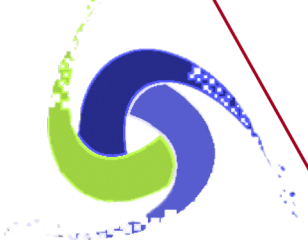


GEANT4
A SIMULATION TOOLKIT
Version 10.5

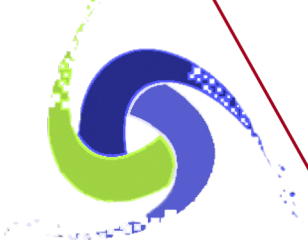
Multithreading II

Makoto Asai (SLAC)

Geant4 tutorial course

Contents

- The challenge of MT : thread safety
- TBB and MPI
- Reading input file in multithreaded mode



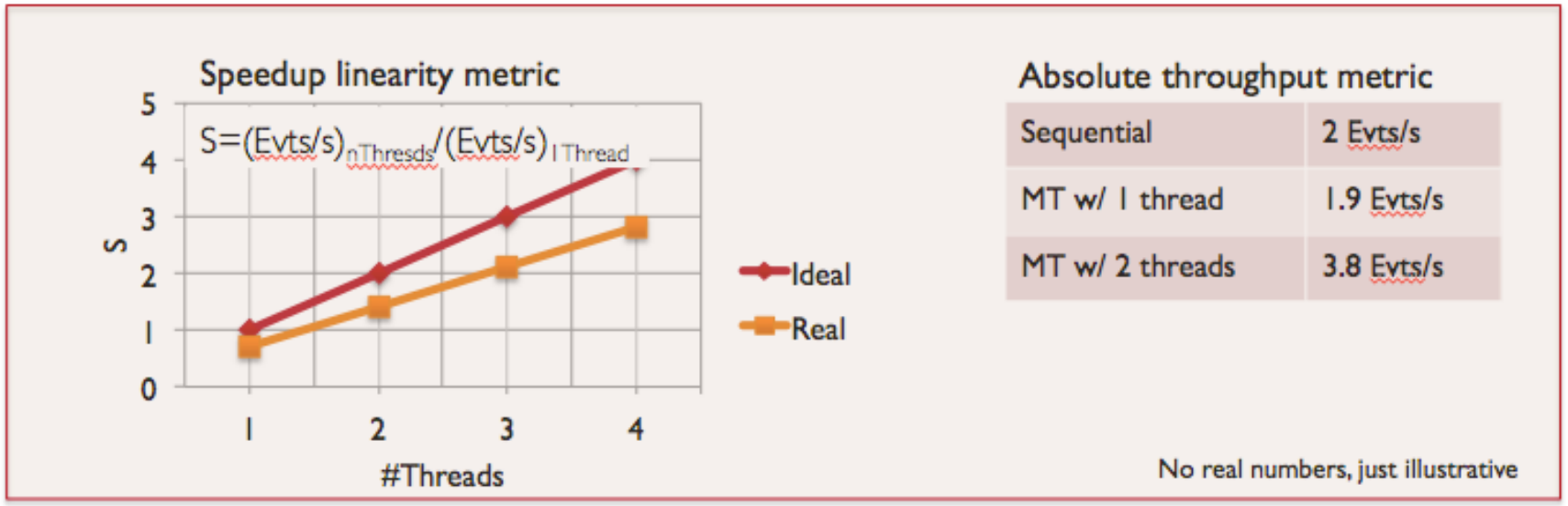
GEANT4
A SIMULATION TOOLKIT

Version 10.5

The challenges of MT: thread-safety

Definitions

- **Design to minimize changes in user-code**
- Maintain API changes at minimum
- Focus on **linearity of speed-up** (w.r.t. #threads) is the most important metric
- Enforce use of **POSIX standards** to allow for integration with user preferred parallelization frameworks (e.g. MPI, TBB, …)



Absolute throughput metric

Sequential	2 Evts/s
MT w/ 1 thread	1.9 Evts/s
MT w/ 2 threads	3.8 Evts/s

No real numbers, just illustrative

Thread safety a simple example

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

```
double aSharedVariable;

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

Thread safety a simple example

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

```
double aSharedVariable;
```

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

T1

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

T2

Thread safety a simple example

- *result* is a local variable, exists in each thread separately i.e. not a problem
- T1 arrives **here** and does something

```
double aSharedVariable;

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

T1

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

T2

Thread safety a simple example

- Now T2 starts and arrives **here**, the shared resource value is not yet updated, what is the expected behavior? what is happening?

```
double aSharedVariable;
```

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

T1

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

T2

Thread safety a simple example

- 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)

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

```
double aSharedVariable;
```

T1

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

T2

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

Thread safety a simple example

- Do we really need to share aSahredVariable?
- if not, minimal change required, each thread has its own copy
- 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

