More on Multithreading

I. Hrivnacova, IPN Orsay

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

Geant4 ED PHENIICS Tutorial,
13 - 17 May 2019, Orsay

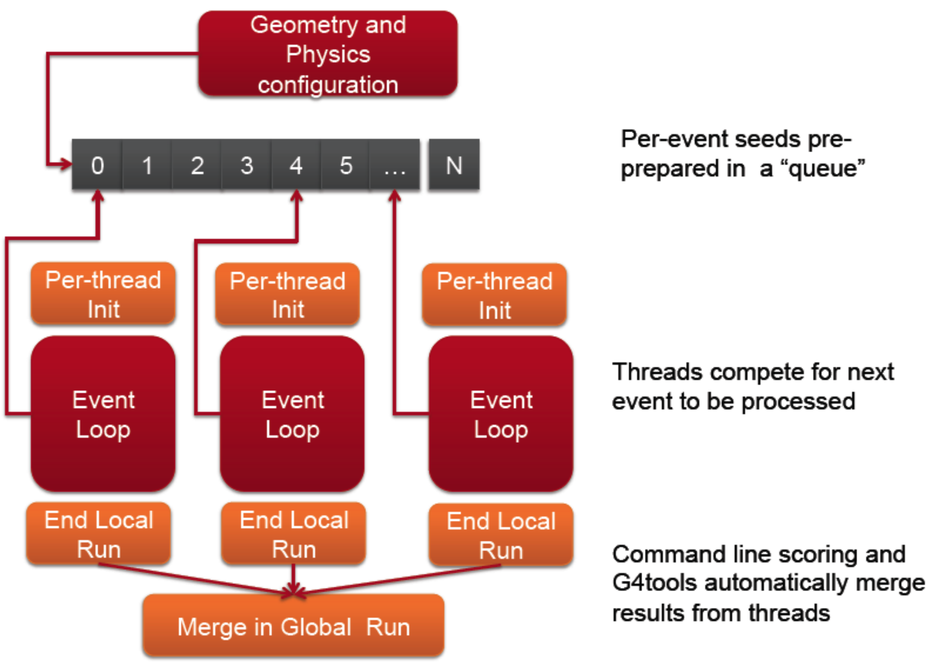
Outline

- Event data reduction
- Migration of a sequential Geant4 application to MT
- Results with Geant4 10.2 MT
  - Geant4 on MIC architecture
  - Scalability, memory reduction , reproducibility
- Integration with external frameworks

Event Data Reduction

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

Run Action

- In some users applications, UserRunAction is used to accumulate data from events and to calculate the result values for the whole run
  - E.g. in basic example B1, an energy deposited in a selected volume is accumulated event by event and a total dose is computed in the EndOfRunAction() method
- In multi-threading mode, the events are accumulated in UserRunAction objects instantiated **on workers** and the quantities accumulated on workers need to be merged in the UserRunAction **on master**
- This merging of the data accumulated on workers should be performed with use of G4Run or G4Parameter<T> class

Run Action - Sequential

- An example of a run action used to accumulate data from events: MyRunAction class
  - The run action class is the only action which is instantiated besides workers also on master

```
class MyRunAction : public G4UserRunAction
{
public:
  MyRunAction();
  virtual ~MyRunAction();

  virtual void BeginOfRunAction(const G4Run*);
  virtual void   EndOfRunAction(const G4Run*);

  void AddEdep (G4double e)
  { fEdep  += e;  fEdep2  += e*e;};

private:
  G4double  fEdep;
  G4double  fEdep2;
}
```

sequential

Data accumulated during event processing

Run Action + G4Run

- Separate data representing accounted data (if present) from your MyRunAction class in a new MyRun class (derived from G4Run)

MT (10.0)

```
class MyRunAction : public G4UserRunAction
{
public:
  MyRunAction();
  virtual ~MyRunAction();

  virtual G4Run* GenerateRun();
  virtual void BeginOfRunAction(co
  virtual void EndOfRunAction(cons

  void AddEdep (G4double e);

private:
  MyRun*  fRun;
}
```

```
class MyRun : public G4Run
{
public:
  Run();
  virtual ~Run();

  void AddEdep (G4double e)
  { fEdep  += e; fEdep2  += e*e;};

  virtual void Merge(const G4Run*);

private:
  G4double  fEdep;
  G4double  fEdep2;
}
```

