coding
Good news!

You don’t have to care (too much) about threading issues, provided that you:

- Don’t manually open external files
- Use `g4analysis / command-based scoring` for output
- Avoid static variables and fields (especially in user actions)
- Correctly merge runs if using accumulables or hits
- Use the `G4 (MT) RunManager` trick in main() (see above)
- Use `G4VUserActionInitialization`
- Don’t experiment with Geant4 kernel (especially not in user actions)

If you don’t meet these conditions, you must write thread-aware code.
Writing thread-safe code

- Find out which variables are modified inside the worker threads:
  - these must not be static!
  - use G4ThreadLocal if possible
  - split the classes if necessary

- Variable “locality”:
  - don’t use global variables
  - don’t use static class fields
  - prefer local variables to class fields

- Be careful about deleting pointers

- Use mutexes & locks when you access a shared resource
Mutex is an object variable that can be locked so that only one thread can use it at the same time.

Lock is an act of locking the mutex:
- locking an open mutex succeeds immediately
- locking a locked mutex blocks and waits until it is available again

Manipulation with shared resources should be encapsulated by locking/unlocking a particular mutex
Mutex'es and locks in Geant4

- **Mutex** is best created as static object inside an anonymous namespace (class *G4Mutex*)

```cpp
namespace { G4Mutex myMutex = G4MUTEX_INITIALIZER; }
```

- **G4AutoLock** is a “clever” implementation of the locking mechanism:
  - you just create it with mutex address as parameter
  - when the object is destroyed (end of function or block), the mutex is automatically freed

```cpp
{    
    G4AutoLock(&myMutex);
    // ... (do something)
} // Now, the mutex is freed.
```
Drawbacks and caveats of locking

- Synchronization & locking is not CPU costly
- Using multiple locks can lead to a dead-lock:
  - Threads need mutexes A and B to proceed
  - Thread1 has locked mutex A
  - Thread2 has locked mutex B
  - No thread can acquire the second lock!!!

Alternatives:
- There are more sophisticated threading tools
- Avoid using shared resources as much as possible
G4AutoDelete

- If you don’t know when to properly delete an object in threads (typical case!), you can register it with **G4AutoDelete**

```cpp
#include "G4AutoDelete.hh"
// ...  
G4AutoDelete::Register(aPointer);  
// ...  
```

- This will ensure that the object is deleted when the worker thread ends.
Thread-safe I/O

- Geant4’s **scoring** and **g4analysis** are **thread-safe**

Custom **output** (alternatives):
- Have **one file per thread** (or per each instance of user action class)
- Have **only one file and guard the procedure** by mutex, add some caching mechanism

Custom **input**:  
- Read **everything in master** thread and share the data as **read-only**  
- Reading **on demand** – protect by mutex, add some caching mechanism
namespace { G4Mutex myMutex = G4MUTEX_INITIALIZER; }

MyFileReader* MyPrimaryGenAction::fileReader = nullptr;

MyPrimaryGenAction::MyPrimaryGenAction(G4String fileName) {
    G4AutoLock lock(&myMutex);
    if (!fileReader) fileReader = new MyFileReader(fileName);
    particleGun = new G4ParticleGun(1);
    // ...Define particle properties
}

MyPrimaryGenAction::~MyPrimaryGenAction() {
    G4AutoLock lock(&myMutex);
    if (fileReader) { delete fileReader; fileReader = 0; }
}

void MyPrimaryGenAction::GeneratePrimaries(G4Event* anEvent) {
    G4ThreeVector momDirection;
    G4AutoLock lock(&myMutex);
    momDirection = fileReader->GetAnEvent();
    particleGun->SetParticleMomentumDirection(momDirection);
    // ...Set other particle properties
}
Conclusion

- Geant4 offers an **optimized multithreaded mode** (optional)
- Multithreading is powerful but a complex and potentially **dangerous** tool
Hands-on session

- Task 4
  - Task4e: Try to run in sequential and MT

- http://202.122.35.42/task4