Geant4 Version 10:

The challenges of the many-core computing era

A. Dotti for the Geant4 Multithreading Task Force

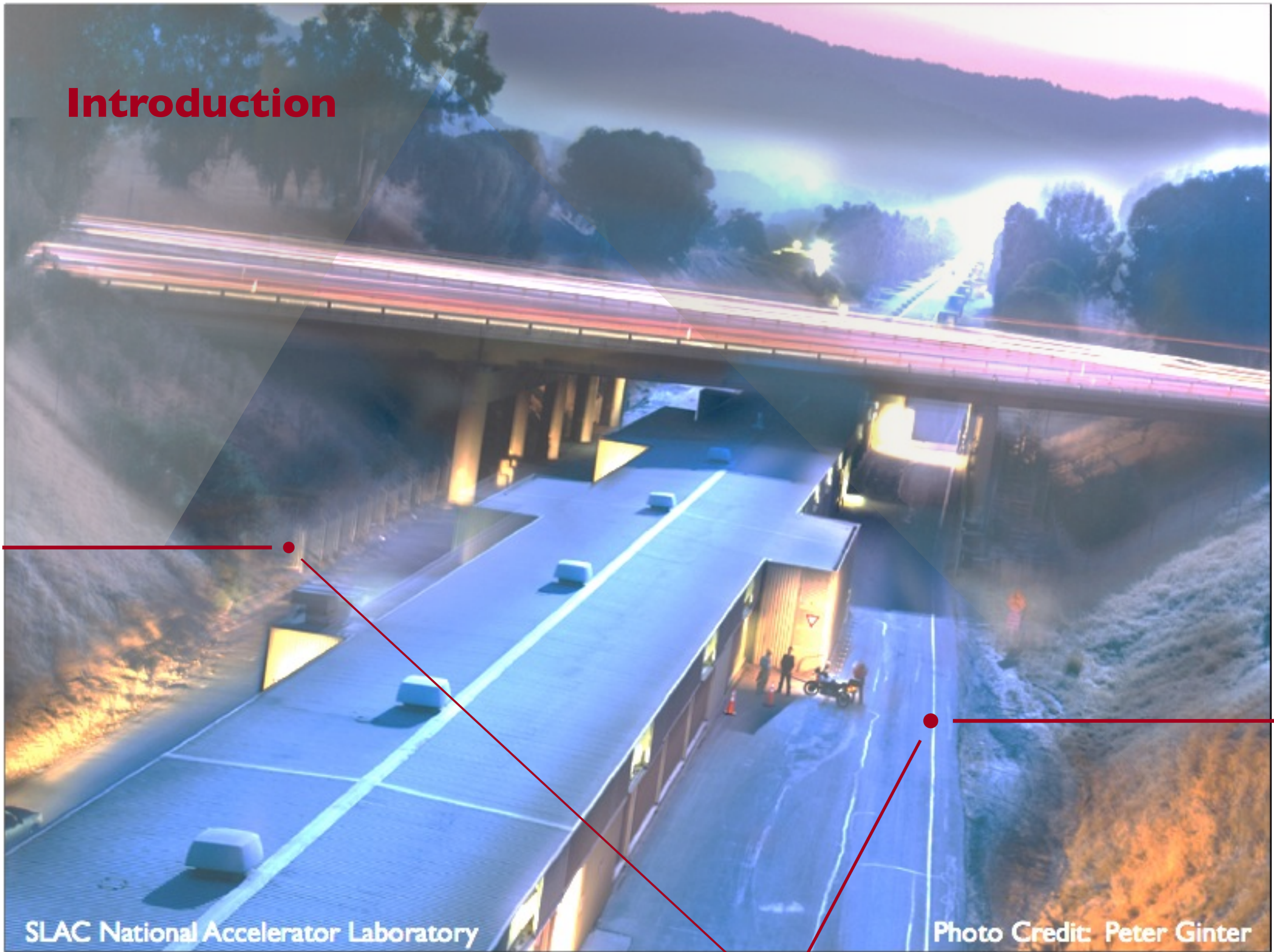
https://twiki.cern.ch/twiki/bin/view/Geant4/MultiThreadingTaskForce

18th Collaboration Meeting, Seville (SPAIN)

U.S. DEPARTMENT OF ENERGY
Office of Science

SLAC NATIONAL ACCELERATOR LABORATORY

Introduction



What's new in Geant4 Version 10?

SLAC

- Next version of Geant4 (December 2013) will be a **major release**
- Incudes several improvements
  - All categories will include important improvements
- **Main highlight is Multi-threading capabilities**
  - Introduce event-level parallelism

Geant4 MT: from prototypes to production

SLAC

- MT code integrated into G4

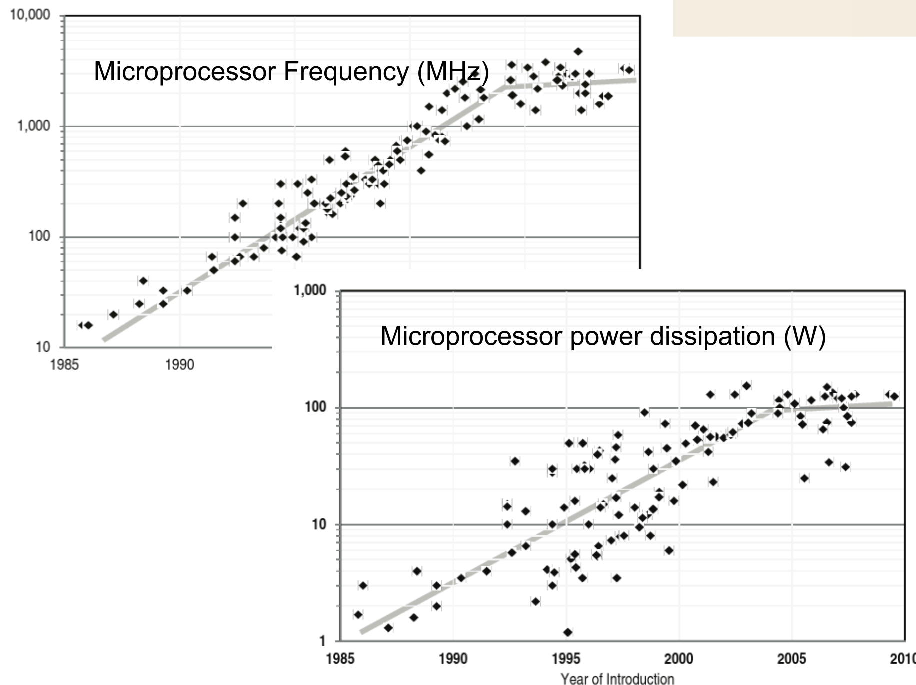
- Public release
- All functionalities ported to MT

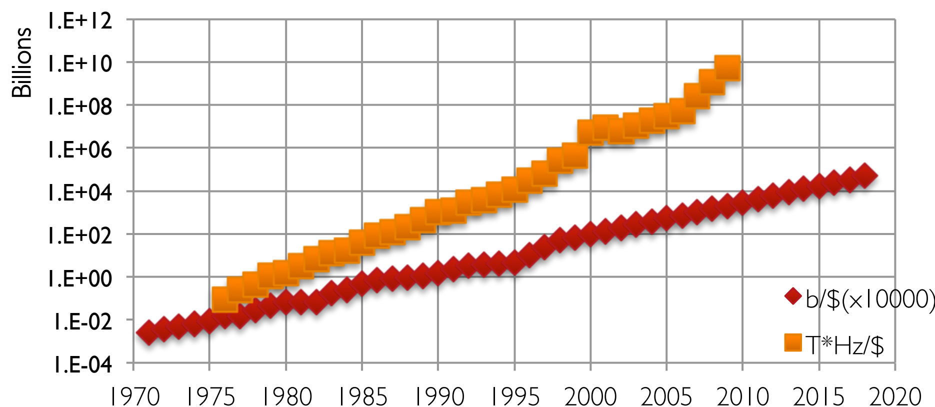
- Proof of principle
- Identify objects to be shared
- First testing

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

- Further Refinements
- Focus on further performance improvements

Why parallelism? (a reminder)





- 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:**
  1. Many core (double/2yrs?)
  2. Single core performance will not increase as we were used to
  3. Less memory/core
- New software models need to take these into account: increase parallelism

CPU Clock Frequecy 1and 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. 16
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 quantity and its access will limit number of concurrent processes running on single chip
- Add parallelism in the application code

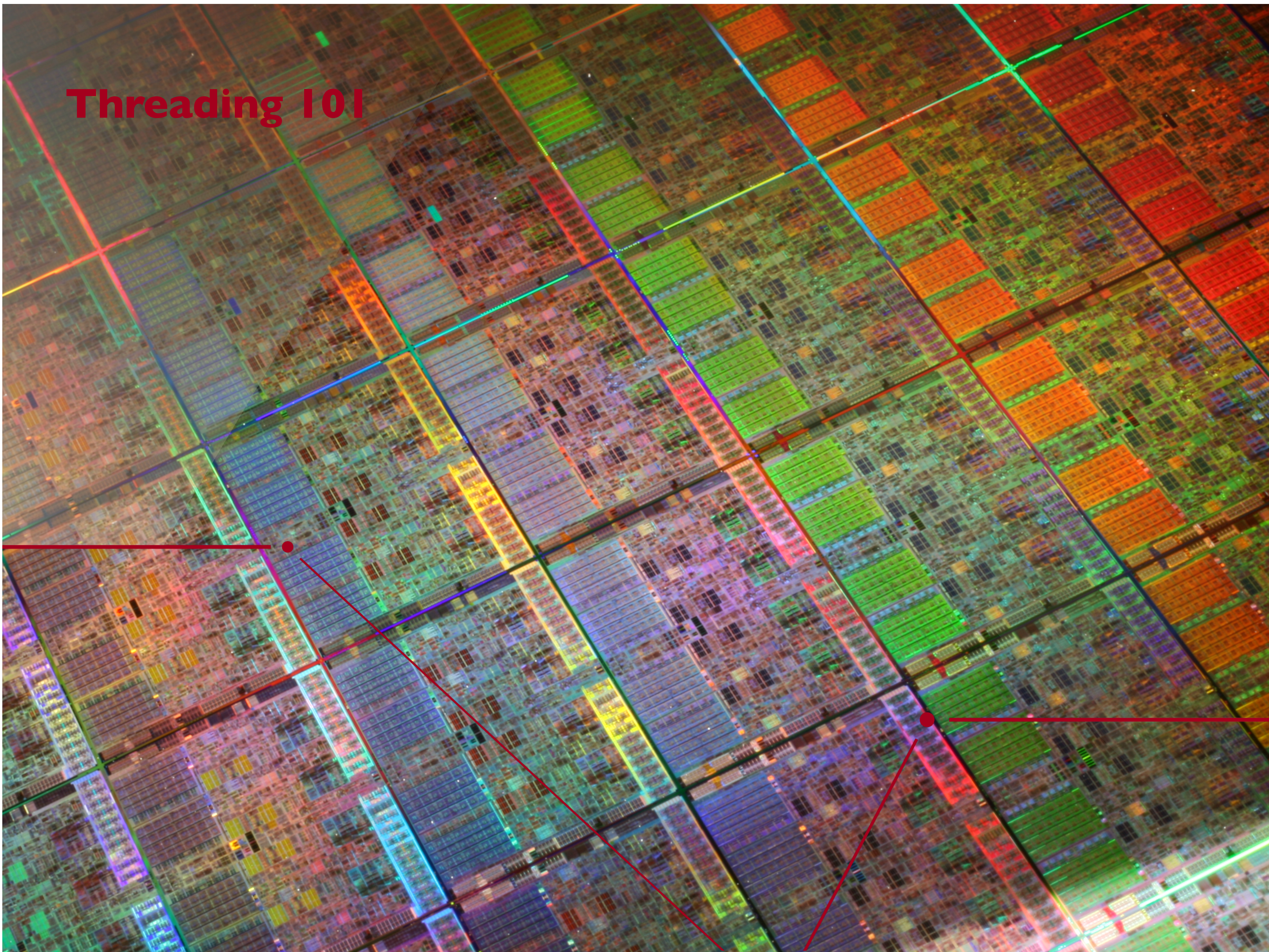
- Geant4 needs back-compatibility with user code and simple approach (physicists != computer scientists)
- Events are independent: each event can be simulated separately
- Multi-threading for event level parallelism is the natural choice

