Creating and Improving Multi-Threaded Geant4

Xin Dong, Gene Cooperman (Northeastern Univ.)
John Apostolakis, Andrzej Nowak, Sverre Jarp (CERN)
Makoto Asai, Daniel Brandt (SLAC)

Introduction

The Geant4 toolkit is used in large scale detector simulation in many High Energy Physics experiments, Space Engineering and Medical Physics applications. To adapt Geant4 to the current era of multi-core computing, we created a prototype multithreaded version of Geant4 (Geant4MT) which employs event-level parallelism. We describe the source code transformation techniques by which Geant4 MT is created. This method is currently also used to port new releases of sequential Geant4 to Geant4MT. After this, tools are used to check that the changes are correct and complete, so that the results of an event are bit-compatible between sequential and MT versions. We present an overview of the design of Geant4MT, and the speedup results achieved. A CPU time overhead was found in Geant4MT with one-worker. We identify key causes of this overhead, and describe how to reduce them significantly.

Aim & requirements

Geant4MT aims to reduce the memory footprint by sharing the largest data structures in Geant4. This will allow the use of tens of threads, while using one a few times the memory of a sequential job.

Key requirements for Geant4MT include:
- bit-level compatibility of results with the sequential version - given the same starting state of a pseudo Random Number Generator (pRNG) for each event;
- simple porting of applications;
- efficient use of multi-core and many-core hardware though good scaling of performance.

1. Separate All Data

To create G4MT source from G4 the first source transformation is to:
1) Make all objects thread local by "**Transformation for Thread Safety**" - threads share only read-only objects.
2) Each thread has its own instance of a pseudo random number generator (pRNG).

This is a 'baseline' for G4MT, and must give exactly the same results as a set of runs of the sequential program with the same seeds for the pRNG.
Each thread has a separate instance of the geometry and all processes at this point.
We now change this, to share the largest data structures.

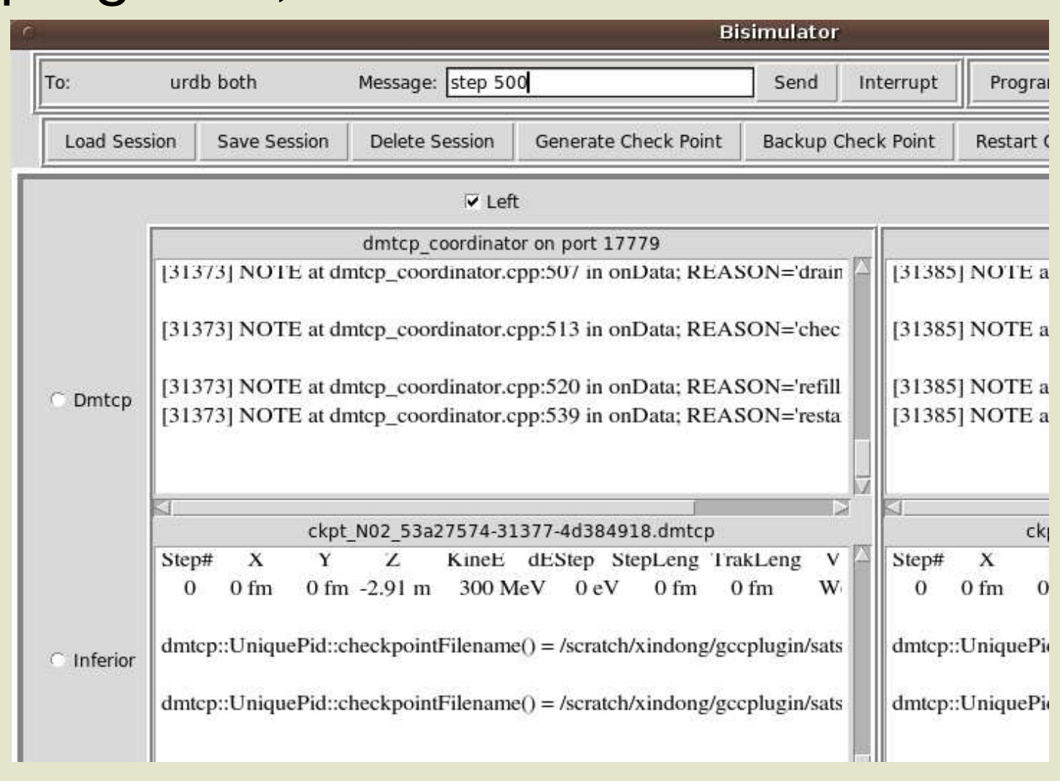
2. Share Large Data

1) Identify data structures which are read-only after initialization. Use **Transformation for Memory Footprint Reduction (TMR)** to share between threads the
   - Geometry
   - Physics tables (σ, dE/dx, ρ)
2) Split classes with read-only and writable parts. A *Particle Type* has:
   - *Invariant properties:* q_0, m_0
   - N_{thread} *Process Managers* (one per thread, each with a full set of processes for the particle.)
3) Revised Physics Table classes to separate a Read-Write part which "cache" the last value.