Run Action + G4Run (2)

- Implementation of new or changed functions:

```
G4Run* MyRunAction::GenerateRun()
{
   fRun = new Run();
   return fRun;
}

void RunAction::AddEdep (G4double edep)
{
   fRun->AddEdep(edep);
}
```

MT (10.0)

```
void Run::Merge(const G4Run* localRun)
{
  fEdep += localRun->fEdep;
  fEdep2 += localRun->fEdep2;
}
```

This function is called by the master run instance for each worker localRun instance

Data in master Run object

Data in worker Run object

See basic/B3b, B4b examples

Accumulables

- Classes for users “accumulables” management were added in 10.2 release
  - Accumulables are named variables registered to the accumulable manager, which provides the access to them by name and performs their merging in multi-threading mode
  - To better reflect the meaning of these objects, the classes base name "Parameter" used in 10.2 was changed in "Accumulable" in 10.3
- G4Accumulabe<T> - ready for use, for simple numeric types (double, int)
- Users can also define their own accumulables derived from the G4VAccumulable base class
  - Tested with std::map<G4String, G4int> used for processes counting in TestEm* examples

Accumulables (2)

- The accumulables are registered to G4AccumulableManager
  - Performs their merging in multi-threading mode according to their MergeMode
  - Provides the access to them by name
- Demonstrated in the basic examples B1 and B3a

Run Action + G4Accumulable

```
class MyRunAction : public G4UserRunAction
{
public:
  ...

  void AddEdep (G4double e)
  { fEdep  += e;  fEdep2  += e*e;};

private:
  G4double  fEdep;
  G4double  fEdep2;
}
```

sequential

MT (10.3)

```
#include "G4Accumulable.hh"
...
class MyRunAction : public G4UserRunAction
{
  public:
    ...
    void AddEdep (G4double edep)
    { fEdep  += e;  fEdep2  += e*e;};
    // ...
  private:
    G4Accumulable<G4double>  fEdep;
    G4Accumulable<G4double>  fEdep2;
};
```

Data accumulated
during event processing

Run Action + G4Accumulable (2)

```
#include "G4AccumulableManager.hh"
...
MyRunAction::MyRunAction()
: G4UserRunAction(),
  fEdep(0.),
  fEdep2(0.)
{
  //Register parameter to the parameter manager
  G4AccumulableManager* accManager = G4AccumulableManager::Instance();
  accManager->RegisterAccumulable(fEdep);
  accManager->RegisterAccumulable(fEdep2);
}

void B1RunAction::EndOfRunAction(const G4Run* run) {
  ...
  // Merge parameters
  G4AccumulableManager* accManager = G4AccumulableManager::Instance();
  acc->Merge();
  ...
}
```

The accumulable are initialized with a name (optional) and a value

The accumulables not created via the manager have to be registered to it

The call to Merge() may be not necessary in future

Migrating Sequential Geant4 Application to MT

Migration to MT

Migration of a sequential application to MT is a 5-steps process:

1. Move user actions instantiation to new G4UserActionInitialization class
2. Use G4MTRunManager in your main() function
3. Split DetectorConstruction::Construct() in two: SD and Field go in new method ConstructSDandField()
4. Use G4Run to accumulate run data, implement G4RunAction::Merge() method
5. If you use anywhere G4Allocator (typically for hits), transform them to be G4ThreadLocal

More details can be found Geant4 documentation and a short “howto” in the TWiki migration page:

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

Results With Geant4 10.x MT

Reproducibility

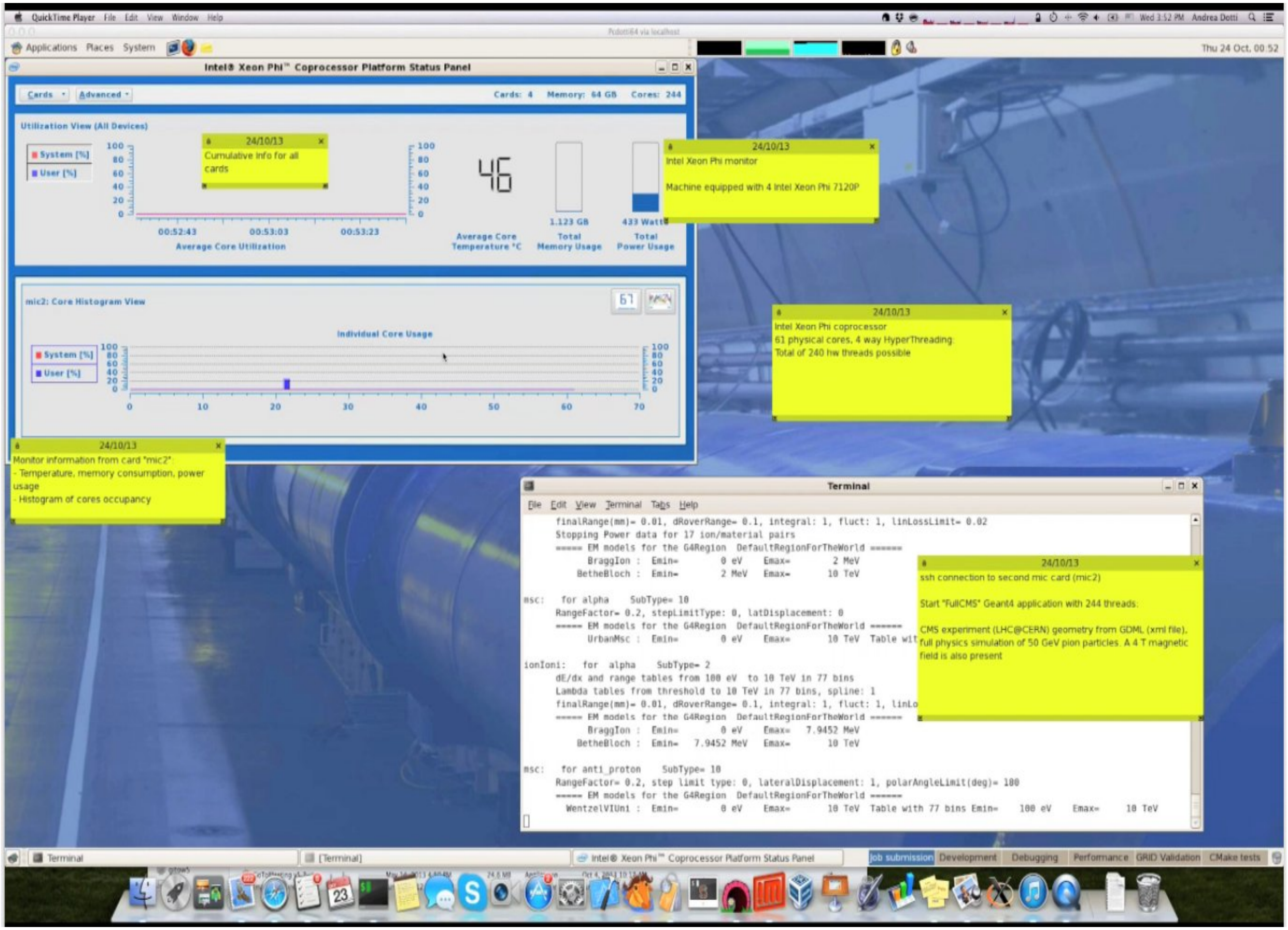
- Geant4 Version >= 10.0 guarantees strong reproducibility
- Given a setup and the random number engine status it is possible to reproduce any given event independently of the number of threads or the order in which events are processed

- Note: (optional) radioactive decay module breaks this in MT, Geant4 MT experts are currently working on a fix
- This does not mean the results are wrong!