References

SLAC

- Very good Gene Cooperman's presentations at 16th Collaboration Workshop:
  - https://indico.fnal.gov/sessionDisplay.py?sessionId=12&confId=4535#20110920
  - Gene, as primary author of MT in Geant4 explains there very important details on thread-safety
- See also:
  - X. Dong et al. Euro-Par 2010 - Parallel Processing Lecture Notes in Computer Science Volume 6272, 2010, pp 287-303
  - X. Dong et al. Xin Dong et al 2012 J. Phys.: Conf. Ser. 396 052029
  - J. Apostolakis st al. : Upcoming SNA-MC2013 conference contribution
- Let me give you some notes, sorry I will not be so precise as Gene, but I'll do my best

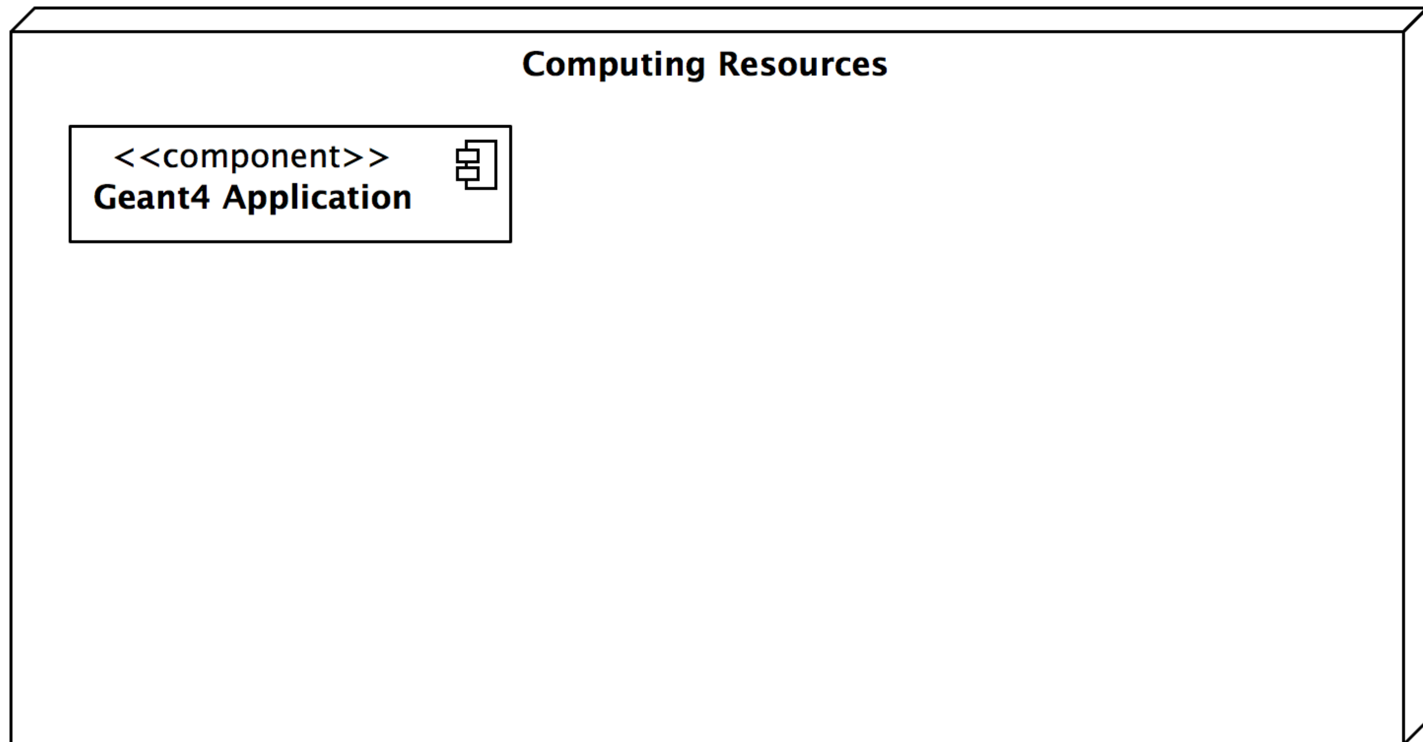
Threading 101



What is a thread?

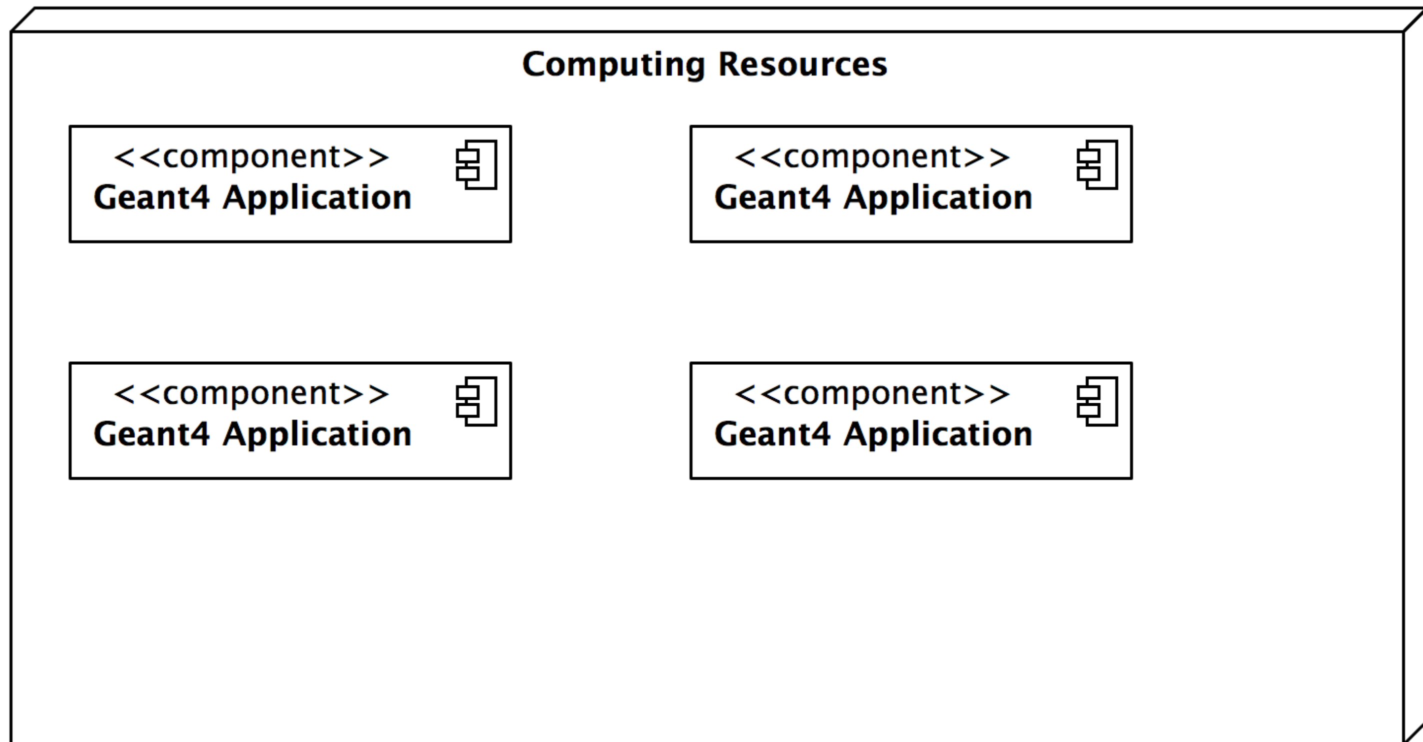
SLAC

Sequential application



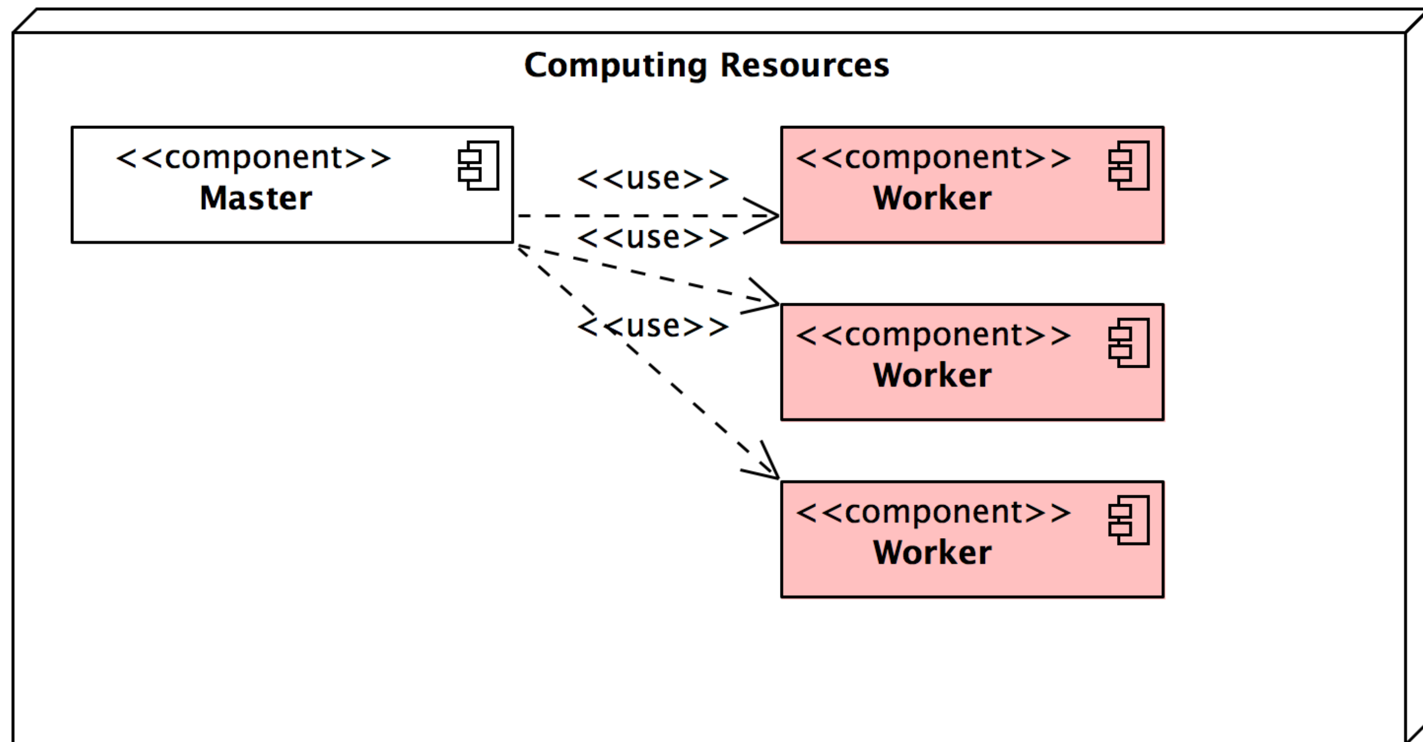
What is a thread?

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



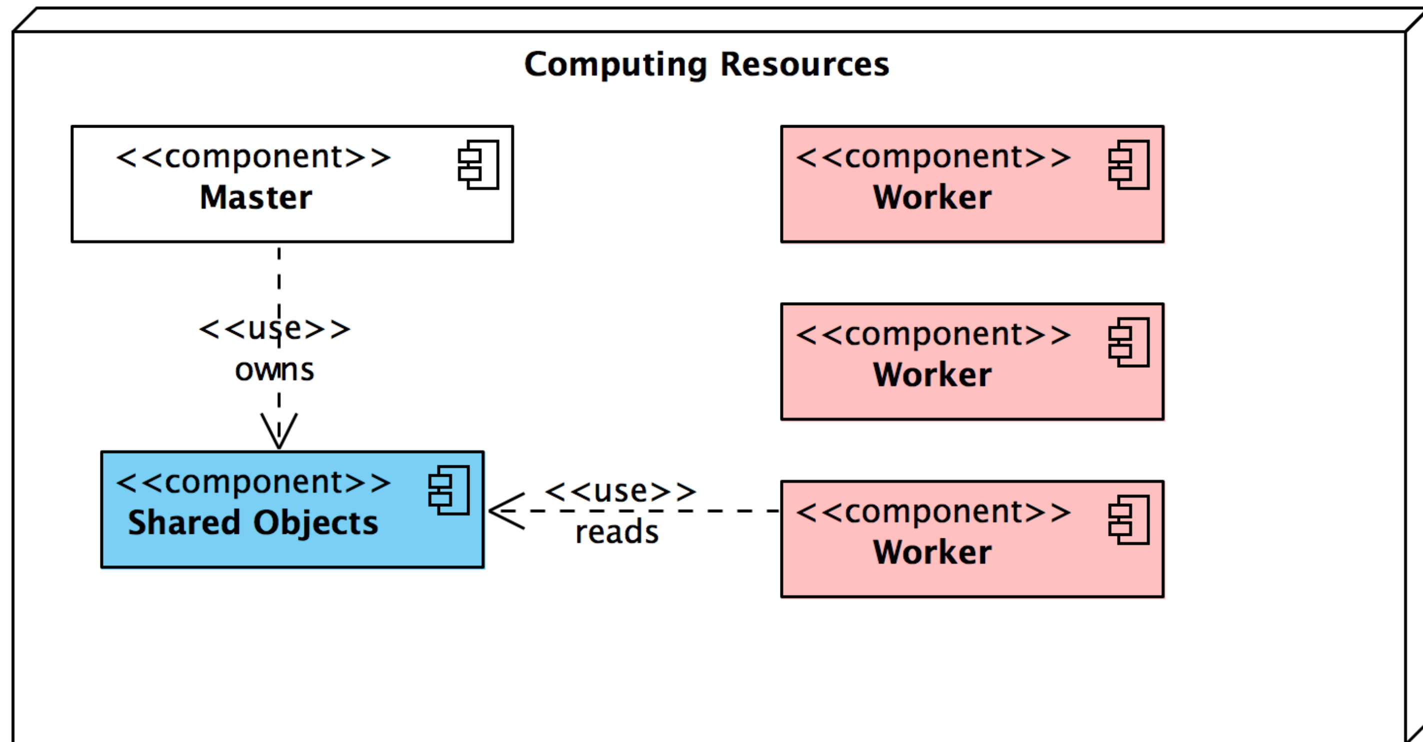
What is a thread?