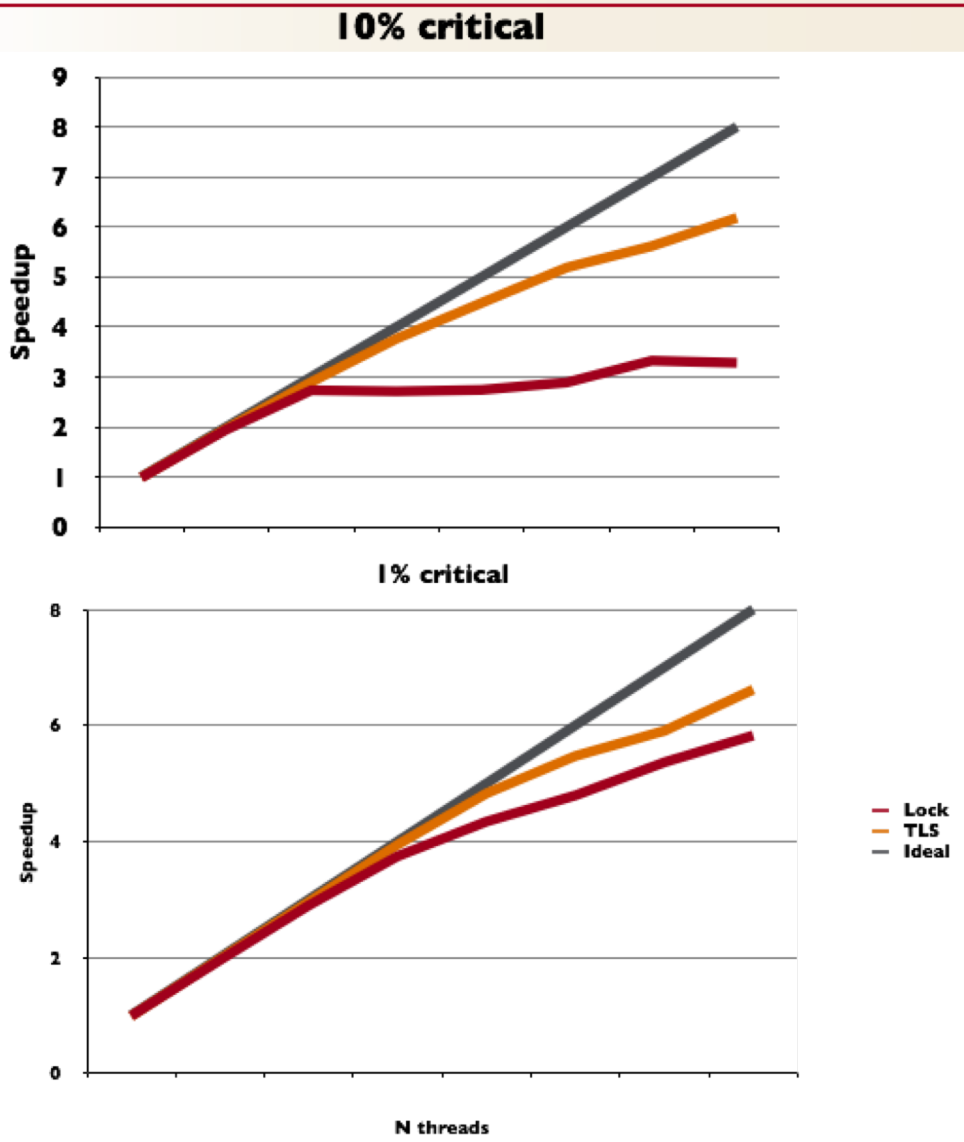
```
double __thread
aSharedVariable;

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

```
double __thread
aSharedVariable;