Create extra code to initialize all objects in a worker thread from corresponding objects in the master

3. Check one worker

We compare a Geant4MT program, running with 1 worker thread and its original sequential Geant4. We use the temporal debugger URDB [3] to record each program's full state at regular intervals, in bisimulation model.
When a difference is observed between the versions, we use the tool to **roll back** execution of both programs, until we find its cause.



```
dmtcp_coordinator on port 17779
[31373] NOTE at dmtcp_coordinator.cpp:507 in onData; REASON='drain
[31373] NOTE at dmtcp_coordinator.cpp:513 in onData; REASON='chec
[31373] NOTE at dmtcp_coordinator.cpp:520 in onData; REASON='refill
[31373] NOTE at dmtcp_coordinator.cpp:539 in onData; REASON='resta

ckpt_N02_53a27574-31377-4d384918.dmtcp
Step#    X     Y      Z      KineE    dEStep  StepLeng  TrakLeng  V
  0    0 fm  0 fm  -2.91 m  300 MeV   0 eV     0 fm      0 fm    W

dmtcp::UniquePid::checkpointFilename() = /scratch/xindong/gccplugin/sats
dmtcp::UniquePid::checkpointFilename() = /scratch/xindong/gccplugin/sats
```

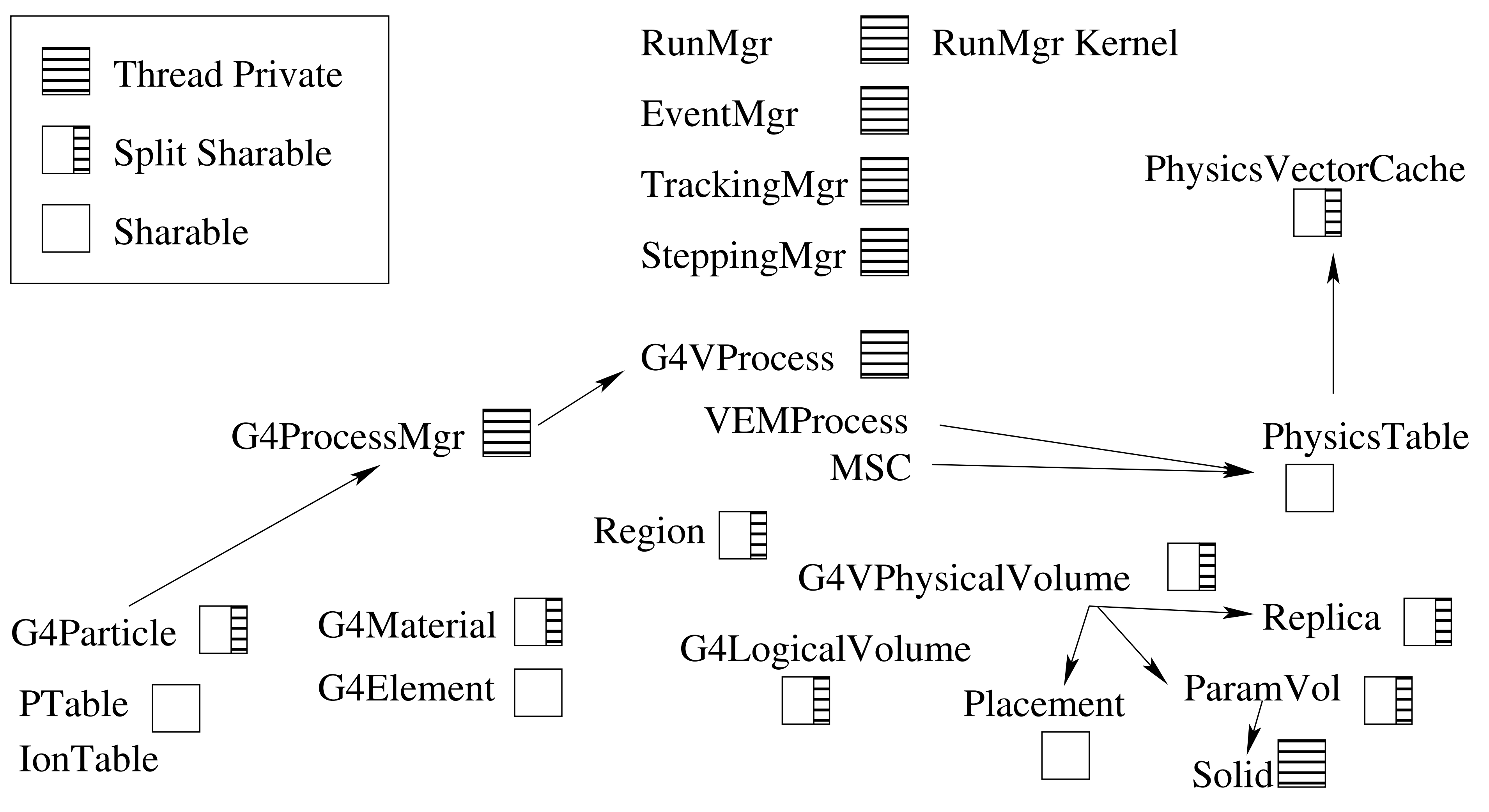
```
[31385] NOTE a
[31385] NOTE a
[31385] NOTE a
[31385] NOTE a
ck
Step#    X
  0    0 fm   0
dmtcp::UniquePi
dmtcp::UniquePi
```