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?

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



Data race

SLAC

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 = aSharedVariable;
       aSharedVariable = -1;
     } else {
       doSomethingElse();
       aSharedVariable = 1;
     }
     return result;
}
```

Data race

SLAC

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

```
double aSharedVariable;
```

Thread1

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

Thread2

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

Data race

SLAC

result is a local variable, exists in each thread separately not a problem, T1 starts arrives **here** and then is halted

Thread1

```
double aSharedVariable;
```

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

Thread2

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

Data race

SLAC

Now T2 starts and arrives **here**, the shared resource value is not yet updated, wrong behavior

```
double aSharedVariable;
```

Thread1

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

Thread2

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

Data race

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)

```
double aSharedVariable;
```

Thread1

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

Thread2

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

Data race: TLS

SLAC

- Does aSahredVariable needs to be shared between threads?
  - if not, minimal change required, each thread has its own copy
  - No memory performances, some CPU penalty
- General rule in G4: do not use unless really necessary

Thread1

```
double __thread aSahredVariable;

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

Thread2

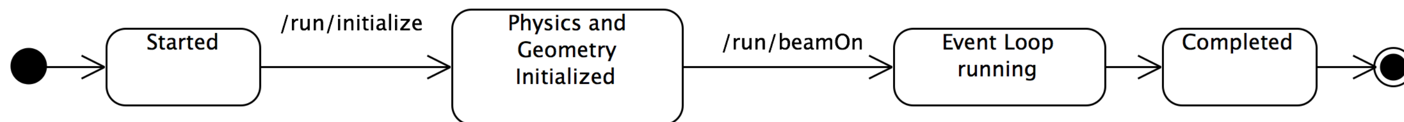
```
 double __thread aSahredVariable;

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

Data race: master/worker model

SLAC

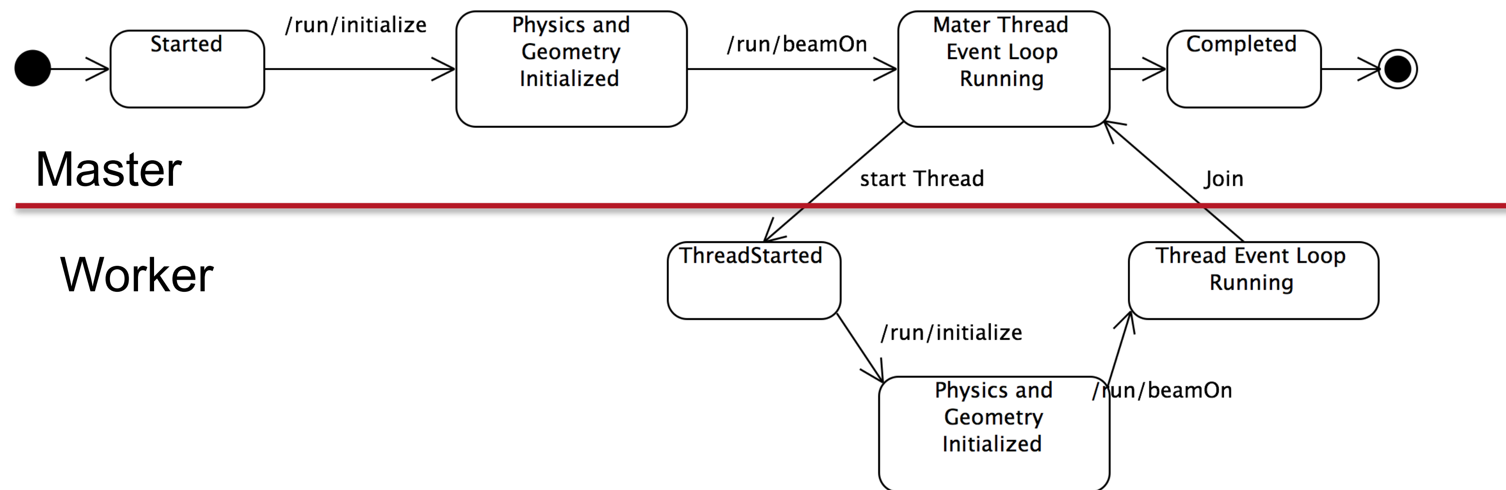
- As we said there is no problem if threads only read shared resource (see Gabriele's presentation on Plenary Session 2 on Valgrind for definition of data race)
- A G4 (with MT) application can be seen as simple finite state machine



Data race: master/worker model

SLAC

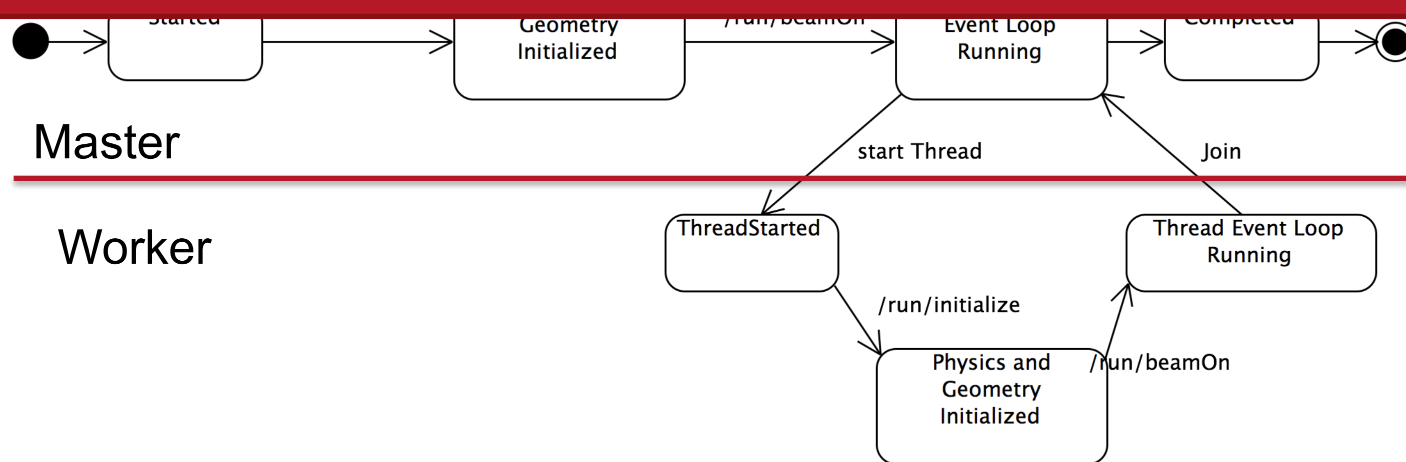
- Threads do not exists before /run/beamOn
- Master can write shared memory without problems
- When threads start they cannot anymore change the shared memory
  - No need to lock



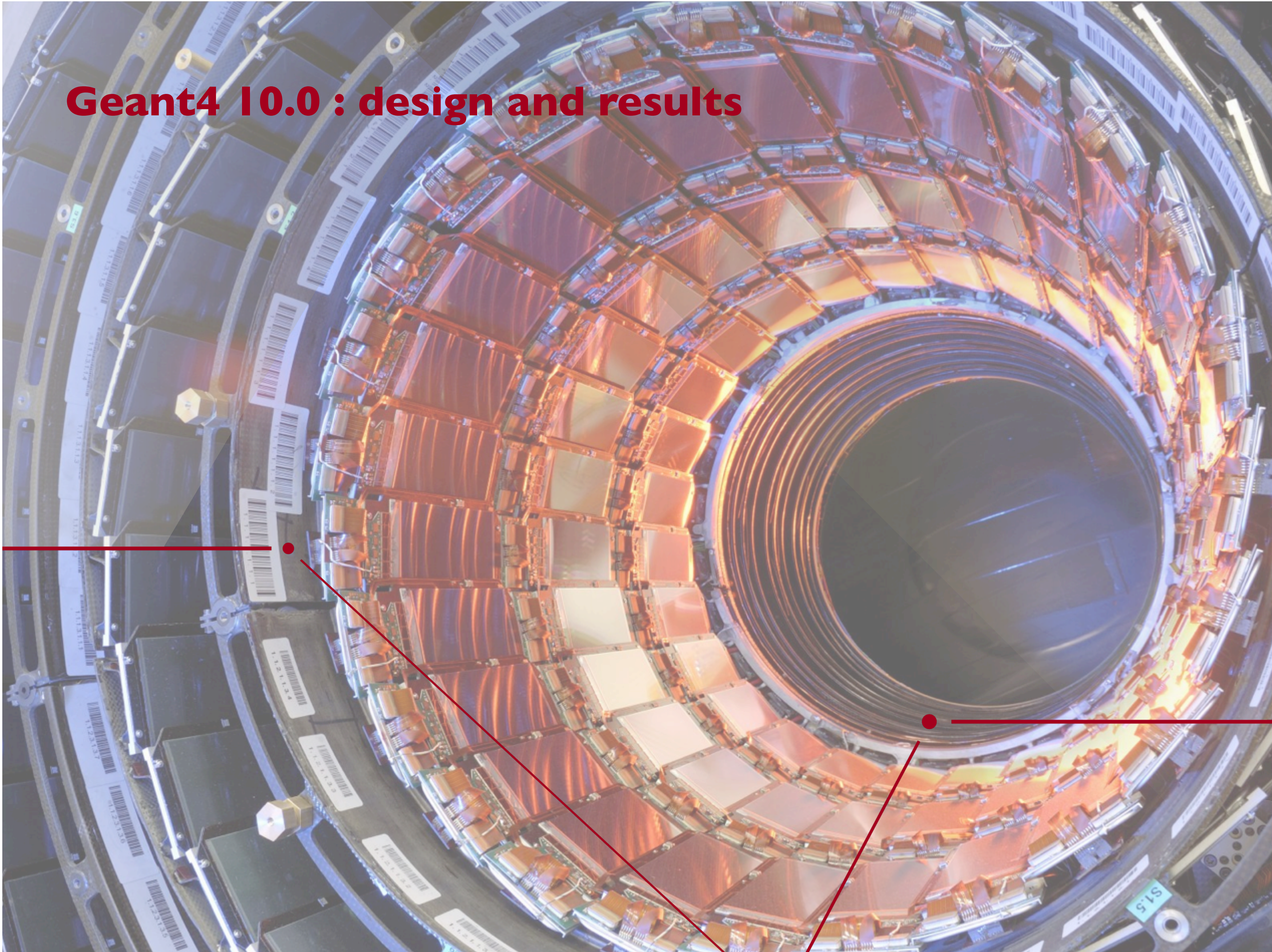
Data race: master/worker model