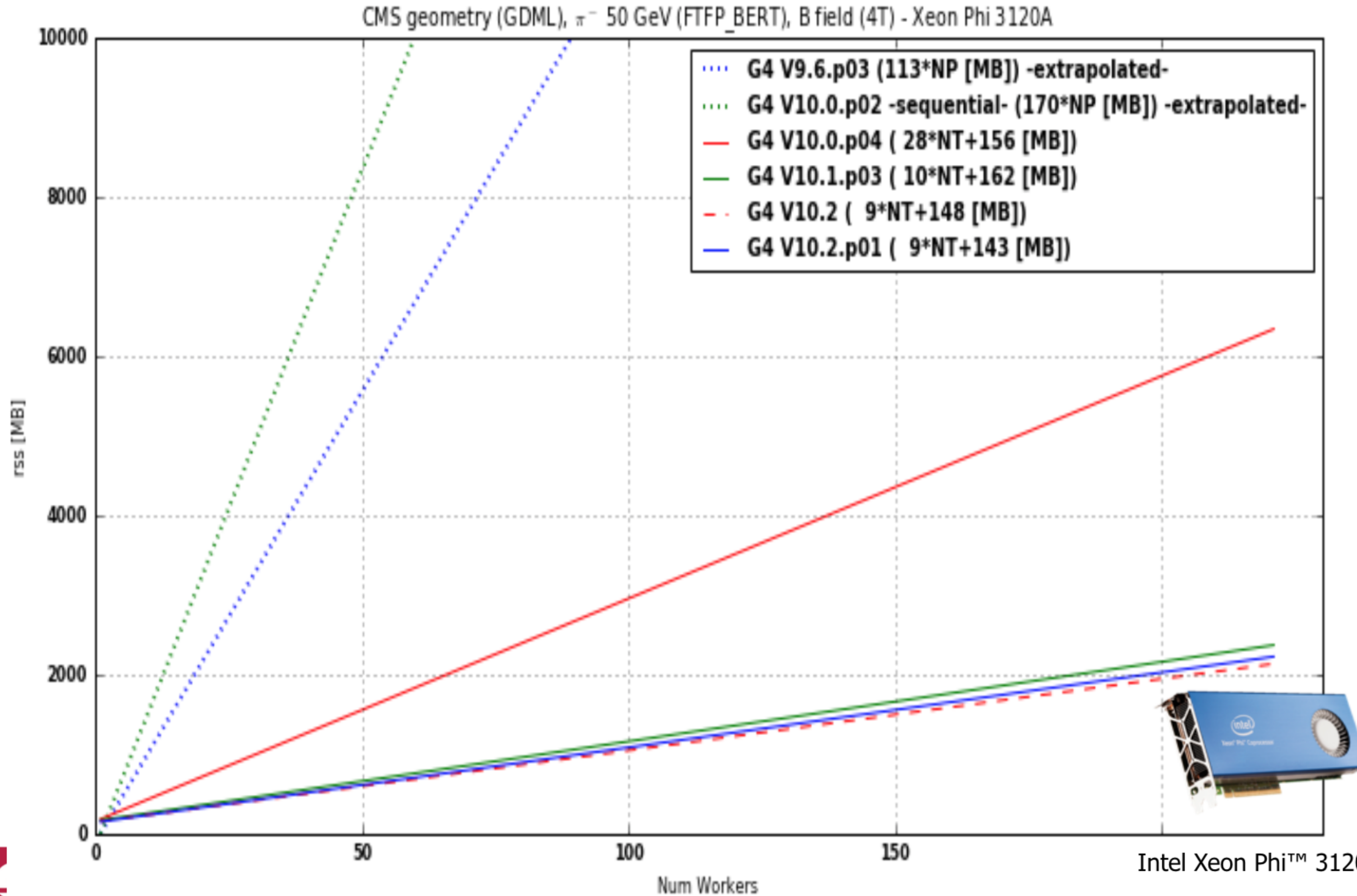
- Simulation results are equivalent between Sequential and MT

MIC Architecture

- Geant4 has been ported to compile and run on Intel Xeon Phi (aka MIC)
  - It requires Intel compiler (not free) and RTE
  - **61 cores (x4 ways hyper-threading), w/ max 16GB of RAM**

