Memory Improvements to Bertini Cascade

Michael H. Kelsey
SLAC GEANT4 Group

GEANT4 Computing Performance Task
27 May 2010
Bertini cascade shown to be memory and CPU intensive (about 25% of CMS simulation)

Several areas of concern, which I've been addressing

• Hardwired masses, particle codes, kinematics
• Internal bookkeeping of cascade secondaries
• Duplication (copy-and-paste) of many code fragments
  * Systematic passing of vectors by value, not reference

Last item is the main source of memory churn and CPU waste
Secondaries generated during cascade process stored in G4CollisionOutput

Two std::vector’s, for hadrons and for nuclear fragment(s)
Passed and returned by value, copied to other vectors, and so on
In many places, temporary lists are created and copied

All of this leads to substantial creation, deletion, and copying of objects (memory churn), with consequent CPU usage and page swapping
In private tag **hadr-casc-V09-03-23**

- All function arguments are now passed by reference, either \texttt{const} (the majority of cases) or non-const (where the argument is a return buffer)
- All functions where \texttt{std::vector} was being returned by value have been recoded to either return a \texttt{const} reference to a data member, or to return \texttt{void}, and accept a non-const input \texttt{std::vector} for filling
- All within-class functions which were passing vectors between them have been recoded to use class data-member buffers

These changes are on top of other development work in the package, and also depend on **particles-V09-03-01**
Performance Improvements

Used **IgProf** with “little” validation job, ppi4GeV
⇒ 1,000 events of 4 GeV $p$ on $^{208}$Pb

g**eant4-09-03-ref-03 Release**

12.3 wall-clock seconds at 69.7% efficiency
8.250±0.333 CPU seconds, and 6,291 page faults
“Memory churn” (total allocations) 220 MB in 3.2 Mcalls

**hadr-casc-V09-03-23 and particles-V09-03-01**

7.52 wall-clock seconds at 96.4% efficiency
7.053±0.205 CPU seconds, and **3 page faults**
“Memory churn” (total allocations) 93 MB in 548 kcalls

**g**eant4-09-03-ref-04 **Release**

Includes **hadr-casc-V09-03-23, particles-V09-03-01**
12.3 wall-clock seconds at 69.7% efficiency
8.250±0.333 CPU seconds, and 6,291 page faults
“Memory churn” (total allocations) 136 MB in 647 kcalls
4.9.3 Performance Bottlenecks

geant4-09-03-ref-03 with debugging info

RHEL5 32-bit Athlon, running ppi4GeV Pb 1000

<table>
<thead>
<tr>
<th></th>
<th>Memory</th>
<th>Calls</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>100.00</td>
<td>220,714,885</td>
<td>3,178,293</td>
<td>Cumulative</td>
</tr>
<tr>
<td>40.62</td>
<td>89,661,000</td>
<td>582,350</td>
<td>G4InuclElementaryParticle</td>
</tr>
<tr>
<td>23.30</td>
<td>51,429,080</td>
<td>1,666,392</td>
<td>double (vector)</td>
</tr>
<tr>
<td>11.46</td>
<td>25,293,424</td>
<td>192,515</td>
<td>G4CascadParticle</td>
</tr>
<tr>
<td>3.74</td>
<td>8,249,344</td>
<td>1,007</td>
<td>iostream</td>
</tr>
</tbody>
</table>

Results courtesy of IgProf, via Peter Elmer/CMS
After -23 and particles Updates

geant4-09-03-ref-03 with debugging info
cvs update -r particles-V09-03-01 source/particles
cvs update -r hadr-casc-V09-03-23 source/processes/hadronic/models/cascade

RHEL5 32-bit Athlon, running ppi4GeV Pb 1000

<table>
<thead>
<tr>
<th>%</th>
<th>Memory</th>
<th>Calls</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>100.00</td>
<td>92,727,821</td>
<td>547,728</td>
<td>Cumulative</td>
</tr>
<tr>
<td>64.55</td>
<td>59,859,520</td>
<td>114,769</td>
<td>G4InuclElementaryParticle</td>
</tr>
<tr>
<td>8.90</td>
<td>8,249,344</td>
<td>1,007</td>
<td>iostream</td>
</tr>
<tr>
<td>5.87</td>
<td>5,439,588</td>
<td>9,860</td>
<td>G4CascadParticle</td>
</tr>
<tr>
<td>4.13</td>
<td>3,832,400</td>
<td>5,277</td>
<td>G4InuclNuclei</td>
</tr>
<tr>
<td>4.02</td>
<td>3,729,408</td>
<td>151,282</td>
<td>double (vector)</td>
</tr>
</tbody>
</table>