- Threads do not exists before /run/beamOn
- Master can write shared memory without problems
- When threads start they cannot anymore change the shared memory

A Question to the collaboration:
How can we improve in our code for MT?
Do we need an ad-hoc tutorial?



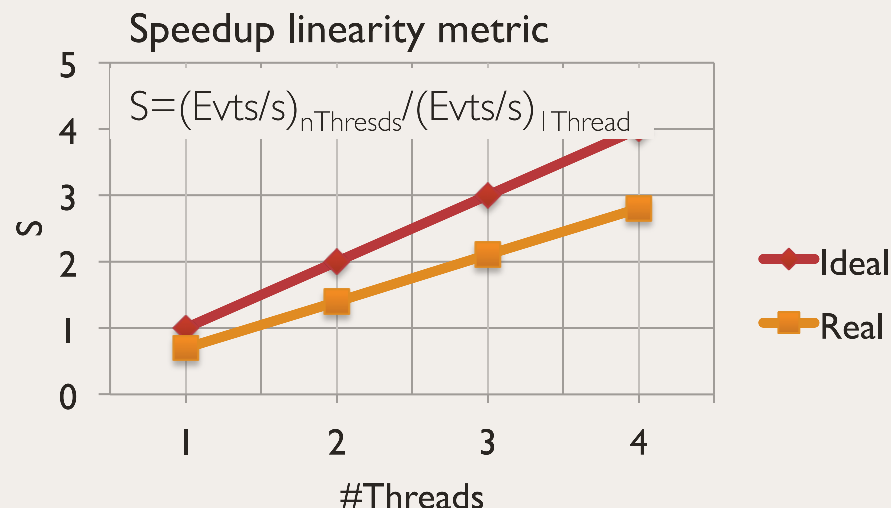
Geant4 10.0 : design and results



Geant4 Multi-threading: event level parallelism

SLAC

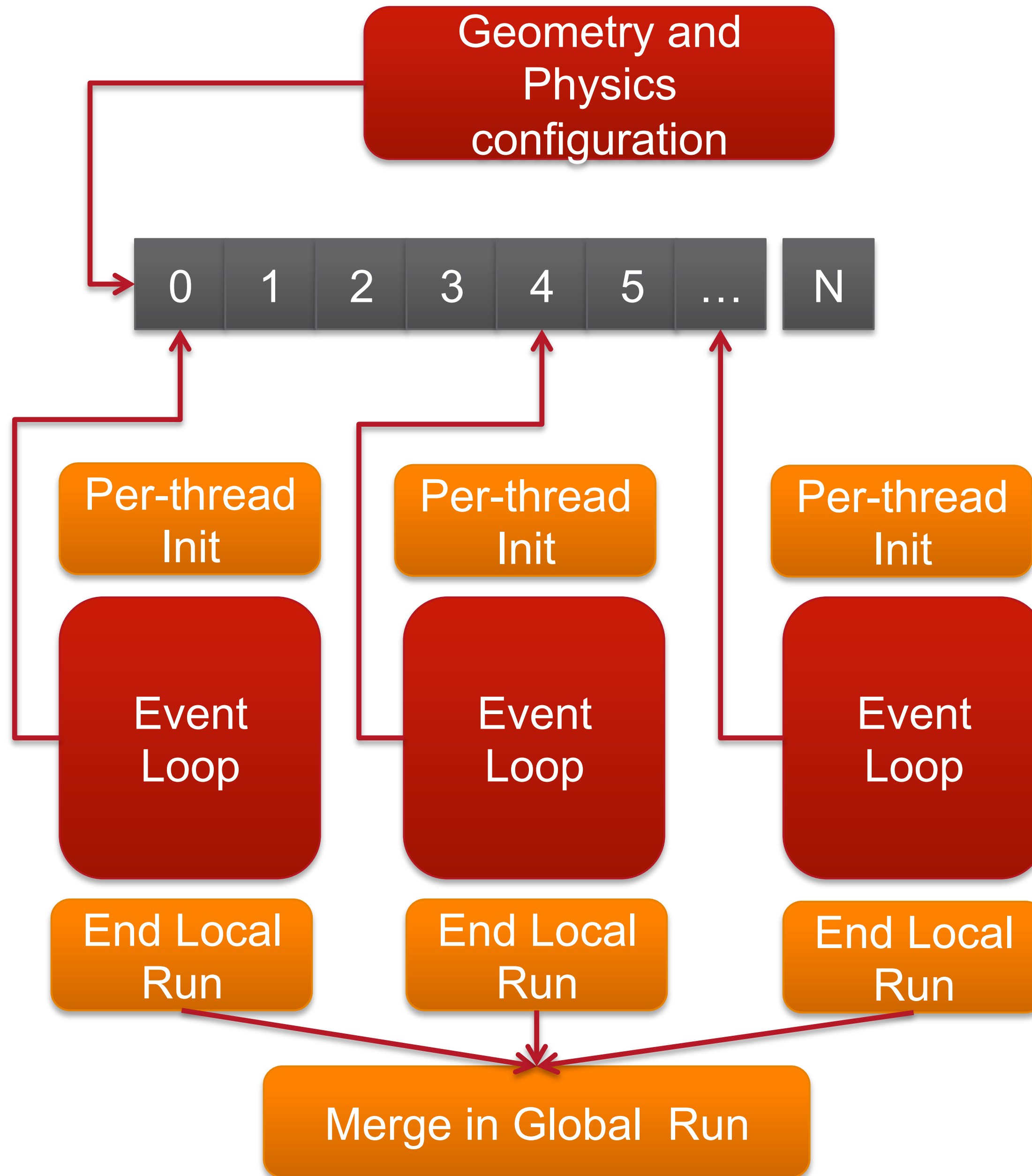
- **Design to minimize changes in user-code**
  - Maintain API changes at minimum
- Focus on **"lock-free" code**: linearity of speed-up (w.r.t. #threads) is the metric we are currently concentrating on (then we'll optimize absolute throughput)
  - Good results obtained for both metrics anyway (see later)
- 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



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

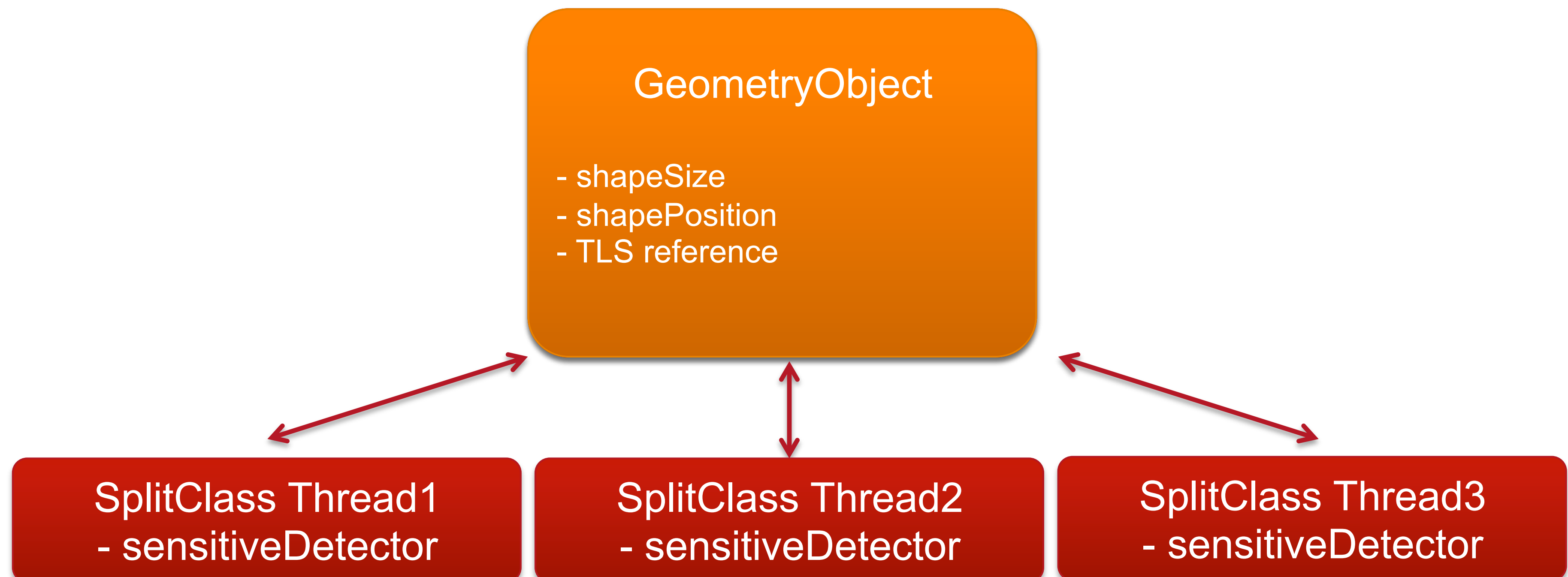
Threads compete for next event to be processes (new in ref-08)

Command line scoring and G4tools automatically merge results from threads

Basic design choice

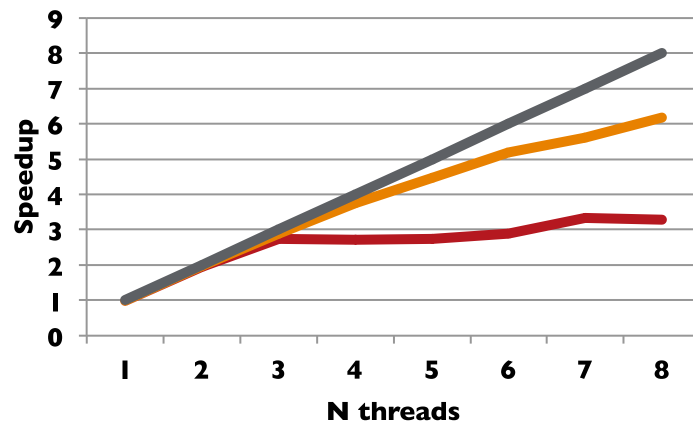
SLAC

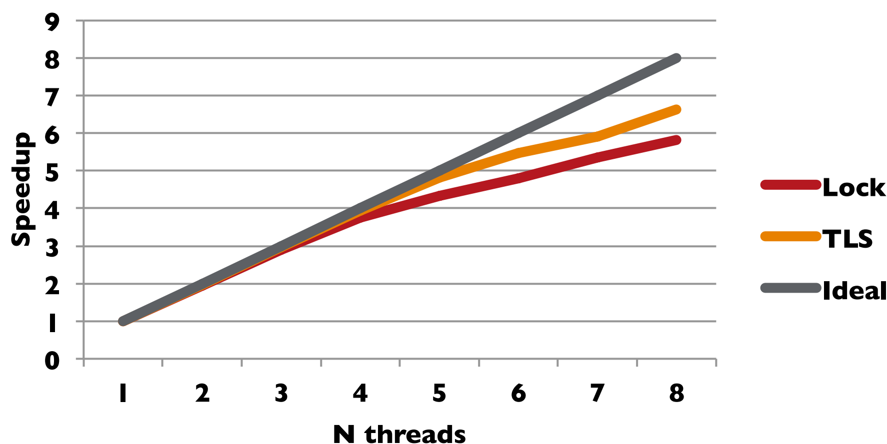
- Thread-safety implemented via **Thread Local Storage**
- "Split-class" mechanism: reduce memory consumption
  - Read-only part of most memory consuming objects shared between thread
  - Geometry, Physics Tables
  - Rest is thread-private



Thread Local Storage

SLAC





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

Important note: static Vs G4ThreadLocal Vs split-class

SLAC

- A **static** class data field is shared among all class instances and all threads
- A **static G4ThreadLocal** is shared among all class instances but not among threads
- The **split-class** mechanism allows for sharing among threads but not among instances

How to migrate code

SLAC

- Documentation is currently in twiki:
  - https://twiki.cern.ch/twiki/bin/view/Geant4/MultiThreadingTaskForce
  - Will be moved to docbook documentation

- See Ivana's presentation next for migration of user code examples

G4MTRunManager

SLAC

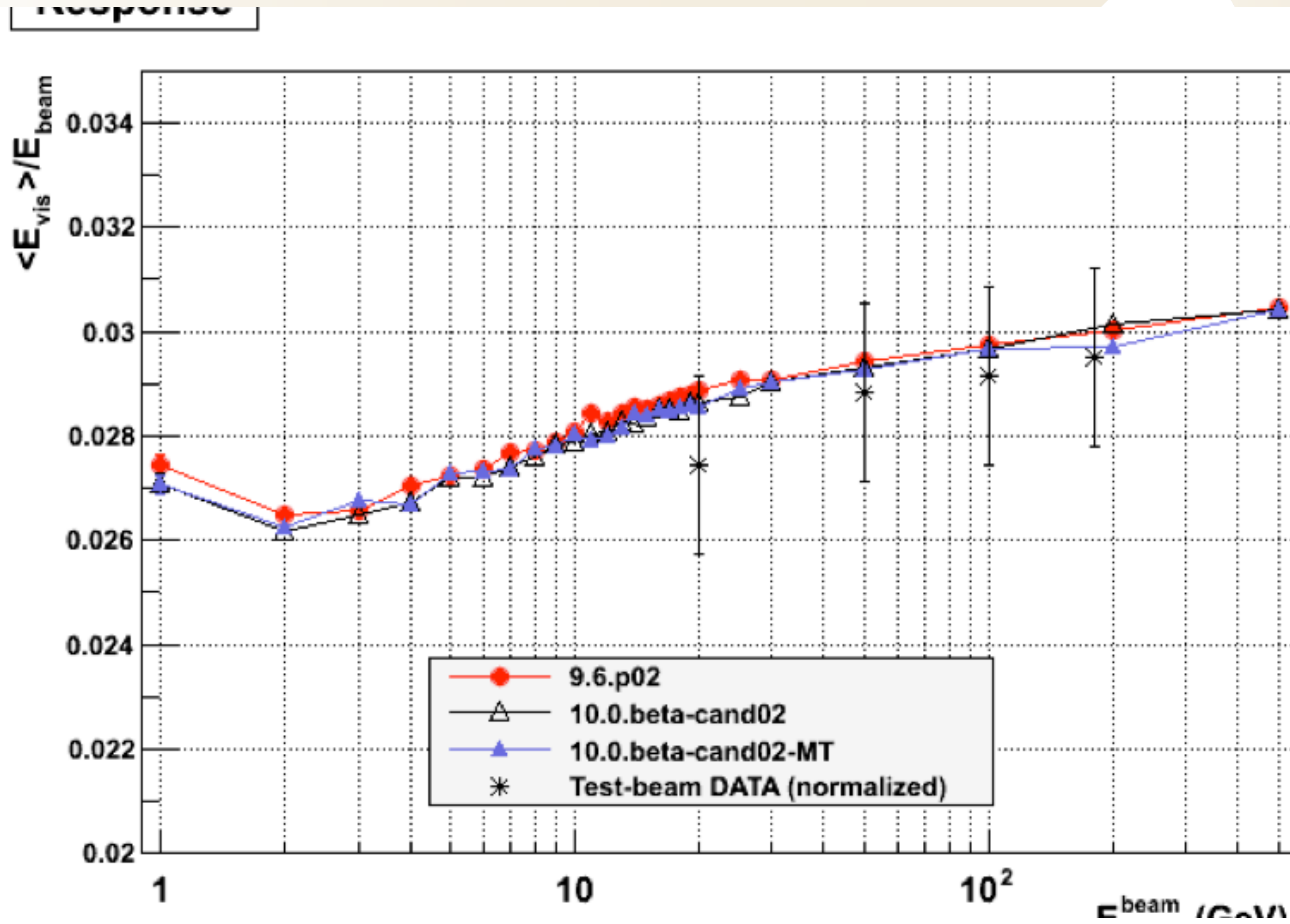
- In brief:
  1. Replace G4RunManager with G4MTRunManager
  2. Create new user-initialization G4VUserActionInitialization (n.b. G4RunManager extended to support this)
  3. New UI commands (ignored if used in sequential mode):
     - `/run/numberOfThreads` *`nThread`*
     - `/run/eventModulo` *`nEvents`* (under development, advanced)
     - `/control/cout/setCoutFile` *`fileName ifAppend`*
