CODE OPTIMIZATION
OF THE BERTINI INTRA-NUCLEAR- CASCADE MODULE
IN THE GEANT4 FRAMEWORK

Michael Hannus[1], Mats Aspnäs[1], Aatos Heikkinen[2], and Jan Westerholm[1]

[1)] Åbo Akademi University,

Joukahainengatan 3-5, 20540 Åbo, Finland

www.it.abo.fi

[2)] Helsinki Institute of Physics,

P.O.Box 64, 00014 University of Helsinki, Finland

www.hip.fi

Optimization of Bertini INC module in the Geant4 framework

Background and goals

The High Performance Computing group at Åbo Akademi
specialize in the software for large scale computing (time and memory)
using modern and new techniques to improve program throughput.

In order to facilitate large scale simulations with
the Geant4 Bertini intranuclear cascade (INC) code,
we investigated methods of speeding up the INC simulation
using code optimization.

Optimization of Bertini INC module in the Geant4 framework

- Observations were made with the help of a code-profiler and a debugger:
  - The profiler was used to select target functions that would benefit the most of optimization
  - The debugger was used to analyze the call patterns for these functions
- Profilings were performed on a simulation where an Yttrium target was bombarded with a neutron having a momentum of 500 MeV. The collision was repeated 10000 times
- The machine used was a 2.8GHz Pentium4 with 1GB of memory

Optimization of Bertini INC module in the Geant4 framework

•A major bottleneck was found to be the use of the dynamically allocated std::vector class:

- The table shows the relative time spent in different portions of the program
- The momentum and position variables of the particle and the nuclei classes were the std::vector variables that were accessed the most
- These variables were easily replaced by a static array-like datastructure, as they always contain the same number of elements

Name	Clockti %
G4InuclElementaryParticle::G4InuclElementaryParticle	9.98%
allocate	8.58%
bindingEnergyKummel	4.90%
generateInteractionPartners	4.08%
~vector	3.35%
operator=	2.62%
~G4InuclElementaryParticle	2.32%
collide	2.32%
toTheCenterOfMass	2.20%
generateNucleon	2.04%
vector	1.94%
particleSCMmomentumFor2to2	1.92%
generateParticleFate	1.87%
_ZNSt12_Vector_baseIdSaIdEE11_M_allocateEj:...	1.66%
G4InuclElementaryParticle	1.62%
collide	1.48%
__copy_aux<const double*, double*>	1.43%
inuclRndm	1.38%
generateSCMfinalState	1.27%
collide	1.24%
__uninitialized_fill_n_a<double*, unsigned int, dou...	1.22%
_ZNKSt6vectorIdSaIdEE4sizeEv:_ZNKSt6vectorId...	1.19%
backToTheLab	1.19%
_ZN9__gnu_cxx11__pool_base18_M_check_thre...	1.17%
_M_insert_aux	1.15%
randomCOS_SIN	1.12%
rotate	1.12%
_Vector_base::_Vector_base	1.05%

Avoiding using dynamic memory

•Objects should be allocated on the stack whenever possible to favour reusing already allocated objects:

- This reduce the number of allocations and improve performance

Optimization of Bertini INC module in the Geant4 framework

- The benchmark took 8.9 seconds to run before any optimizations were made:
  - By replacing the dynamic std::vector with an static stack allocated array the time was reduced to 5.4 s
  - The speedup was thus 8.9s/5.4s = 1.65

Memoization – caching results of repeated heavy calculations

- The bindingEnergyKummel method was found to be called repeatedly using the same parameters resulting in the same results:
  - A cache was implemented for storing already calculated values
  - When the function was called using the same parameters as previously then the binding-energy was found by using a simple lookup instead of performing the expensive calculations

Optimization of Bertini INC module in the Geant4 framework

- The std::map and std::hash_map classes from the C++ standard library makes the implementation of caches easy:
  - These are associative containers that stores key/value pairs and they perform well
  - A static cache object can be queried to check if the tuple of function parameters forming the key is contained in the cache:
    - If so, then the result can be returned immediately,
    - otherwise the rest of the original funtion is executed, and the resulting result is put into the cache.

- As a result of this optimization the runtime was reduced from 5.4 to 4.7 seconds, a speedup of 1.15

Optimization of Bertini INC module in the Geant4 framework

Using precomputed lookup-tables

- Computations of the form pow(A,1.0 / 3.0) and pow(A,2.0 / 3.0) were heavily performed throughout the code:
  - Lookup-tables were implemented to speedup calculations for integer values of atomic number A in the range [1,300]

The overall perfomance gain was about 10% when using these lookup tables.

Optimization of Bertini INC module in the Geant4 framework

Conclusion

- After these rather simple optimizations the overall runtime of the benchmark had dropped from 8.9s to 4.3s, which means a total speedup of about 2.1
- At this point it was hard to find any suitable spots in the code, where any optimization would have significantly reduced the runtime:
  - The output of the code profiler now showed a very flat time distribution
  - The code consisted completely of numerical computations made on double precision variables, and it did not contain any memory or IO intensive portions that otherwise are areas that benefit greatly from optimization