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

Thread Local Storage

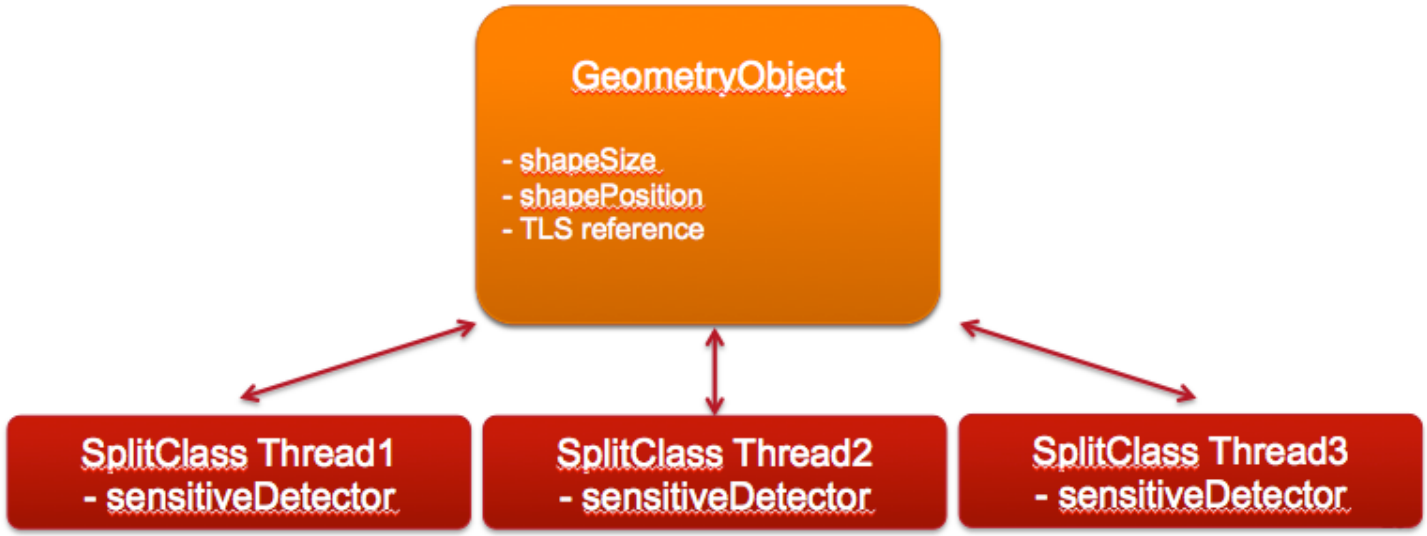


NB: results obtained on toy application, not real G4

- 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

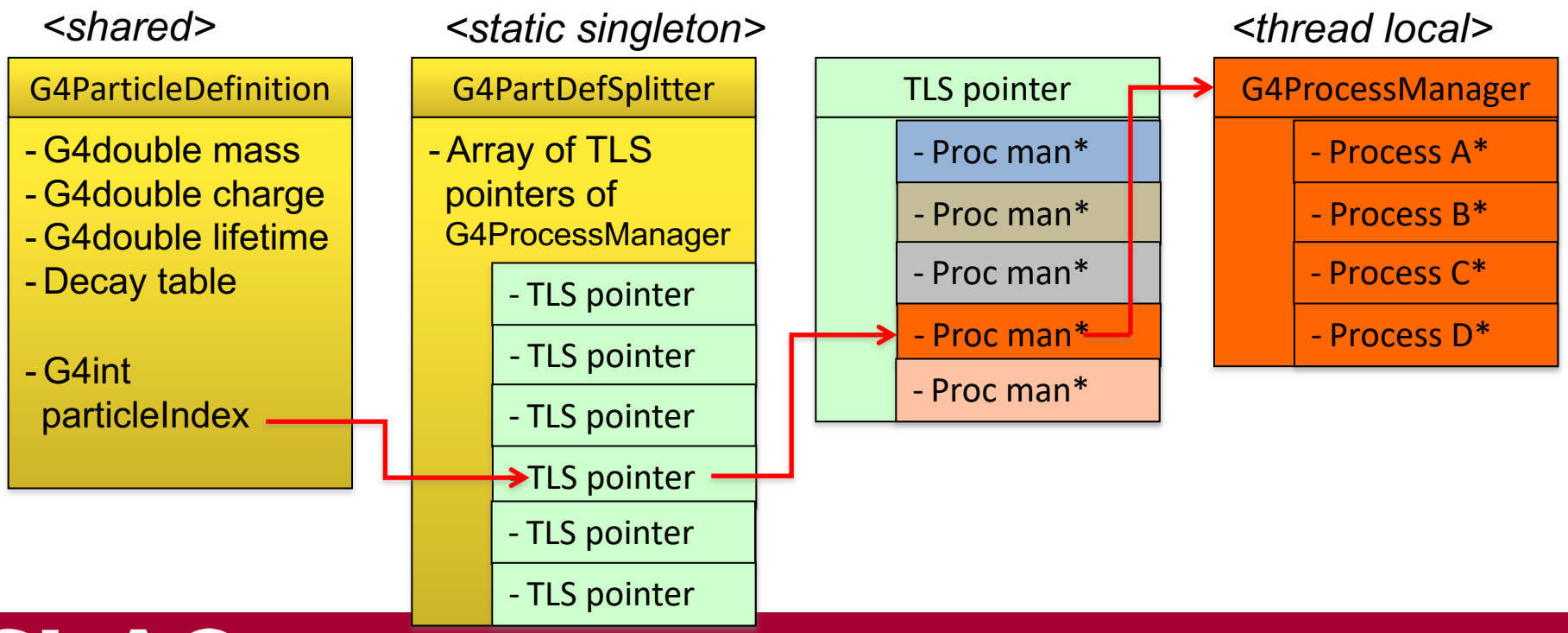
The split-class mechanism concept

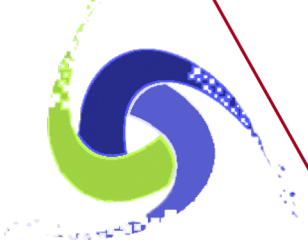
- Thread-safety implemented via **Thread Local Storage**
- "Split-class" mechanism: reduce memory consumption
- Read-only part of most memory consuming objects shared between thread
- Geometry, Physics Tables
- Rest is thread-private



Split class – case of particle definition

- 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.