     - `/control/cout/setCerrFile` *`fileName ifAppend`*
     - `/control/cout/useBuffer` *`flag`*
     - `/control/cout/prefixString` *`prefix`*

Results: Physics Performances

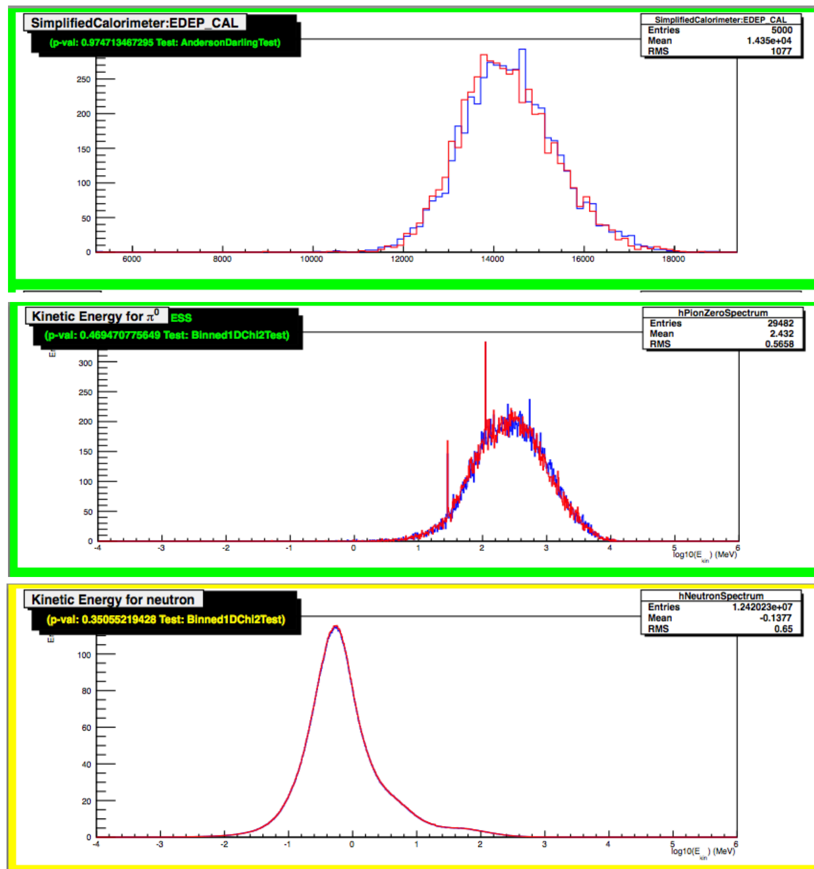
SLAC

- Strong reproducibility:
  - single events of MT and SEQ versions of the same application start with the same RNG seed. Verify after event engine status is the same. Ref-08 + bug-fix 100% reproducible
- Statistical Tests with SimplifiedCalorimeter 1:
  - Compare full "GRID style" validation between SEQ application w/ SEQ G4 libs and SEQ application w/ MT G4 libs
  - Compare MT application with SEQ application w/ SEQ G4 libs
- See Soon's and Alberto's presentation next

MT libs Vs SEQ libs



MT Application Vs SEQ Application



- Full showers simulated
  - All physics process (excluding Low-E) included
- Observables (secondaries spectra, energy deposits) are statistically equivalent
- Need to test with a Low-E application: volunteers?

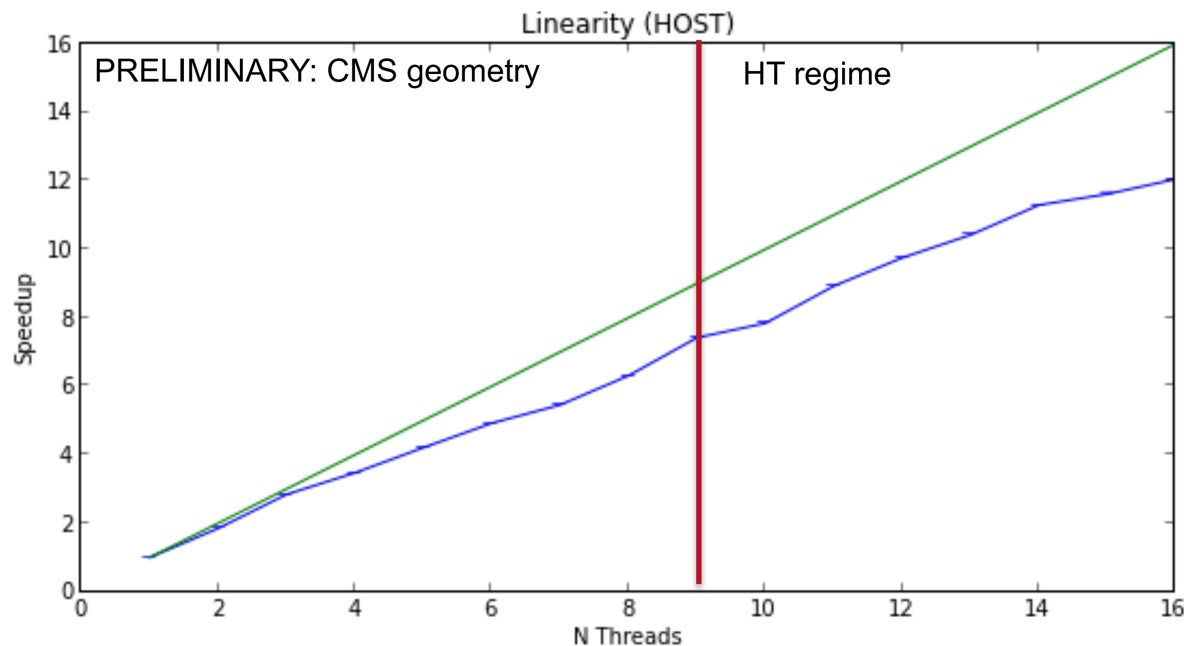
CPU performances

SLAC

- Please refer to session Parallel 3A – Computing Performances for details
- Only few plots here to give fast overview
- Obtained with 10.0.beta
- G4 computing performances page kept up-to-date

Results: Linearity

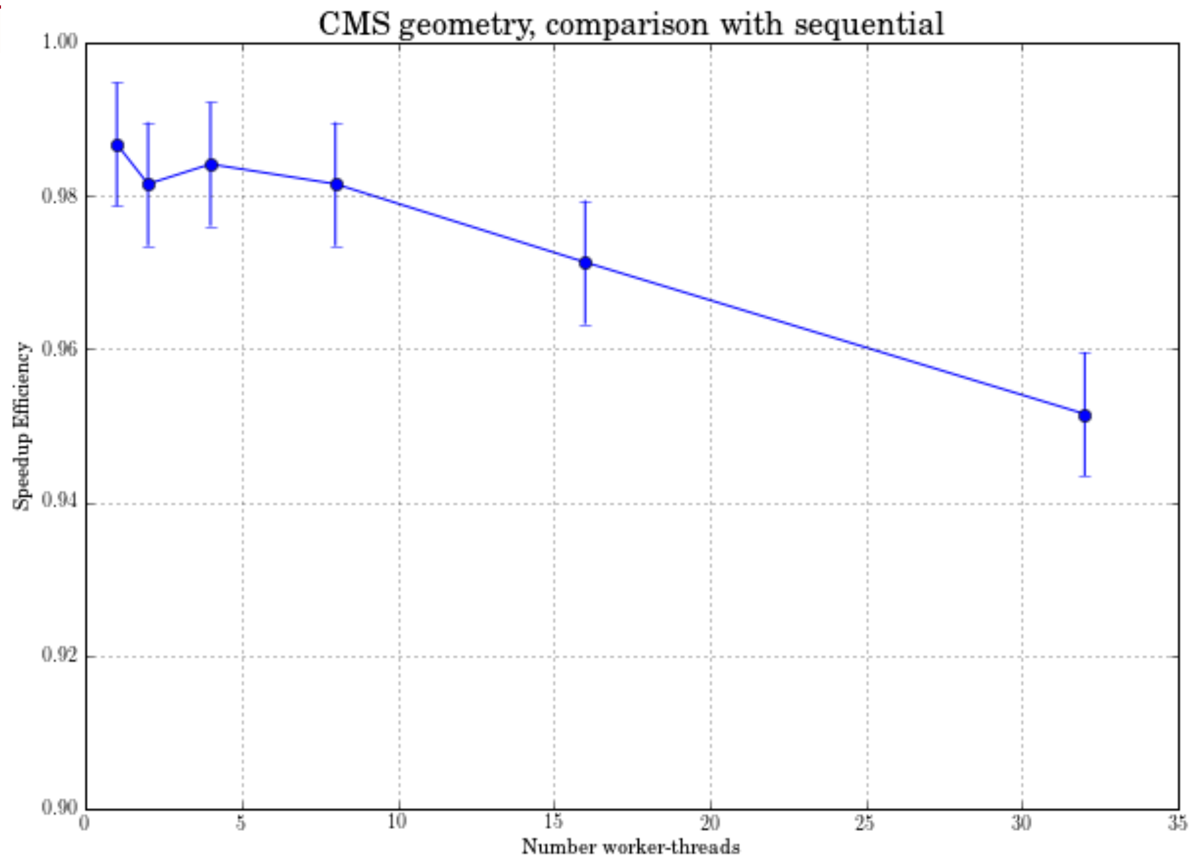
SLAC



Intel(R) Xeon(R) CPU L5520 @ 2.27GHz

- **Good linearity demonstrated**
- Efficiency w.r.t. perfect linearity 90% (80% in HT)
- **Out-of-the box** Geant4 with MT=ON
- Further improvements expected
  - Test use of thread-private malloc library
  - Reduce use of TLS when not necessary
  - See *Euro-Par2010, Part II LNCS6272, pp.287-303*: full efficiency recovered

Results: Absolute performances



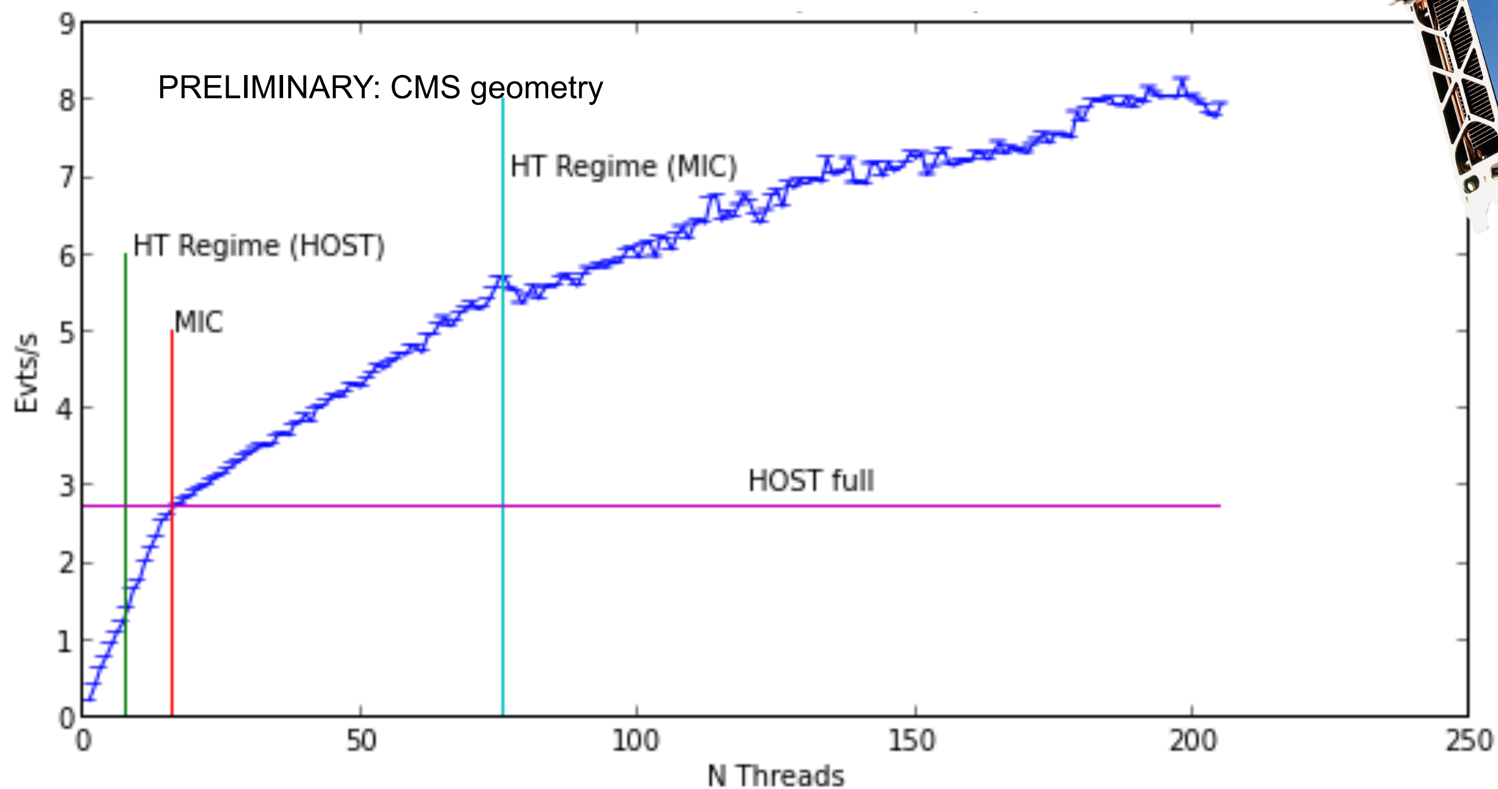
5%

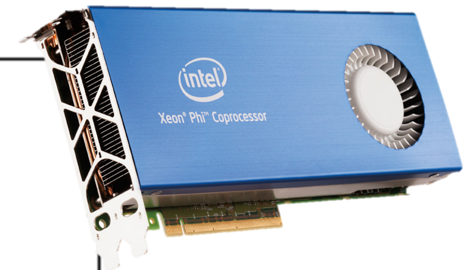
- Compare a single MT application with N threads w.r.t. N copies of the same application (assuming infinite memory)
- N=1 result is very important: it shows the overhead due to MT "machinary" (e.g. TLS perfomance loss)
- Only 1% loss: early prototype penalty fully solved

Results: large number of threads (MIC architecture)

SLAC

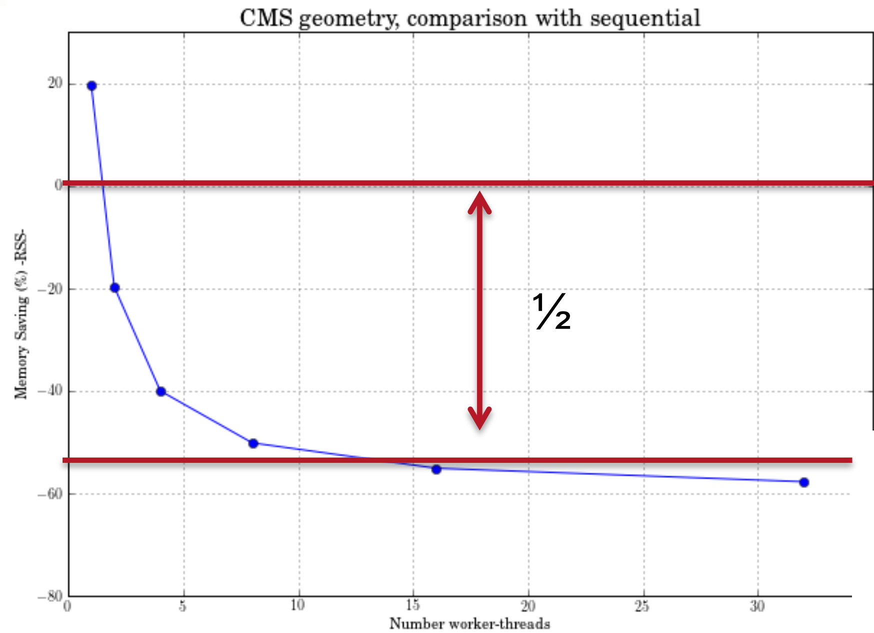
- **Hybrid mode: Host + Intel Xeon Phi coprocessor**
- First look at total throughput: Evts/s
  - **Very good results: factor ~x3 in events produced w.r.t. host only**
- Up to 8 MIC cards can be hosted by single host
- Need to coordinate processes (e.g. MPI, intel-MIC-offload)
- Very different initialization time between host and MIC

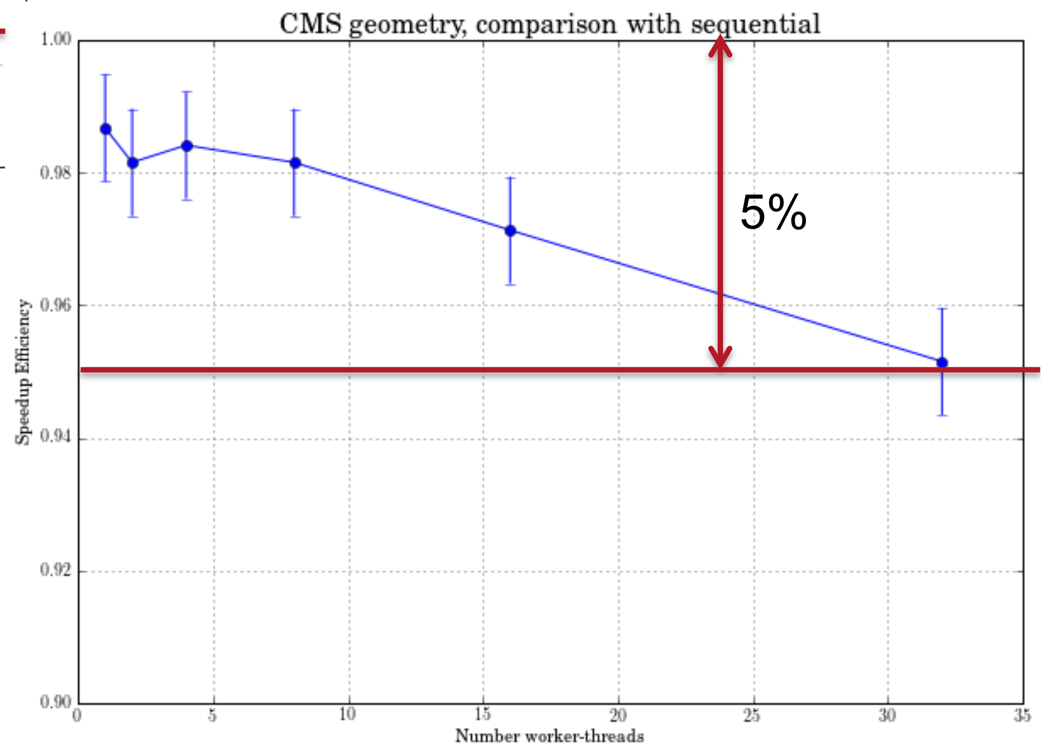