Figure 1: Design of Geant4 MT: private, split and shared classes.

Classes in which large amount of data is stored are shared ("**Sharable**"): e.g. Physics-Table.

To enable this some classes have one part shared, and another private ("**Split Shareable**"): e.g. Particle-Definition (G4Particle) which shares properties and has a separate Process-Manager per thread.

The remaining classes are replicated for each thread ("**Thread Private**"), including all physics processes and all classes dealing with tracking, event and run management.



Consequences of Geant4MT's design

Classes which are split ("**Split Shareable**") must be re-engineered to separate all thread-private parts into one (or more) separate objects. Each thread is given an instance of each new subobject; all code references to these data members must be changed in the class implementation.

When a worker thread is created, Geant4MT must create a copy of all objects necessary for a separate simulation to start:
- new instances of all **Thread Private** objects, including the managers for run, event, and tracking;
- separate instances of the non-shared parts of "Split" objects.

Figure 2: **CPU time for Geant4 MT running on a 40-core system** with Westmere EX CPUs.

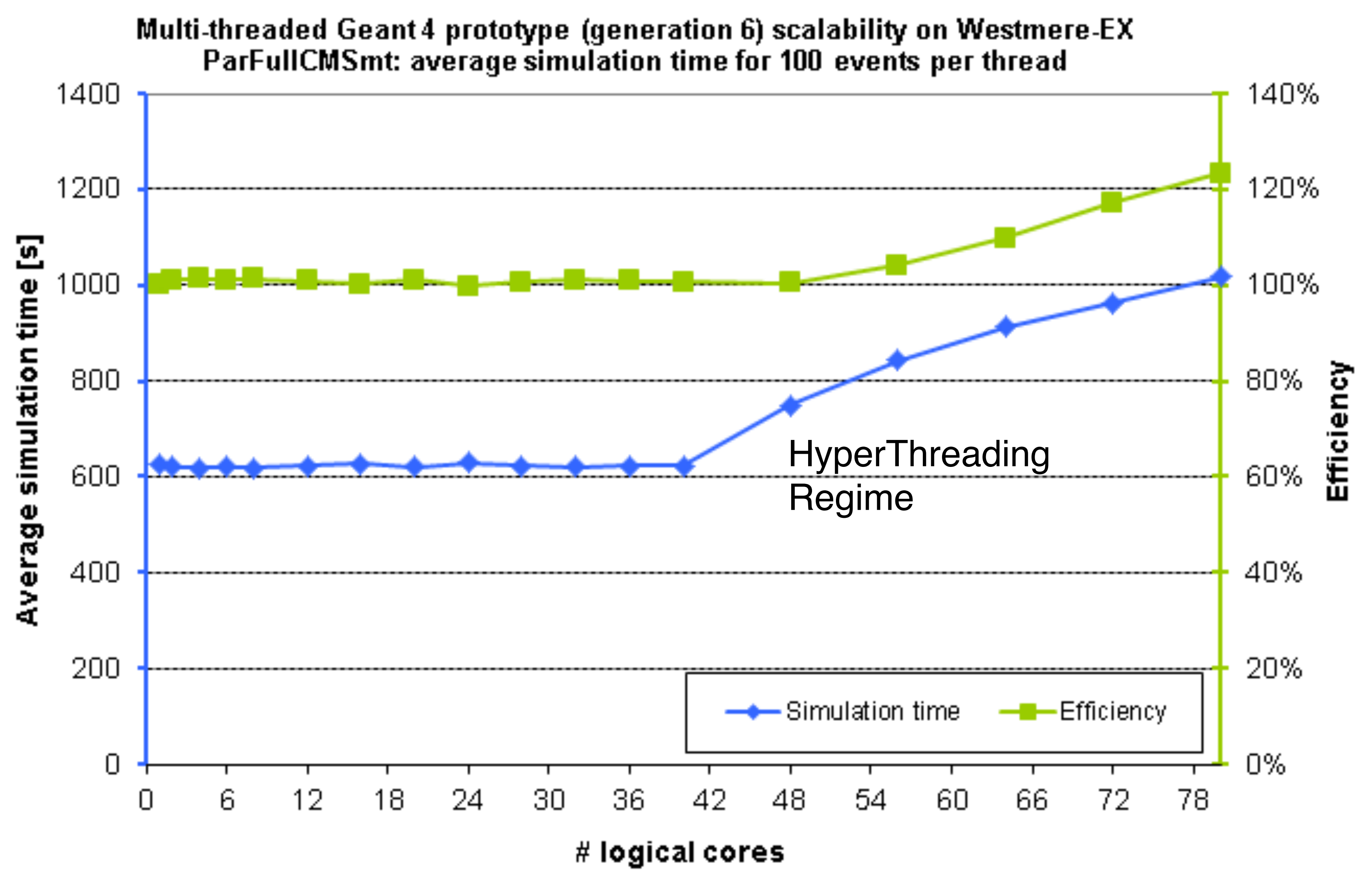
Details :
Geant4 MT based on Geant4 9.4.patch 01 (pre-release version.)