GEANT4

A SIMULATION TOOLKIT

Version 10.5

TBB and MPI

Integration with TBB

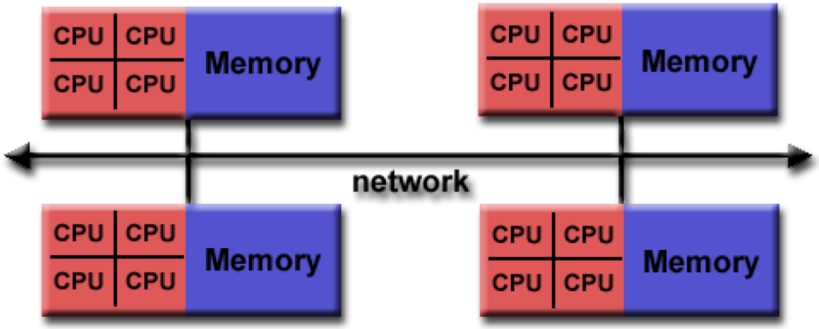
- Intel Thread Building Block library
  - Task-based parallelism
  - Freely available for Linux/Mac/WIN
- We provide an example:
  - example/extended/parallel/TBB
  - Basic integration of TBB with Geant4 Version 10.0
  - Basically it replaces the POSIX multi-threading system we provide
- Note: we plan to intensively work on TBB examples in 2016, we will review and extend this example

HPC Resources

- Since few years Geant4 provides an 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

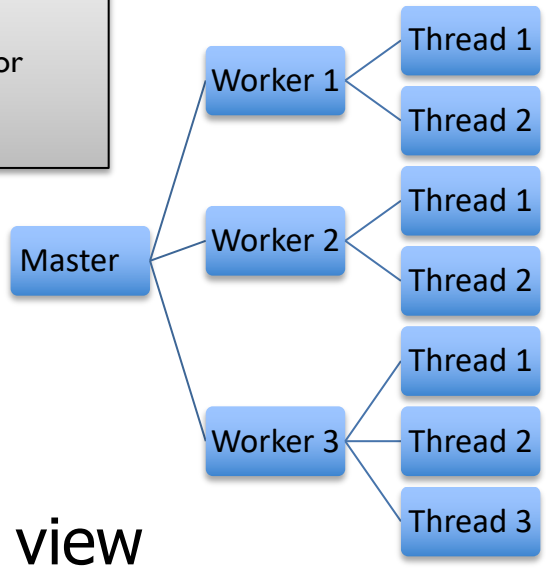
MPI and Geant4

- MPI optimized for large (and/or frequent) messages
  - Geant4 ranks have very little communication among them…
- …still MPI is an attractive possibility for several reasons:
  - excellent support from a very large community (HPC resources)
  - attractive for smaller communities (simpler use w.r.t. e.g. GRID)
  - preferred way of heterogeneous running on Xeon Phi systems



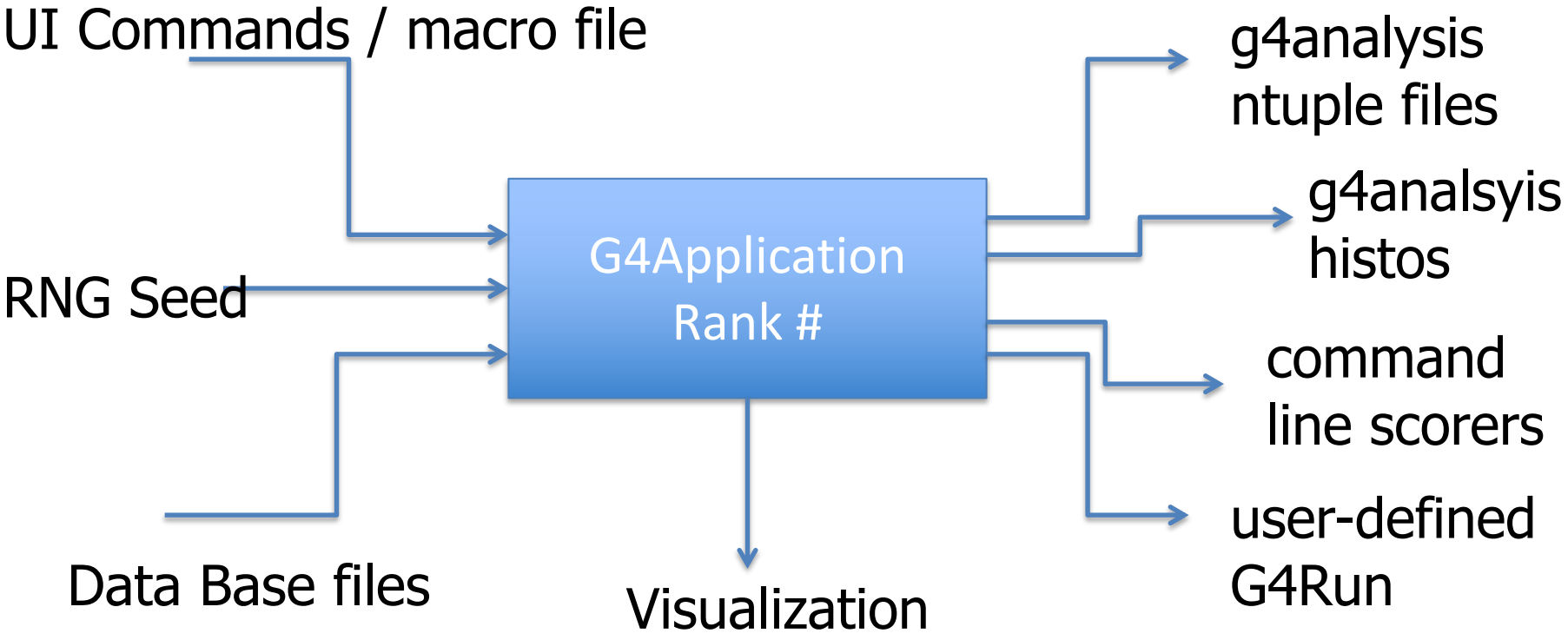
Physical view

Master broadcasts UI commands and RNG seeds
Workers send back results for merging: histograms, ntuples, scorers