MC 2013 Paper: comparison to sequential

SLAC





Results summary

SLAC

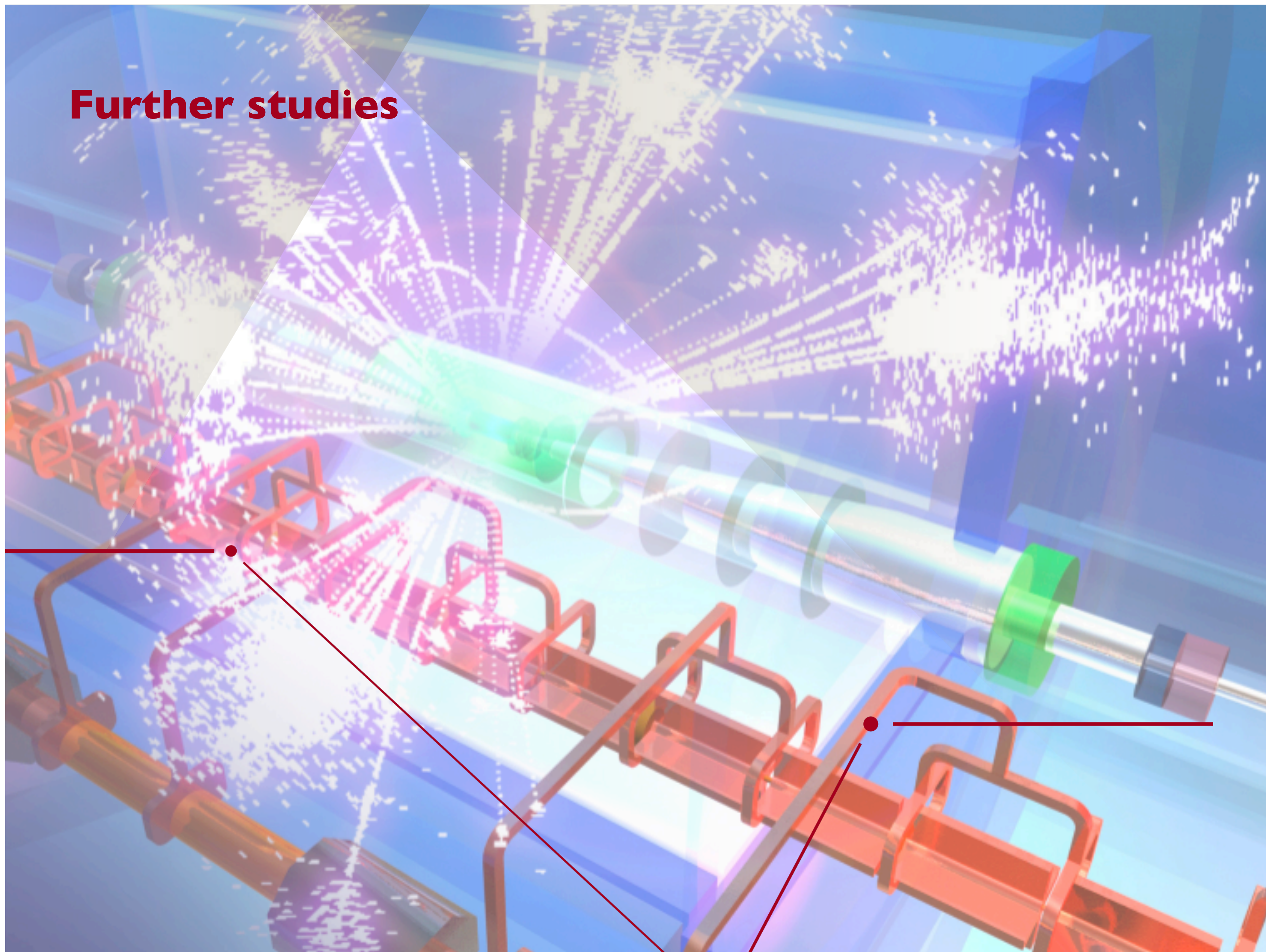
- Very good and promising results achieved so far
- Physics:
  - Sequential and MT are identical in physics output
  - Reproducibile has been achieved (need to test each tag, though)
- CPU Performances:
  - Linearity of speed-up obtained with >90% efficiency
  - Very promisng results on large number of threads O(100) with >80% efficiency
- MEM Performances:
  - Good reduction of memory usage (20-80MB/thread)
  - Example (FullCMS application): 50% memory usage @10Threads (e.g. 10 threads use ½ memory w.r.t. 10 spawned processes)

Status of Geant4 Version 10.0

SLAC

- **Beta version announced on time**
- All main functionalities have been ported to MT
  - **Only one limitation remains:** Visualization is not yet fully functional (no event-by-event visualization during the event loop)
  - WIN7 not yet working
- **Migration of user code is relatively simple:** existing examples and tests can be migrated in few hours, complex user applications will require more work

Further studies

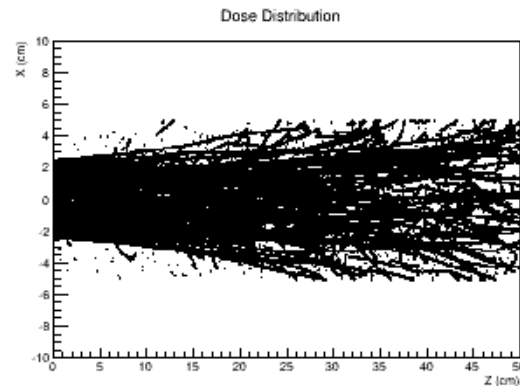


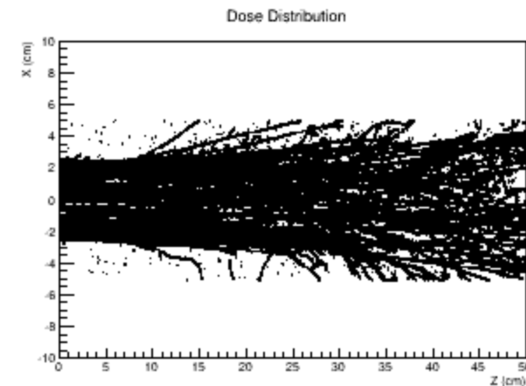
Heterogeneous parallelism: MPI based G4MT

SLAC

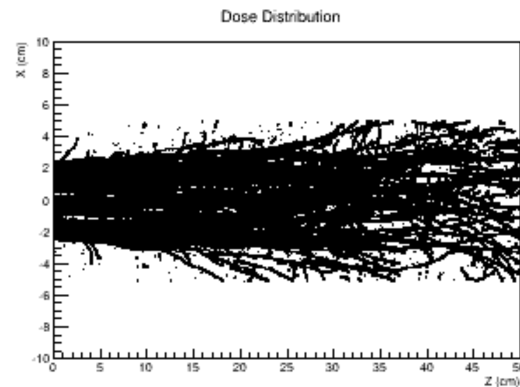
- **MPI based parallelism** already available in Geant4
  - MPI works together with MT

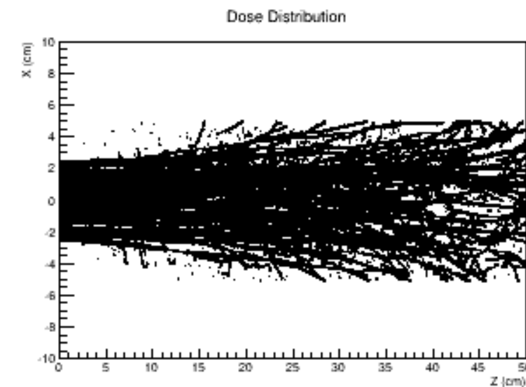
Example:
4 MPI jobs
2 threads/job
MPI job owns histogram





Next Step:
Host + MIC simulation
Based on MPI





Improve start-up time: Checkpointing

SLAC

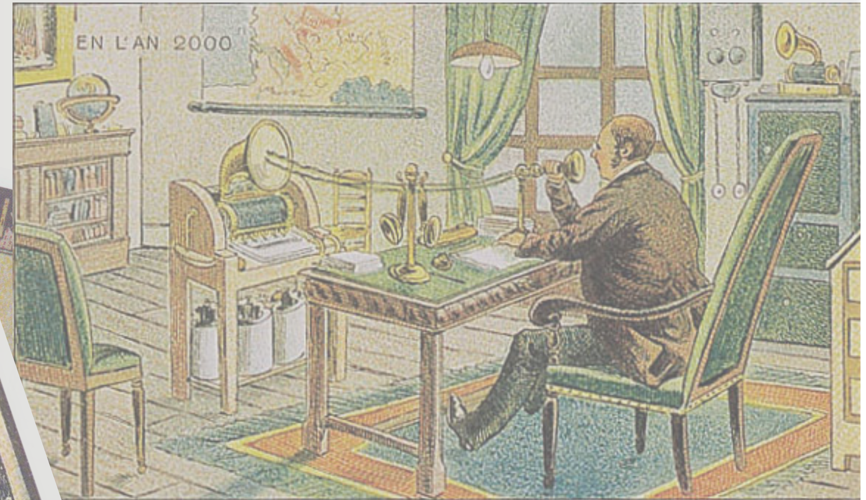
- Each parallel application has sequential part (G4MT: geometry definition, physics initialization, threads creation and initialization)
- With large number of threads **sequential CPU-time** can become important fraction of total run-time (especially true for accelerators)
- **Substantially reduced via checkpoints** (dump of program image to disk at specific points. A controlled "core-dump"), **restart from checkpoint image**
- **DMTCP**: checkpointing for multi-threaded programs (G. Cooperman et al)
- Tested with success for Geant4 MT (in collaboration with CMS experiment). On Xeon Phi:
  - Start CMS simulation, checkpoint at first event (5 mins initialization)
  - Replay/restart application from checkpoint file (10 seconds restart)
  - Interesting possibility for production system: copy checkpoint image on many machines (or accelerators), start clones of process, re-seed processes