Results courtesy of IgProf, via Peter Elmer/CMS
Bertini Updates in Production

geant4-09-03-ref-04 with debugging info
⇒ Many other G4 changes included as well

RHEL5 32-bit Athlon, running ppi4GeV Pb 1000

CPU time: 7.120u 0.280s 0:09.92 74.5% 0+0k 0+22io 2011pf+0w

<table>
<thead>
<tr>
<th>%</th>
<th>Memory</th>
<th>Calls</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>100.00</td>
<td>136,044,893</td>
<td>647,632</td>
<td>Cumulative</td>
</tr>
<tr>
<td>74.40</td>
<td>101,223,216</td>
<td>212,673</td>
<td>G4InuclElementaryParticle</td>
</tr>
<tr>
<td>6.06</td>
<td>8,249,344</td>
<td>1,007</td>
<td>iostream</td>
</tr>
<tr>
<td>4.15</td>
<td>5,640,300</td>
<td>7,277</td>
<td>G4InuclNuclei</td>
</tr>
<tr>
<td>4.08</td>
<td>5,550,600</td>
<td>9,860</td>
<td>G4CascadParticle</td>
</tr>
<tr>
<td>2.74</td>
<td>3,729,408</td>
<td>151,282</td>
<td>double (vector)</td>
</tr>
</tbody>
</table>

Results courtesy of IgProf, via Peter Elmer/CMS
Pete Elmer built and ran CMS simulation using ref-04

Relative to G4.9.3.p01 I see 1.9-4.5% improvements in CPU, depending on event types (minimum bias vs. top)

The Bertini code seems much better relative to memory allocations! There are some minor things left, but the largest ones seem to be gone...

Many thanks to Pete for running these tests and confirming the improvements
CMS utility, courtesy Pete Elmer: very nice calling and memory performance analysis tool, IgProf

- Combines features of gprof and valgrind into a common interface
- gprof report format for easy analysis
- No special compilation (uses shared-library substitution)
- Reports calls, memory churn (allocations), and memory leaks
- Text reports or navigable SQL-based Web files

Available “soon” through SourceForge
Side Note: Interpolation Performance

Bertini code uses interpolation in many places, for cross-sections, angular distributions, multiplicities and final states.

“Should” use subclass of G4PhysicsVector for this purpose.

Class looked pretty heavy, so I wrote simple linear interpolator based on C arrays, templated on array dimension for type-safety.

G4CascadeInterpolator<N> caches last interpolated $x$ value and bin indexing, to speed up multiple operations on same particle.
Side Note: Interpolation Performance

Simple comparison using clock() (#include <time.h>)

- G4CascadeInterpolator vs. G4PhysicsOrderedFreeVector with identical $x$ bin arrays, three different $y$ (data) arrays
- Millions of iterations, with successively different $x$ values
- One pass interpolates just one data array each time, another pass interpolates all three data arrays

<table>
<thead>
<tr>
<th></th>
<th>Single</th>
<th>Triple</th>
</tr>
</thead>
<tbody>
<tr>
<td>G4CascadeInterpolator</td>
<td>1.53971</td>
<td>2.63096</td>
</tr>
<tr>
<td>G4PhysicsOrderedFreeVector</td>
<td>5.31744</td>
<td>15.0670</td>
</tr>
<tr>
<td>Penalty factor</td>
<td>3.45</td>
<td>5.73</td>
</tr>
</tbody>
</table>
With **IgProf**, evaluating hot-spots for both memory usage and excessive calling “loops” is simple.

Run test jobs through **IgProf** frequently (every tag?), compare with last baseline.

Integrate performance improvements with bug fixes and other code changes.

Encourage availability and use of **IgProf** elsewhere in GEANT4.