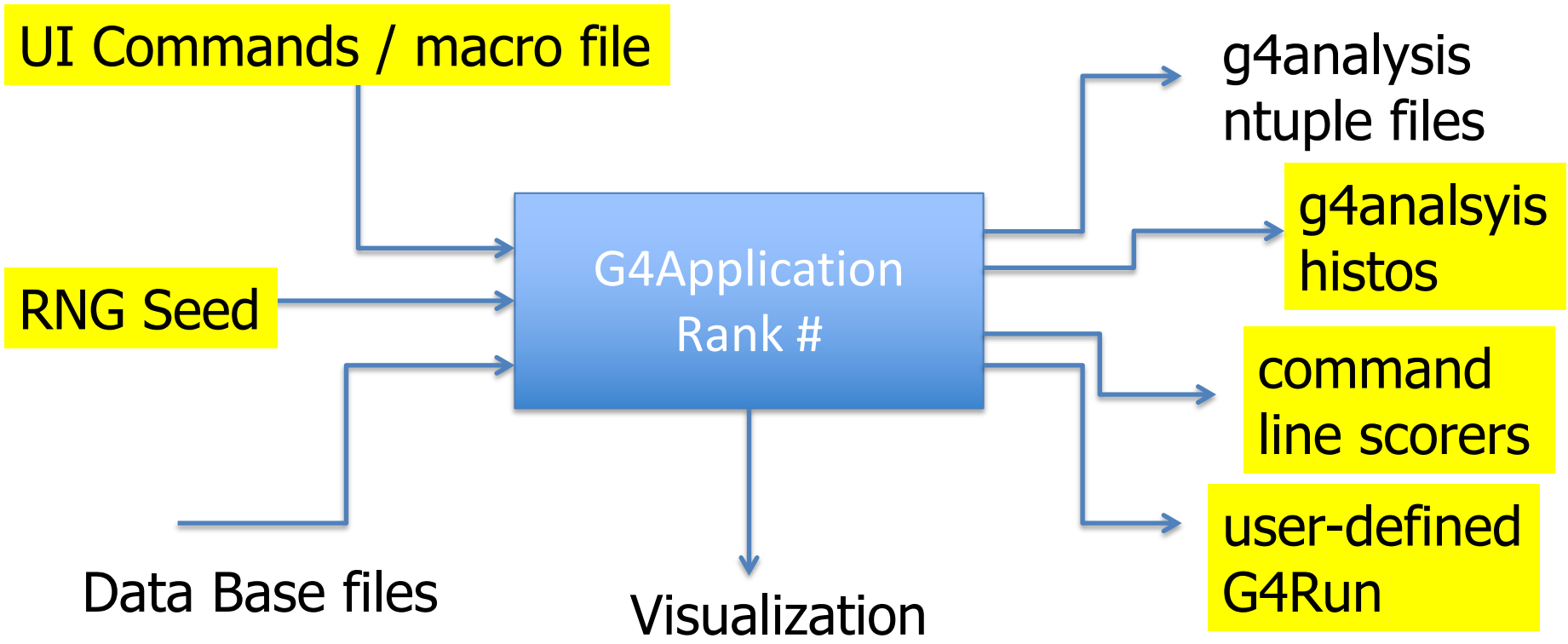
Logical view

Geant4 applications from MPI point of view



Ideally all input/outputs should become MPI enabled and do not use on filesystem

Geant4 applications from MPI point of view



Ideally all input/outputs should become MPI enabled and do not use on filesystem

Provided in Geant4 Version 10.3 (December 2016)

More memory-efficient, more HPC friendly

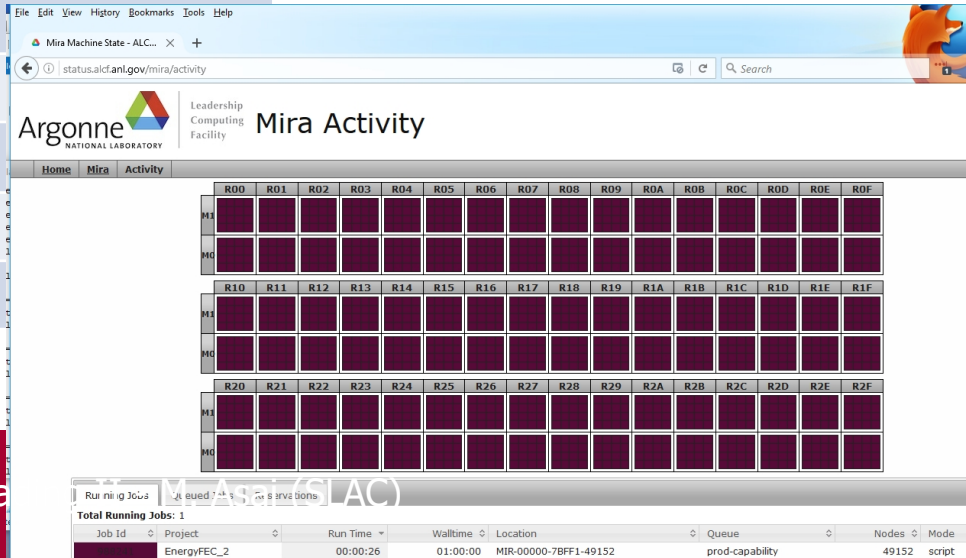
Version	Intercept	Memory/thread
9.6 (seq.)	113 MB	(113 MB)
10.0.p02-seq	170 MB	(170 MB)
10.0.p02-MT	151 MB	28 MB
10.3.beta-MT	148 MB	9 MB

