Progress Report Toward a Thread-Parallel Geant4

Gene Cooperman and Xin Dong
High Performance Computing Lab
College of Computer and Information Science
Northeastern University
Boston, Massachusetts 02115
USA
{gene,xindong}@ccs.neu.edu

Multithreaded Geant4 (Geant4MT)

- Master/Worker paradigm
- Event-level parallelism to simulate separate events by multiple threads
- For each event, there is a corresponding seed for CLHEP random number generator
- Seeds come from a sequence of random numbers on master
- *Reproducibility:* Given same initial random seed, Geant4MT produces same result.
- Efficiency for future many-core CPUs
- Testing and validation on today's 4-, 8- and 16-core nodes
- Preliminary results available based on testing on `fullCMS bench1.g4`

Current Results

`fullCMS bench1.g4` (electromagnetics), 1 master and 3 worker threads:

1. Phase I: multi-threaded implementation; code sharing (same as multiple processes), but no data sharing
   (600 MB: $\approx$ 30 MB text/code + 4 $\times$ 140 MB)
2. Phase II: Sharing of geometry, materials, particles, production cuts, EM physics tables
   (400 MB: $\approx$ 30 MB text/code + 80 MB shared geom. + 70 MB electromagnetics physics tables + 4 $\times$ 22 MB)
3. Phase III: Performance Analysis&Adjustment (Ongoing)
4. Phase IV: Other physics processes (Considering)
5. Phase V: General Software Schema: new releases of sequential Geant4 drive corresponding multi-threaded releases (TODO)

Phase IV: easy in principle, since physics tables are read-only, aside from small caches: Difficulty is that each physics table may have a distinct author using a distinct API

Multi-Processing: Forked Processes and Copy-On-Write

The UNIX fork system call uses copy-on-write semantics (COW) to create a child process that shares all data with the parent *until* the parent or child writes to a particular page. This provides easy sharing of those data pages that are accessed *only in read-only mode* by parent and child.

A Copy-On-Write version of Geant4 has been written. Its uses are two-fold:

1. *Reference version:* to compare Multi-Threaded Geant4 with best alternative technology.
2. *Easy Data Sharing:* few assumptions, less dependency on specific Geant4 source code.

Issues:

- *Coarser granularity:* If even one field of a C++ object is read-write, then the entire data page containing the object is not shared.
  Good news: The technique from Geant4 multithreading helps to solve this problem!
- *When to fork:* Geant4 initialization of different components happen prior to the first event. We chose to fork *after the detector construction*, which captures most of the initializations.

Geant4MT Methodology

- Patch parser.c of gcc to output static and global declarations in Geant4 source code; recompile and reinstall gcc
- Build Geant4 and collect output of parser.c (similar to UNIX grep)
  1. static variables in each function
  2. static class members
  3. global variables and if they exist, all corresponding “extern” declarations
- Transform the Geant4 source code using the “__thread” keyword
- Choose sharable classes and detect read-write data members
- Sharable class transformation: separate read-write data members and make them thread private.
  1. By detaching read-write data members, large read-only memory chunks are formed
  2. Copy-On-Write does not replicate those read-only chunks

Comparison of Different Parallel Approaches

- Separate Processes

  Requires the memory is sufficient and fast since no reduction for the memory footprint

- Copy-On-Write Using the Original Geant4

  May reduce the memory footprint on some special cases
  As an example, geometry is not replicated among all processes if no replica or parameterized volume is used.

- Geant4MT (Multithreaded Geant4): Process + Thread Parallel Geant4

  Guarantee to reduce the memory footprint greatly
  Adjust memory allocations& frees and scale to 16 threads on a Dell AMD 16-core of 2.0GHz.

- Copy-On-Write Using Geant4MT

  Apply the Geant4MT phaseII change only and reduce the memory footprint
  Unlimited scalability when the aggregation for results is not an issue.

Experimental Results For Different Approaches

Using the Dunnington machine to test on `fullCMS bench1.g4` with 24 workers and 4000 events per worker (electromagnetics).

