Scalability tests of a multi-threaded Geant4 prototype

Andrzej Nowak
June 16th 2009
A small part of a story of turning a sequential program into a parallel application

All results presented are preliminary, this is a work in progress.
Geant4

- Prominent software framework (toolkit) used for simulating the passage of particles through matter
- [http://cern.ch/geant4](http://cern.ch/geant4)
- LHC Users:
  - ATLAS
  - CMSSW
  - ALICE
  - Gauss (LHCb)
- Other users:
  - BaBar
  - Fermilab
  - ESA
  - Others
Rationale: Multi-core “crisis”
Rationale: “many-core mega-crisis?”

> We’ve been talking about multi-core for a long time
  - It’s here
  - We’ve done little to use it
  - Is it already too late?

> The many-core crisis is looming
  - 6-core parts from AMD and Intel are a reality today
  - 24-core systems are available in your local “computer shop”
  - Larrabee is coming – 4-way SMT, many cores: reasonable to expect >20
  - Nehalem-EX (“Beckton”) is around the corner – 64 threads in a box by the end of this year

> Will we still need 2GB per process at CERN?
Rationale

> Geant4 + “core crisis” = multi-threaded Geant4 prototype

> Xin Dong and Gene Cooperman from NEU (Northeastern University) are working on a multi-threaded prototype of Geant4 since 2007

> Working prototype of CMS-SW delivered in early 2009
  - Based on FullCMS
  - Full correctness maintained
  - Well planned approach to parallelizing an existing, sophisticated application
  - Excellent initial results

> Work continues with the involvement of the Geant4 team and CERN openlab
Problem decomposition and approach

- Event level parallelism (implemented using the TOP-C library)
- Code needed to be thread-safe and reentrant
- Semi-automatic way devised to parse existing code and “upgrade” it to a multi-threaded version
- Some manual changes needed as well
- Ongoing work to automate the whole process
Multi-threaded Geant4

> Significant amount of data shared read-only and only 1 critical data structure is shared with explicit locking – the ion table

> Huge reduction in terms of memory consumption: \(~25\text{MB of memory per thread}\)
  - A 64 core machine could be fully filled and have only 2GB of memory!

> Several distinct phases:
  - Serial initialization
  - Parallel initialization
  - Parallel runtime (simulation)
  - Parallel termination
Scalability tests at openlab (Q2 2009)

> Harpertown systems – 2x4 cores (8 total)

> Dunnington systems – 4x6 cores (24 total)
Step 1 – “Stopwatch runs”

MTG4 - Harpertown scaling

<table>
<thead>
<tr>
<th>Hardware threads</th>
<th>Execution time (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10000</td>
</tr>
<tr>
<td>2</td>
<td>5000</td>
</tr>
<tr>
<td>4</td>
<td>2500</td>
</tr>
<tr>
<td>8</td>
<td>1250</td>
</tr>
<tr>
<td>10</td>
<td>1000</td>
</tr>
</tbody>
</table>

Harpertown E5450, SLC5, 8 cores, 8 threads, gcc 4.1.2, 64-bit, 16GB memory (olbl0132)

Perfect scaling
Step 1 – “Stopwatch runs”

MTG4 - Dunnington scaling

- **Dunnington X7460**, SLC5, 24 cores, 24 threads, gcc 4.1.2, 64-bit, 48 GB memory (plitehp24)
- Perfect scaling
Step 1 – Initial conclusions

> **Resource contention**
  - High system time (is ~5%)
  - CPU usage could be better (is ~92%)
  - Those problems are gone if we start 3x8 processes

> **Expected better scaling past 8 cores**
  - Going from 8 to 24 gives ~25% instead of ~200%

> **Event processing time not the issue? What is the impact of the different phases?**

> **Good example of the “multi-core vs. many-core” issue**
Step 2 – Focus on the simulation part

CPU graph
Step 2 – Focus on the simulation part

Memory usage graph

Serial initialization ~ 230 seconds

Parallel worker initialization ~ 120 seconds

Parallel computation ~ 460 seconds
Step 2 – Focus on the simulation part

**Speedup**

MTG4 - Dunnington scaling (500 evts, pi-, 300GeV)

Execution time (seconds)

Incremental speedup (to previous)

Hardware threads

- Time [s] 500 evts, pi-, 300GeV
- Time [s] 500 evts, pi-, 300GeV, PERFECT
- Reported simulation time
- Execution time speedup (#cores, value)
Step 2 – Focus on the simulation part (red line should be flat)

MTG4 - Dunnington scaling (500 evts per thread, pi-, 300GeV)

![Graph showing MTG4 - Dunnington scaling (500 evts per thread, pi-, 300GeV)]
Step 2 – Conclusions

> The initialization and termination phases are not an issue

> Adding resources past 8 threads yields little improvements in the simulation part (up to 24 threads)

> Running 3 processes x 8 threads gives expected results
  - Nearly 300% throughput increase compared to 1p x 8t
  - When 3p x 8t are running, each of the processes is 1-2% slower than when running alone (i.e. 1p x 8t)