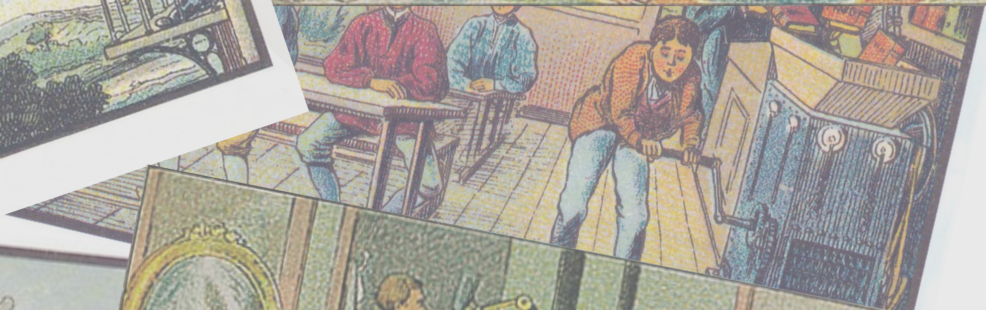
Work in progress

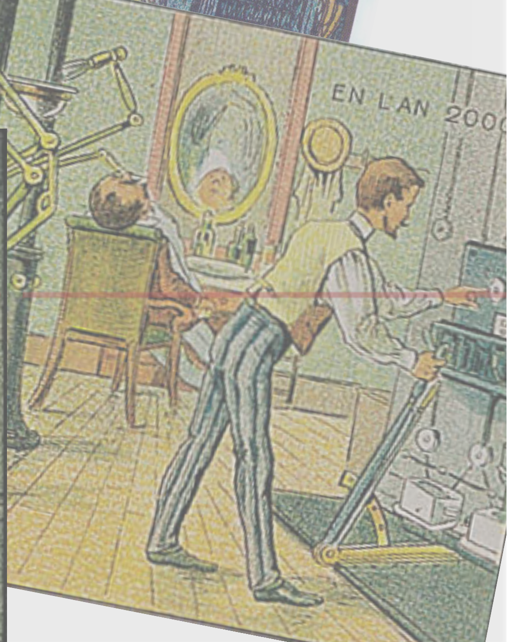
SLAC

- **Intel Thread Building Block (TBB)**: task based parallelism framework (expression of interest by some LHC experiment)
  - TBB works with G4MT: provide one or two examples for final release
- **ThreadPrivate malloc library** (TPMalloc – G. Cooperman et al): each thread has its own heap, remove hidden locks in "new/delete". Target to be provided as "external optional component".
- Review APIs with feedback from early users: **further simplify user-code migration**
- Identify and solve hotspots, **improve performances**
- Fully functional Visualization drivers

Future directions







"The only thing we know about the future is that it will be different."
Peter Drucker

Accelerators

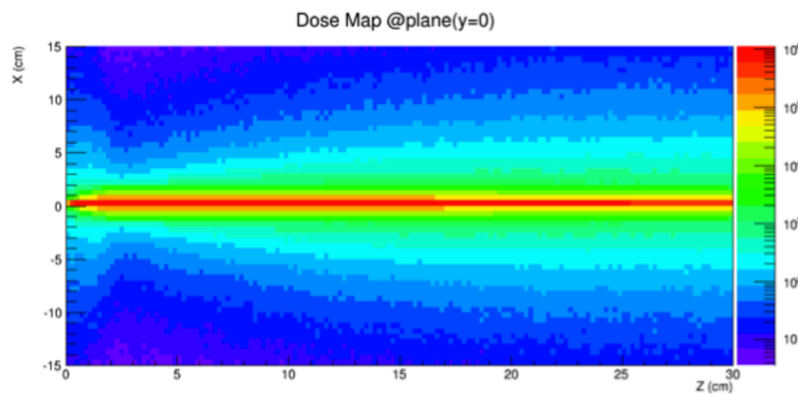
SLAC

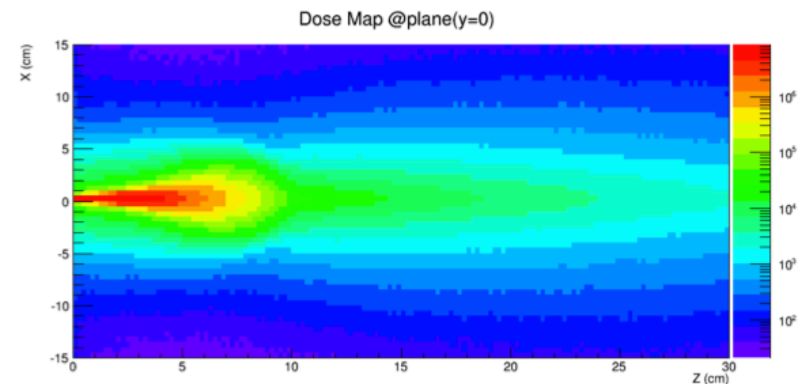
- **Increasing interest in accelerators**: two main technologies GPGPU / MIC architectures
  - First two Top500 supercomputers based on accelerators (#1: Thianhe-2, Intel Xeon Phi ; #2: Titan, Nvidia K20)
- In some cases (GPU) **rewrite completely code** in technology specific language
- GPGPUs are particularly tailored to specific problems: **very high performances** can be reached in specific domain completely rewriting code in specific language
- Intel Xeon Phi advantage: **no-need to rewrite code**, optimizations done for MIC architecture are valid for host CPU

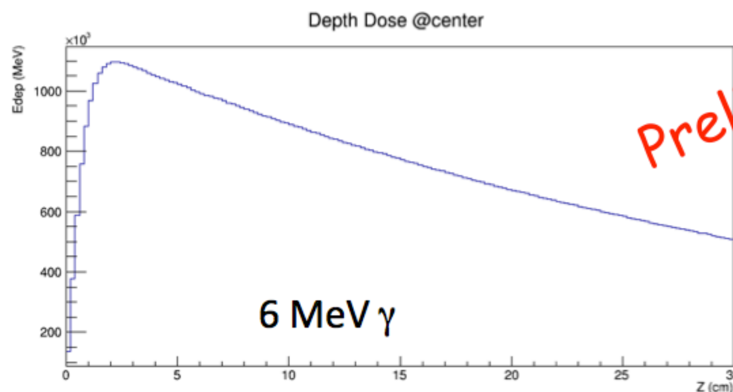
GPU

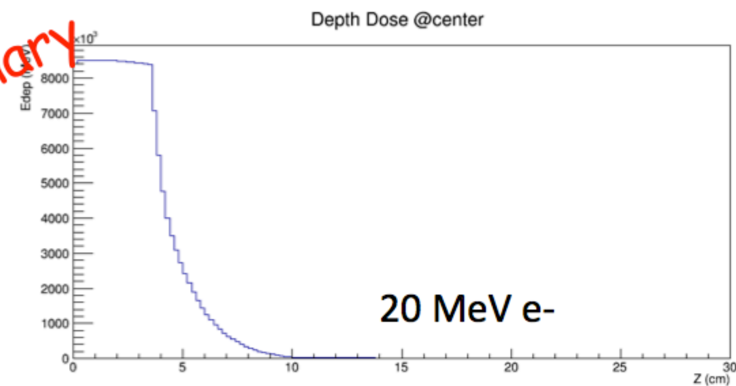
SLAC

- SLAC/Stanford-ICME/KEK/NVIDIA project
- Full EM physics <100 MeV electrons/gammas
  - Only one kind of material (water) with varying density (medical DICOM)
  - No geometrical navigation (only voxelization)
- Benchamark on TESLA K20: O(100) faster than CPU G4 job (full normal navigation, full physics)
- Very promising for domain-specific applications





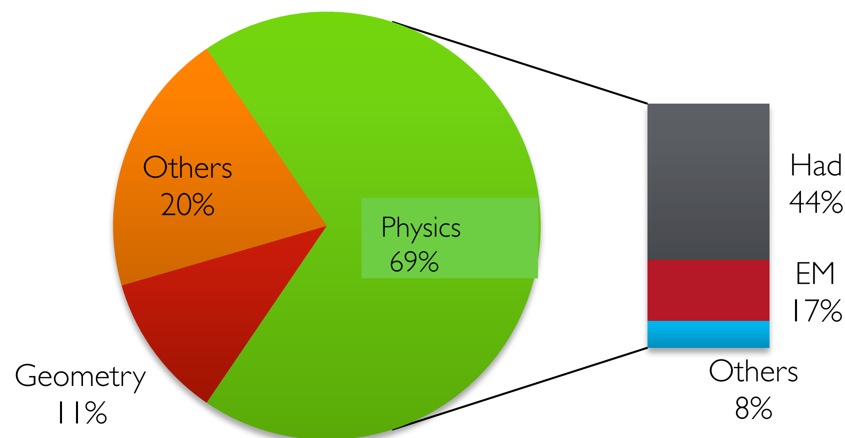




Preliminary

Beyond event-level parallelism

- To fully use new architecture potentials we need to **investigate further level of parallelism**
- SIMD (a.k.a. vectorization): very challenging (very limited success for HEP sw). Two options available:
    1. Rewrite code with intrinsics or low-level constructors (bad idea: not portable, very complex for large sw as G4)
    2. Use compilers **auto-vectorization** features together with high-level construct
- 2011 results compiler auto-vectorization for G4 (out of the box): ~10k candidate loops, only about 5% actually auto-vectorized. Good point: compilers improve constantly