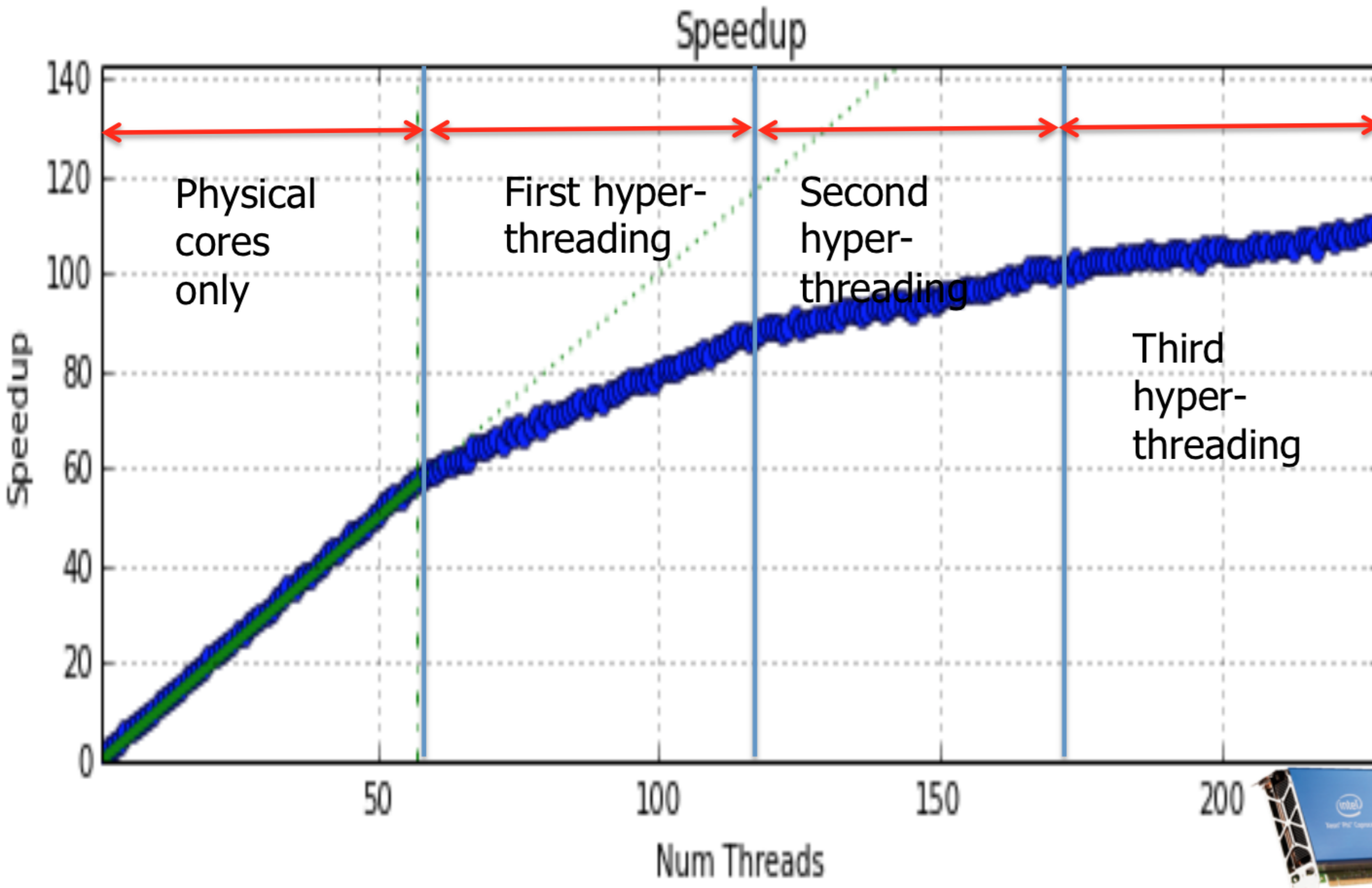


Memory consumption on Intel Xeon Phi



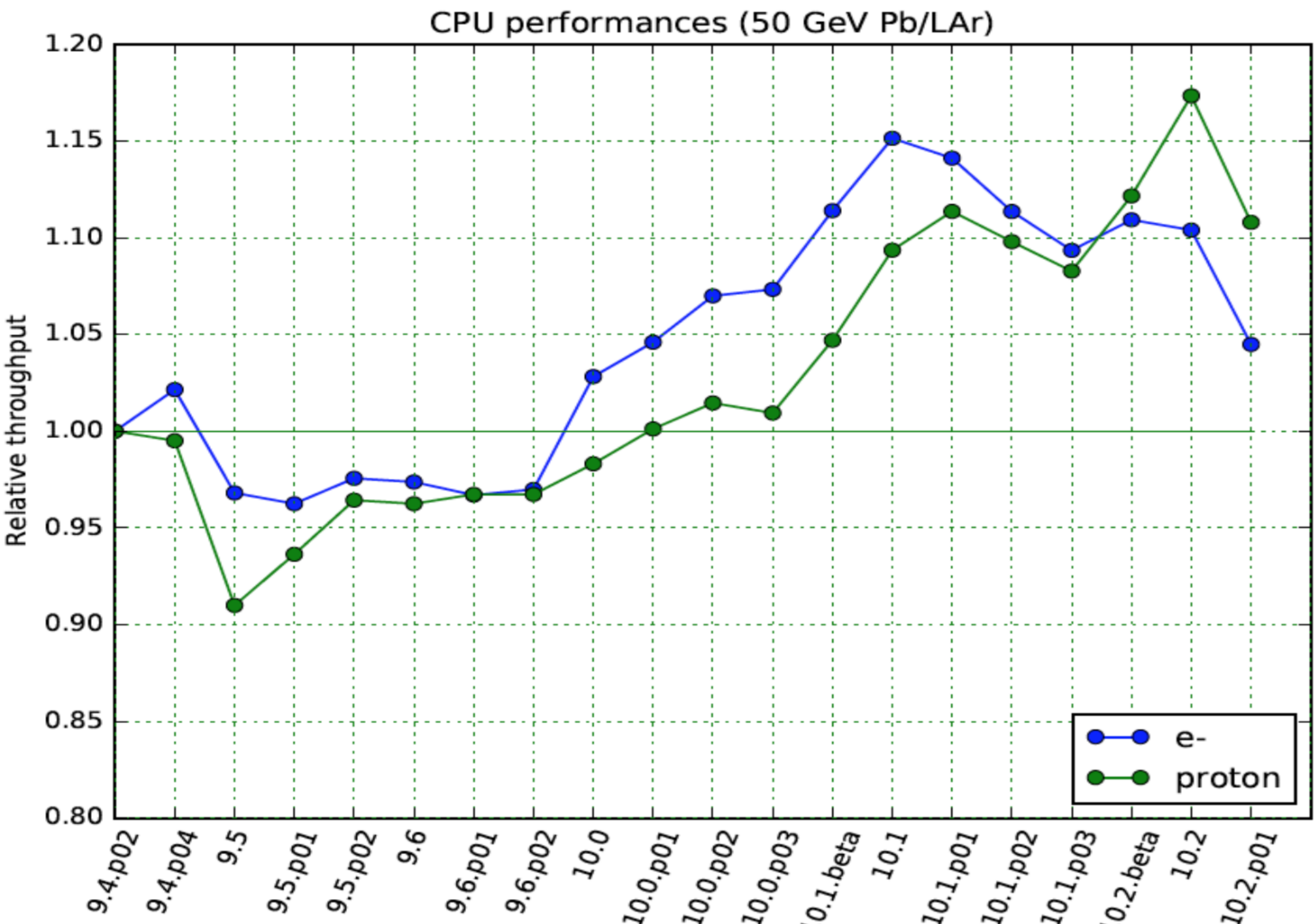
Intel Xeon Phi™ 3120A

Scalability on Intel Xeon Phi



Intel Xeon Phi™ 3120A

Throughput in sequential mode



Integration With External Frameworks

Heterogeneous Parallelism

- MPI
  - MPI works together with MT
  - The examples of MPI parallelism with Geant4 MT are provided in Geant4 examples/extended/parallel/MPI
  - New features in this category expected in the future: Geant4 MT experts are currently evaluating extensions !

- TBB
  - Intel Thread Building Block (TBB): task based parallelism framework
    - https://www.threadingbuildingblocks.org/
  - Freely available for Linux/Mac/WIN
  - Expression of interest by some LHC experiment
  - One example is provided in Geant4 examples/extended/parallel/TBB