> There is a software scaling problem
Step 3 – OS level analysis

> Perfmon2, strace and code instrumentation used

> Perfmon 2 monitoring
  ▪ Looking for cache effects, false sharing, congestion points

> System call histogram generated with strace – high system time means kernel activity

> Code instrumented to verify locking frequency, time and side effects
Step 3 - Strace results – syscall profiles (all inclusive, 500 pi- 300GeV)

> System time spent doing the futex call:

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>4</th>
<th>8</th>
<th>12</th>
<th>16</th>
<th>20</th>
<th>24</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>System time [s]</strong></td>
<td>0.04</td>
<td>0.5</td>
<td>6.95</td>
<td>68.18</td>
<td>219.47</td>
<td>411.94</td>
<td>767.09</td>
</tr>
<tr>
<td><strong># calls</strong></td>
<td>5264</td>
<td>23791</td>
<td>8’517’227</td>
<td>11’116’321</td>
<td>21’448’012</td>
<td>29’540’920</td>
<td>26’885’198</td>
</tr>
<tr>
<td><strong>μs per call</strong></td>
<td>8</td>
<td>22</td>
<td>1</td>
<td>6</td>
<td>10</td>
<td>14</td>
<td>29</td>
</tr>
</tbody>
</table>

- System time: 1900x increase (1 -> 24)
- Call frequency: 5000x increase (1 -> 24)

> The read call (1 thread -> 24 threads):
  - The amount of read calls grows as expected (5x)
  - The system time spent in read calls grows rapidly (58x) also due to the growth of the length of the servicing period per call (13x)

> mremap usage/service time grows, but insignificant
Step 3 – Conclusions

> Perfmon counts and profiles look “normal”

> Locking frequency not a prime suspect at this point

> Unlikely causes:

  - Time spent in explicit locks
    - Only 1us spent in a critical region on average
    - Translates to ~1% of the time spent in critical regions
  - Cache effects and false sharing
    - Roughly 1% cache misses, virtually no false sharing effects
  - Linux scheduling
    - 3 processes x 8 threads works fine
  - I/O
Step 3 – Mysteries

> First symptoms appear already when moving from 4 to 8, but the system is able to handle it.

> Why is there a futex explosion when moving from 4 to 8 and from 8 onwards?

> Why is there a disproportional system time increase when increasing the number of threads?

> Why are there 2 million SIGSEGV handler reassignments? Why does the handling time increase with the number of threads?
Step 4 – OS and code level analysis, round 2

> System call analysis - high system time means kernel activity
  ▪ Strace traces + home made tools

> Code analysis
  ▪ Intel Thread Checker
  ▪ Intel Thread Profiler
  ▪ ltrace

> IP tracing with strace was a disappointment

> Intel tools initially wouldn’t work with our application – bugs filed, activity put on hold

> ltrace – too slow to get meaningful output
Locking and system call statistics

> Home made tools used to analyze the traces (no solution ready)

> Per-thread system call statistics
  - Number of calls
  - Max / min / avg time
  - Deviation
  - Errors
  - Total time spent in calls

> I/O breakdown
  - file ops

> Futex histogram
  - count / time spent

> More items planned
Step 4 – Conclusions

> Locking is definitely a problem

> Lock decomposition needed to distinguish different locks – upgrades for the home made tools needed
  - Network I/O breakdown
  - Detailed futex statistics (total time spent, taking concurrency into account, futex breakdown and decomposition)
Step 5 – Low level analysis

> Kernel-level analysis (SystemTap, Utrace): inconclusive, ongoing

> Thread Checker is in conflict with the internal structure of Geant4 – won’t work unless G4 is recompiled with certain options
  ▪ Put on hold

> Thread Profiler
  ▪ Experimental version from Intel works
  ▪ 1 hour just to open the trace file on a modern machine
  ▪ Analysis limited to 100’000 events (average files we generate have millions), which is about 10 seconds of runtime
  ▪ Issues with symbols
Step 5 – Thread Profiler overview

> ~10 seconds of execution analyzed at a time

> Yellow is bad. (synchronization objects)
Step 5 – Interesting side effects in TP

> Work imbalance
Concurrence graph for the 10s fragment

> Green (efficient work) portion is barely 20%
Loop fragment - Thread utilization

> Computing resources heavily underutilized, some threads appear to be starved, others appear to be dominating
Loop fragment zoom

> Dark green = good work

> Light green = no work, waiting, idle
Concurrency graph - loop fragment zoom

- Concurrency level in the middle of the event loop is low, hovering around 12-16.
- Expected level (“perfect”) is 24
> It’s possible to determine the exact locations of problematic mutexes

> Even lower levels accessible, not shown
Current plans

- Scalability improvements (locking system upgrade)

- Updating the multi-threaded Geant4 prototype to work with the latest version of Geant4

- Further scalability investigations
  - New versions of code
  - Lock decomposition
  - Continued activities with SystemTap and utrace
  - Thread Checker?
Summary – Conclusions

> Drilling down from a very high level to a low level for the first time takes effort and time

> Good to have a process for such activities

> Commercial tools can help a lot

> GetIon is the main culprit?