Memory space required for Intel Xeon Phi 3120A
Full-CMS geometry (GDML), 4 Tesla field, 50 GeV pi- (FTFP_BERT)

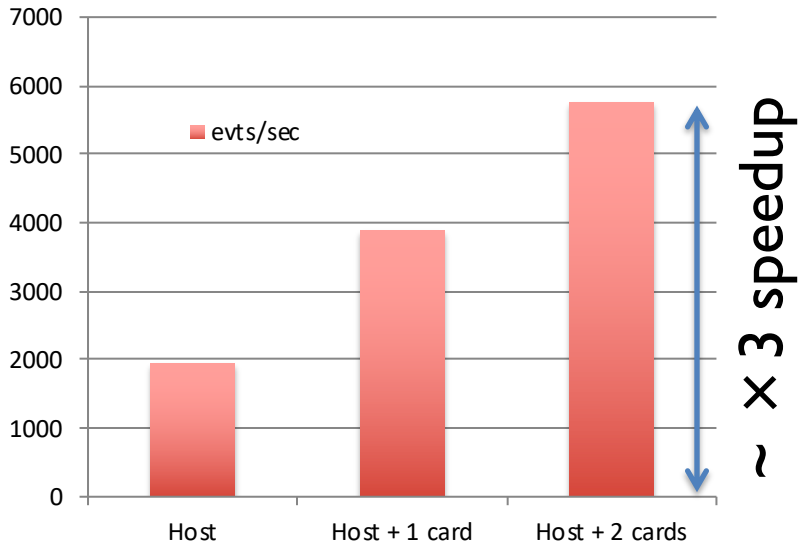
# of CPU	# of threads	Speed-up factor	efficiency
10	80	79	98. 8%
20	160	158	98. 8%
40	320	317	99. 0%
80	640	626	97. 8%
160	1280	1251	97. 7%
320	2560	2297	89. 7%
640	5120	3555	69. 4%

Tachyon-2 supercomputer @ KISTI (South Korea)
FTFP_BERT physics validation benchmark

- Geant4 has successfully run with a combination of MT and MPI on Mira Bluegene/Q Supercomputer (@ANL) with all of its 3 million threads
  - Full-CMS geometry & field
- I/O is the limiting factor to scale large concurrent threads:
  - Granular input data files, output data/histograms, etc.
  - 2017 work item
  - Targeting also Cori @ NERSC



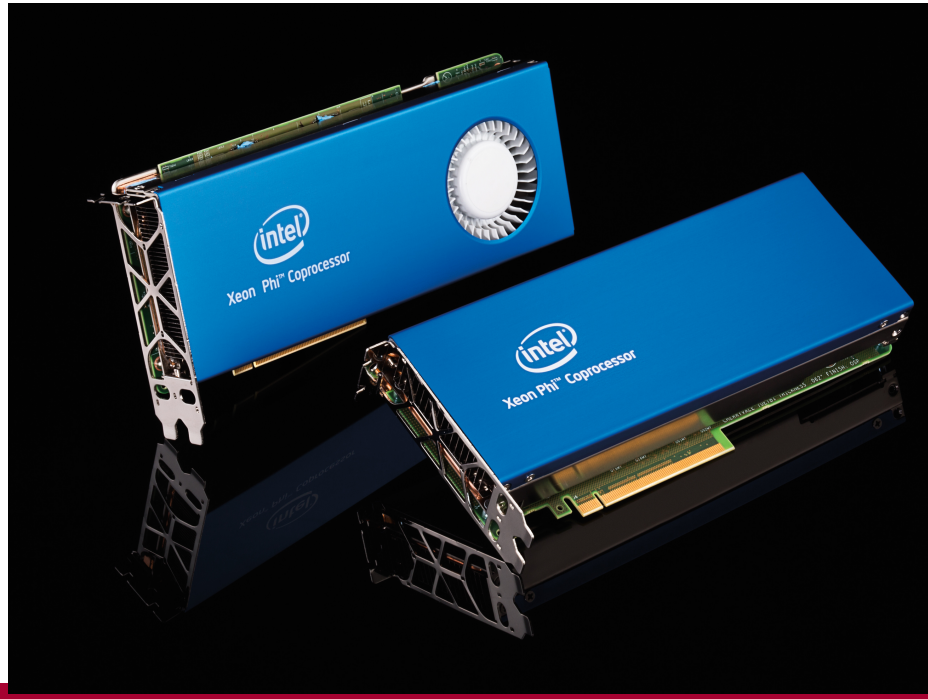
An interesting possibility…



System:

Intel E5-2600 @ 2.2GHz (8C/16T)