- More study needed for more-than-event-level parallelism



Possible strategy:
Map candidate loops with most time
Consuming routines, iteratively try out auto-vectorization

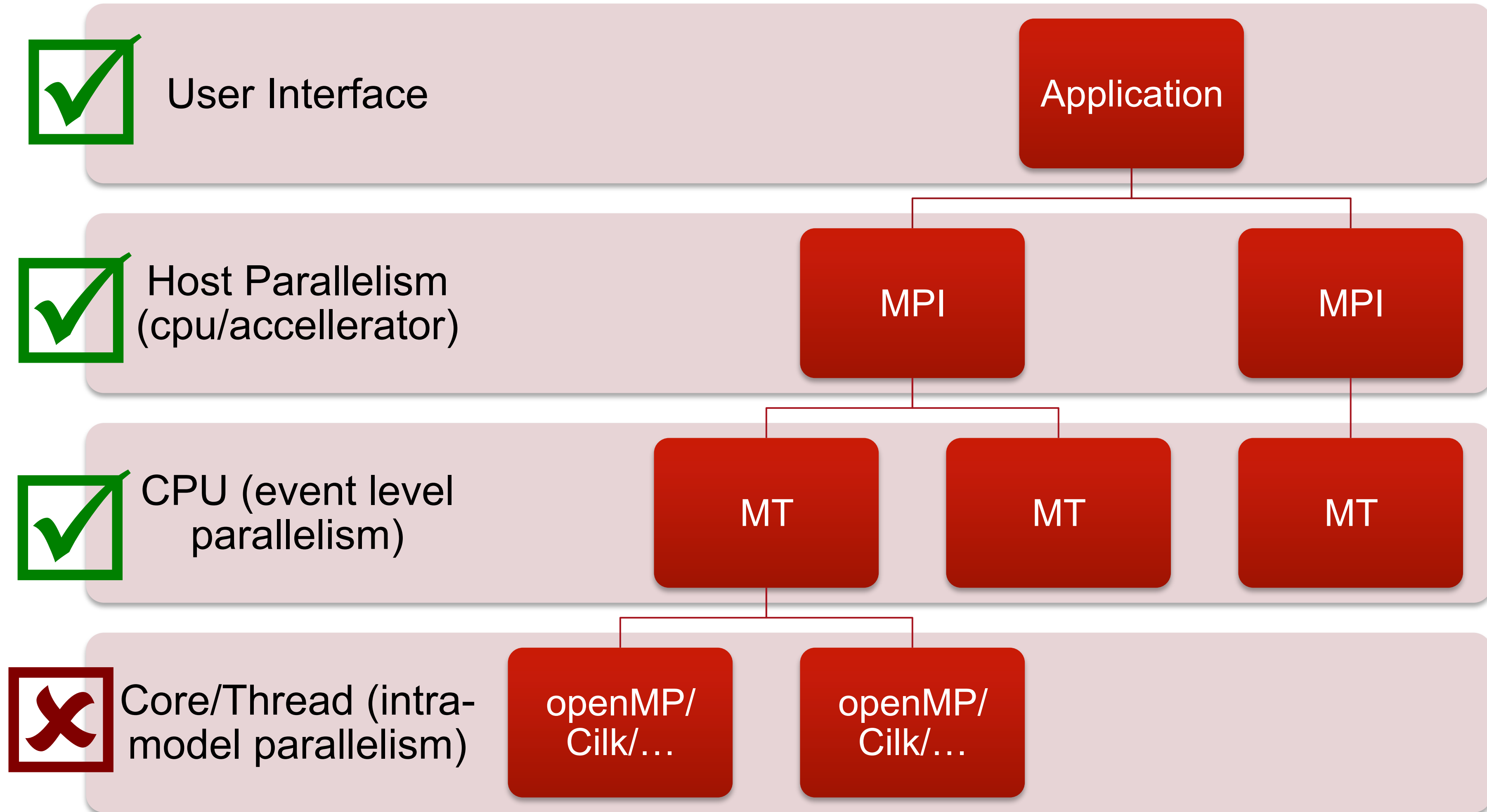
My personal view

SLAC

- **Use of high-level parallelization** constructs to put parallelization in modules/algorithms for example:
  1. **openMP** : de-facto standard, support for Intel accelerators
  2. **Intel CilkPlus** : very simple usage (function vectorization via #pragmas), GCC support only in branch (integration with TBB)
  3. **openACC** : relatively new open standard aimed at developing directives for accelerators (technology independent)
  4. Your favorite goes here …
- Need to experiment with all technologies and understand benefits / challenges
- Possible path: identify one or two of the most time consuming elements in simulations (e.g. cross-section calculations , Bertini intra-nuclear cascade) and try to apply inter-algorithm parallelism

A roadmap: my view

SLAC



Conclusions

SLAC

- Geant4 Version 10.0 well on track for end of year release
- Major developments for event-level parallelism
  - **Very promising results obtained on both "traditional" CPUs and MIC architectures**
  - Expect further improvements
- Possible to integrate G4MT with additional high-level parallelization frameworks (TBB, MPI)
- **Scalability demonstrated up to O(100) threads**
  - Ready for future challenges of current and next generation large simulation campaigns (i.e. LHC –scale)
  - New possibilities for "smaller" simulation needs (efficient use of many core machines, accelerators on desktops)
- **Multi-threading and thread safety is the first indispensable step towards further review of Geant4 code**

Acknowledgments

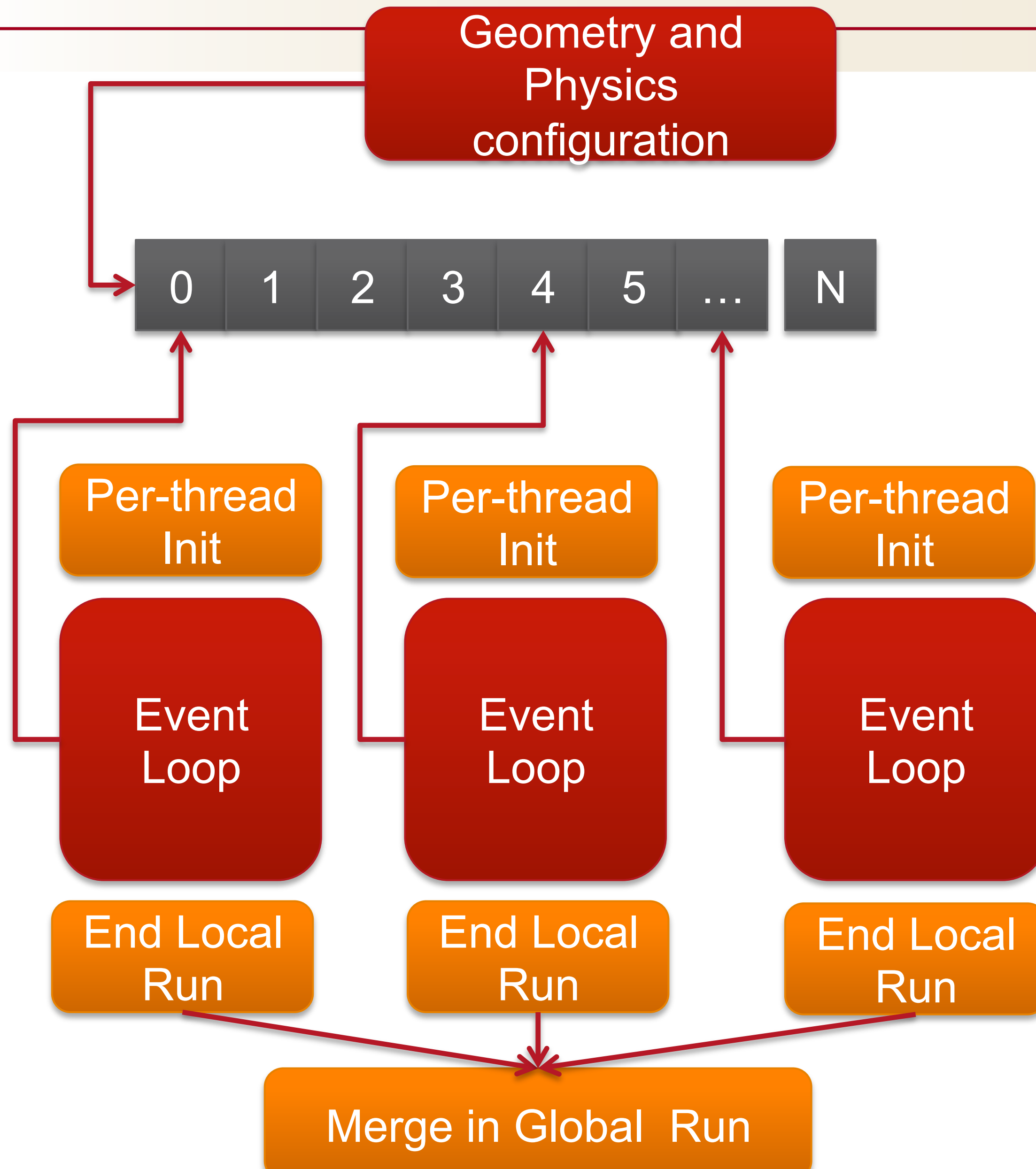
SLAC

I would like to thank few people that helped to produce the material here presented and gave important contributions during discussions:

- K. Murakami (KEK), N. Henderson (SU), A. Vladimirov (SU), G. Cooperman (NortheasternU), X. Dong (NortheasternU), G. Cosmo (CERN), P. Elmer (PrincetonU)

Backup
U.S. DEPARTMENT OF
ENERGY
Office of Science
SLAC
NATIONAL
ACCELERATOR
LABORATORY

Random Seeds and Run



- To gurantee reproducibility each thread has its own RNG
- Master thread pre-generates per-event seed
- Each event is re-seeded
- Further refinement on RNG to be studied (pRNG)
- New: threads compete for (group of) next events to process