gcc 4.3.4, 64-bit, 128 GB memory

Ran 100 events per thread. Each event consists of one 300 GeV pion.

The setup is a model of the full CMS geometry - which was obtained in 2008 (and is imported from a GDML file.)

Efficiency = Time (N workers) *



6. Scaling

Comparing against the time for the 1 worker + 1 master, we observe good scaling on systems with up to 40 cores.

No sequential bottleneck is seen for 40 or 80 workers.

Hyper-Threading adds 20% to the throughput.

4. Check Data Races

In case results with several threads are different from one thread results, we must identify which shared data is changed during the simulation.

To do this we extended the selective memory protection technique from TMR to monitor the runtime correctness - with a new tool, the **Data Race Coordinator (DRC).**

This can be used in two modes:
- check mode, which flags a problem and exits;
- correction mode, which creates separate memory in case of unexpected sharing.

The run time overhead is small - proportional to the number of errors detected.

5. Fix Performance

Sharing objects initially increased CPU time. The causes was large number of mutex calls due to:
1) Allocation of per-thread local objects by common allocator; scaling restored by introducing *per-thread allocator*.
2) Updating of shared variables - for printing - was suppressed.

After these issues were addressed, good performance is obtained, as seen in the figure above.

Note: significant effort was required to identify the causes in each case, due to the primitive nature of the of the tools that were available.

7.Thread Overhead

Comparing time per event on sequential version with 1-worker version, identified an overhead of 30% [4].

Two sources have been identified:
1) calls to identify the thread, required in order to resolve the address of thread-local data;
2) extra indirection added to split objects.

By changing the model for Thread Local Storage [5] from "init" to "gnu2" [6] the overhead was reduced to 18%.

8. Porting Applications

To port a stand-alone application to G4MT a user must change its main program to use G4MTRunManager, and to review all User Actions classes: Stepping, Event and Run Actions. (Hit classes are unchanged, as hits are collected per event.) We estimate that this will take 0.5 to 2 days, including first tests.

9. Porting Geant4MT

Today we port Geant4MT to a new G4 version redoing each stop of this process: applying TTS and TMR, porting the "extra" G4MT code and checking with URDB and DRC. The effort required is of the order of one week. We plan to incorporate key changes into Geant4 in 2012-13.

Resources and further reading

1. G. Cooperman et. al, Proc of CHEP 2000, arXiv:hep-ph/0001144.
2. X. Dong et al. *"Multithreaded Geant4: Semi-Automatic Transformation into Scalable Thread-Parallel Software"*. In: Proc. of Euro-Par'2010, 2010.
3. A. M. Visan et. al., *in* Software Engineering (2009) and **arXiv:0910.5046v1**.
4. P. Canal (FNAL), private communication.
5. U. Drepper "*Elf Handling for Thread-Local Storage*" http://www.akkadia.org/drepper/tls.pdf
6. A. Oliva and G. Araujo, *"Speeding Up Thread-Local Storage Access in Dynamic Libraries"*, in GCC Developers' Summit 2006, 2006, pp. 159-178.