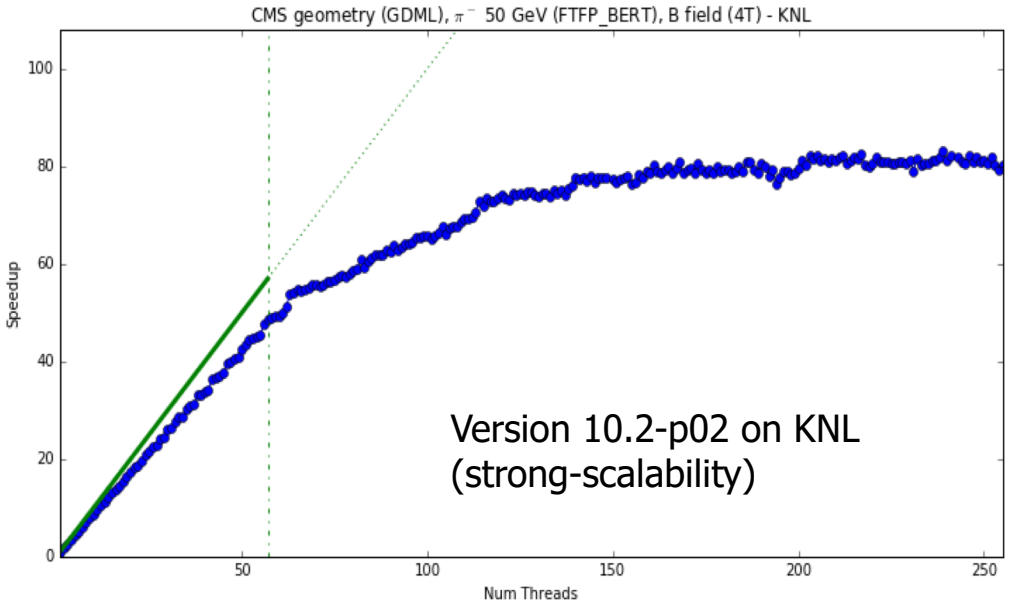
2 Xeon Phi cards model 3120A (57C/228T)



MPI application started on host and on two MICs: a small cluster in your desktop

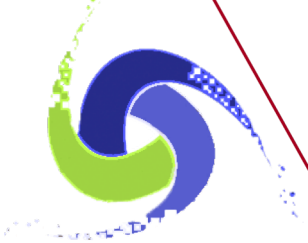
"Medical" benchmark: proton 200 MeV on water phantom

KNL vs. KNC



- For three years we have provided support for running Geant4 on KNC.
  - ATLAS, CMS successfully multithreaded
- We will soon extend our support to KNL.
  - With KNL, thanks to x86 binary compatibility including the use of gcc, work-flow is tremendously simplified.

System	Time to completion (5k events)
Xeon E5-2620 @ 2.1 GHz (12 cores, 24 threads)	570 s
KNC (31s1P) @ 1.0 GHz (228 threads)	1000 s
KNL (7210, quadrant mode, MCDRAM only) @ 1.3 GHz (255 threads)	378 s (x3 improvement w.r.t. KNC)
KNL (shared library)	480 s (25% slower than static library)



GEANT4
A SIMULATION TOOLKIT
Version 10.5

Reading input file for primary particles in MT mode

Input data file for PrimaryGeneratorAction

- PrimaryGeneratorAction is a thread-local class. Thus, special attention is required if the user needs an input file to read primary particles.
  - If thread-local objects of PrimaryGeneratorAction naively open a file and read it, they all read the file from the beginning and they all read exactly the same input.
- The appropriate implementation of reading an input file for PrimaryGeneratorAction depends on the use-case, more precisely, on the ratio of the cost of accessing to the data file to the total execution time of one event on one thread.
  - If this ratio is low, i.e. execution time dominates (typically the case of high-energy physics experiment where the energy of primary particles are high), the file access can be shared with Mutex locking mechanism.
  - If this ratio is high, i.e. file access is a significant fraction of the execution time (typically the case of medical and space applications where the energy of primary particles are rather low), the input data should be cached in addition to the Mutex locking.
- To use Mutex locking mechanism, include G4AutoLock.hh header file.

high-energy physics sample code with G4HEPEvtInterface

- Here is a high-energy physics sample code with G4HEPEvtInterface that reads a Pythia input file. G4HEPEvtInterface has to be a single object shared by all the PrimaryGeneratorAction objects, and the access to G4HEPEvtInterface::GeneratePrimaryVertex() should be protected by Mutex.

MyHepPrimaryGenAction.hh

```
#include "G4VUserPrimaryGeneratorAction.hh"
class G4HEPEvtInterface;
class MyHepPrimaryGenAction
 : public G4VUserPrimaryGeneratorAction
{
 public:
   MyHepPrimaryGenAction(G4String fileName);
   ~MyHepPrimaryGenAction();
   . . .
   virtual void GeneratePrimaries(G4Event* anEvent);
 private:
   static G4HEPEvtInterface* hepEvt;
};
```

Detailed example of reading Pythia output file is found in examples/extended/runandEvent/RE05

MyHepPrimaryGenAction.cc

```
#include "MyHepPrimaryGenAction.hh"
#include "G4HEPEvtInterface.hh"
#include "G4AutoLock.hh"
namespace { G4Mutex myHEPPrimGenMutex = G4MUTEX_INITIALIZER; }
G4HEPEvtInterface* MyHepPrimaryGenAction::hepEvt = 0;

MyHepPrimaryGenAction::MyHepPrimaryGenAction(G4String fileName)
{
  G4AutoLock lock(&myHEPPrimGenMutex);
  if( !hepEvt ) hepEvt = new G4HEPEvtInterface( fileName );
}

MyHepPrimaryGenAction::~MyHepPrimaryGenAction()
{
  G4AutoLock lock(&myHEPPrimGenMutex);
  if( hepEvt ) { delete hepEvt; hepEvt = 0; }
}

void MyHepPrimaryGenAction::GeneratePrimaries(G4Event* anEvent)
{
  G4AutoLock lock(&myHEPPrimGenMutex);
  hepEvt->GeneratePrimaryVertex(anEvent);
}
```

lower-energy sample code with G4ParticleGun

- Here is a lower-energy sample code with G4ParticleGun to shoot 10 MeV electrons. The input file contains list of G4ThreeVector of the momentum direction (ex, ey, ez) and it is read by a dedicated file reader class MyFileReader. MyFileReader is shared by all threads, and reads 100 events at a time and buffers them. Primary vertex position is randomized.
- Please note that, for the simplicity of this sample code, it does not consider the end-of-file.

MyLowEPrimaryGenAction.hh

```
#include "G4VUserPrimaryGeneratorAction.hh"
class G4ParticleGun;
class MyFileReader;
class MyLowEPrimaryGenAction
 : public G4VUserPrimaryGeneratorAction
{
 public:
   MyLowEPrimaryGenAction(G4String fileName);
   virtual ~MyLowEPrimaryGenAction();
   . . .
   virtual void GeneratePrimaries(G4Event* anEvent);
 private:
   static MyFileReader* fileReader;
   G4ParticleGun* particleGun;
};
```

MyLowEPrimaryGenAction.cc – part 1/2

```
#include "G4ParticleTable.hh"
#include "G4ParticleDefinition.hh"
#include "Randomize.hh"
#include "G4AutoLock.hh"
namespace { G4Mutex myLowEPrimGenMutex = G4MUTEX_INITIALIZER; }
MyFileReader* MyLowEPrimaryGenAction::fileReader = 0;

MyLowEPrimaryGenAction::MyLowEPrimaryGenAction(G4String fileName)
{
 G4AutoLock lock(&myLowEPrimGenMutex);
 if( !fileReader ) fileReader = new MyFileReader( fileName );
 particleGun = new G4ParticleGun(1);
 G4ParticleTable* particleTable = G4ParticleTable::GetParticleTable();
 G4ParticleDefinition* particle = particleTable->FindParticle("e-");
 particleGun->SetParticleDefinition(particle);
 particleGun->SetParticleEnergy(10.*MeV);
}

MyLowEPrimaryGenAction::~MyLowEPrimaryGenAction()
{
 G4AutoLock lock(&myLowEPrimGenMutex);
 if( fileReader ) { delete fileReader; fileReader = 0; }
}
```

MyLowEPrimaryGenAction.cc – part 2/2

```
void MyLowEPrimaryGenAction::GeneratePrimaries(G4Event* anEvent)
{
  G4ThreeVector momDirction(0.,0.,0.);
  if(fileReader)
  {
    G4AutoLock lock(&myLowEPrimGenMutex);
    momDirection = fileReader->GetAnEvent();
  }
  particleGun->SetParticleMomentumDirection(momDirction);
  G4double x0 = 2.* Xmax * (G4UniformRand()-0.5);
  G4double y0 = 2.* Ymax * (G4UniformRand()-0.5);
  particleGun->SetParticlePosition(G4ThreeVector(x0,y0,0.));
  particleGun->GeneratePrimaryVertex(anEvent);
}
```

MyFileReader.hh and .cc

```
#include <list>
#include <fstream>
class MyFileReader
{
 public:
  MyFileReader(G4String fileName);
  ~MyFileReader();
  . . .
  G4ThreeVector GetAnEvent();
 private
  std::ifstream inputFile;
  std::list<G4ThreeVector> evList;
};
```

```
MyFileReader::MyFileReader(G4String fileName)
{ inputFile.open(filename.data()); }

MyFileReader::~MyFileReader()
{ inputFile.close(); }

G4ThreeVector MyFileReader::GetAnEvent()
{
 if( evList.size() == 0 )
 {
  for(int i=0;i<100;i++)
  {
   G4double ex, ey, ez;
   inputFile >> ex >> ey >> ez;
   evList.push_back( G4ThreeVector(ex,ey,ez) );
  }
 }
 G4ThreeVector ev = evList.pop_front();
 return ev;
}
```

Additional note for low-energy case

- Please note that Mutex has a performance penalty.
- Thus, in case the user uses many threads and the execution time of one event is very short, the most efficient way is splitting the input file to the number of threads so that each thread reads its own dedicated input file without Mutex lock.
- In this case, the buffering shown in above-mentioned MyFileReader class should be still used to reduce the file I/O overhead.