Implementation	Total Memory on master	Additional Memory per Worker	Total Memory (master + 24 workers)	Runtime
Separate Processes	250 MB	250 MB	6 GB	4575 s
Original Geant4 + COW	250 MB	70 MB	2G MB	4571 s
Geant4MT + COW	250 MB	20 MB	730 MB	4540 s
Geant4MT 4 processes $\times$ 6 threads	250 MB	20 MB	730 MB	5798 s

If the memory and its bandwidth are sufficient, the multiprocesses are still powerful.

Current Work on Performance Analysis&Adjustment

Performance: Bottleneck (April 2009)

No more than 7 times speed-up on Dunnington

24 Intel 2.8G core	1	2	4	8	12	16	18	24
Time [s] 500 evts, pi-, 300GeV	5232	2902	1689	1088	926	871	830	810
Speed-up	1	1.8	3.1	4.8	5.7	6.0	6.3	6.5

Increasing number of system calls

24 Intel 2.8G core	24 processes × 1 thread		4 processes × 6 threads		2 processes × 12 threads		1 process × 24 threads	
Sys call	Number	Time	Number	Time	Number	Time	Number	Time
mmap	48	0.00	48	0.00	55	0.01	72	0.01
futex			12K	0.12	19M	225.64	55M	1068.22
madvise			1837	0.05	2172	0.08	2310	0.12
brk					19	0.00	101	0.00
munmap					9	0.00	38	0.00
...	...	...	...	...	...	...	...	...

Performance: Sources for Futexes

- The result from counting the output of strace shows two addresses that issue most of futexes for the 2 processes $\times$ 12 threads case:

Address	Number	Time
32558ded3e	6814549	85.9533490007106
32558ded6f	13147364	139.155402001531

- Backtracing breakpoints *0x32558ded3e and *0x32558ded6f in GDB, we see the following:

```
#0 0x00000032558ded3e in __lll_lock_wait_private (from /lib64/libc.so.6
#1 0x0000003255876944 in _L_lock_15349 () from /lib64/libc.so.6
#2 0x0000003255875901 in __libc_free (mem=<value optimized out>)
at malloc.c:3589
#3 0x0000000000b83334 in G4TouchableHistory::~G4TouchableHistory()
...
```

Performance: Sources for Futexes (Continue)

- More output

#0 0x00000032558ded6f in __lll_unlock_wake_private () from /lib64/libc.so.6
#1 0x000000325587695f in _L_unlock_15366 () from /lib64/libc.so.6
#2 0x000000325587592 8 in __libc_free (mem=<value optimized out>) at malloc.c:3592
#3 0x0000000000b83334 in G4TouchableHistory::~G4TouchableHistory()
#0 0x00000032558ded6f in __lll_unlock_wake_private () from /lib64/libc.so.6
#1 0x00000032558764c6 in _L_unlock_11790 () from /lib64/libc.so.6
#2 0x000000325587404e in __libc_malloc (bytes=<value optimized out>) at malloc.c:3538
#3 0x00002ac4f5501a6d in operator new (sz=1008) at ../../../../libstdc++-v3/libsupc++/new_op.cc:57
#4 0x00002ac4f5501ba9 in operator new[] (sz=46913458012192) at ../../../../libstdc++-v3/libsupc++/new_opv.cc:37
#5 0x0000000000c7a220 in G4AllocatorPool::Grow()

Performance: The Most Mallocs&Frees

- Using GDB scripts to output the backtracing information for all __libc_malloc and __libc_free occurrences
- Counting the number of mallocs/ frees for different locations of Geant4 code and figure out the most intensive one
- From 100000 times of mallocs or frees, there are

  free; G4SteppingManager::Stepping; 0xa17792 [14084]
  free; G4TouchableHistory::~G4TouchableHistory; 0xb83294 [14524]
  malloc; G4Transportation::PostStepDoIt; 0x9987a2 [15611]
  malloc; G4NavigationHistory::G4NavigationHistory; 0xb778e5 [15634]

- From 1000000 times of mallocs or frees, there are

  free; G4SteppingManager::Stepping; 0xa17792 [171418]
  free; G4TouchableHistory::~G4TouchableHistory; 0xb83294 [185433]
  malloc; G4Transportation::PostStepDoIt; 0x9987a2 [186017]
  malloc; G4NavigationHistory::G4NavigationHistory; 0xb778e5 [186042]

Performance: Mallocs&Frees Reduction

- G4TouchableHistory

  Override the “new” and the “delete” method such as:

  1. hold a set of thread-private instances with a flag to show the instances is “used” or “unused”;
  2. malloc 512 instances whenever there is no more unused instance;
  3. the “new” method: seach an unused instance and change the flag to be “used” before return the address of the instance;
  4. the “delete” method: only change the flag to be “unused” for the freed instance;

- G4NavigationHistory

  Change “vector<G4NavigationLevel> fNavHistory;” to

  “vector< G4NavigationLevel, MyAllocator< G4NavigationLevel > > fNavHistory;” where “MyAllocator” implements the same mechanism as mentioned above

Performance: More Mallocs&Frees Reduction

From 822366 times mallocs and frees, there are following mallocs and frees:

- free; ~G4QCandidate; DeleteQCandidate::operator(); std::for_each<__gnu_cxx::__normal_iterator<G4QCandidate**,; G4Quasmon::ClearQuasmon; 0x6998f1 [46361]
- malloc; operator; G4QNucleus::InitCandidateVector; 0x6c7c73 [53415]
- malloc; operator; G4QNucleus::InitCandidateVector; 0x6c7e1c [73594]
- malloc; operator; G4QNucleus::InitCandidateVector; 0x6c7d28 [85464]
- free; ~G4QCandidate; G4QNucleus::InitCandidateVector; 0x6c7c31 [165973]

Performance After Mallocs&Frees Reduction

24 Intel 2.8G core	24 processes × 1 thread		4 processes × 6 threads		2 processes × 12 threads		1 process × 24 threads	
Time	W-CLK	Sys	W-CLK	Sys	W-CLK	Sys	W-CLK	Sys
Worker 1	214.4	0.04	244.8	0.3	336.3	1.4	400	4.6
Worker 2	225.7	0.05	250.2	0.3	335.7	1.4	405.2	4.7
Worker 3	239.4	0.05	268.1	0.3	355.3	1.6	429.4	4.7
Worker 4	222.9	0.05	248.5	0.3	337	1.5	408.5	4.7
Worker 5	237.1	0.05	267.8	0.3	361.5	1.6	428.4	4.5
Worker 6	221.9	0.04	249.3	0.3	338.8	1.4	404.1	4.4
Worker 7	229.3	0.04	256.1	0.3	352	1.5	410	4.7
...	...	...	...	...	...	...	...	...
Worker 24	236.4	0.04	268	0.3	354	1.5	425.7	4.4
Average	228.15	0.04	254.82	0.33	343.48	1.43	413.23	4.47
× #Processes	5475.7		1019.27		686.96		413.23	
× #Threads		0.04		1.98		17.1		107.3
Speed-up			5.37		7.97		13.25	

Performance: On a Cheaper Hardware

16 AMD 2G core	16 processes × 1 thread		4 processes × 4 threads		2 processes × 8 threads		1 process × 16 threads	
Time	W-CLK	Sys	W-CLK	Sys	W-CLK	Sys	W-CLK	Sys
Worker 1	494.3	0.18	512.7	0.9	521.3	1.6	571.7	3.9
Worker 2	480.3	0.15	492.4	0.9	513.1	1.7	558.2	3.7
Worker 3	520.1	0.11	547	0.8	555.9	1.7	597.5	3.6
Worker 4	462.8	0.22	470.4	1	482.6	1.8	531.6	3.8
Worker 5	483.1	0.18	492.2	0.9	485.8	1.9	550.2	3.9
...	...	...	...	...	...	...	...	...
Worker 16	491.9	0.18	501.1	0.7	513.1	1.7	551.9	3.6
Average	500.62	0.19	516.72	0.84	527.48	1.72	573.14	3.75
×#Processes	8009.9		2066.88		1054.95		573.14	
×#Threads		0.19		3.38		13.75		60
Speed-up			3.88		7.59		13.98	

The two steps of Mallocs&Frees reduction greatly improves the scalability.

There are still a huge number of mallocs&frees left.

Performance: Corresponding Sys-Call Statistics

24 Intel 2.8G core	24 processes × 1 thread		4 processes × 6 threads		2 processes × 12 threads		1 process × 24 threads	
Sys call	Number	Time	Number	Time	Number	Time	Number	Time
mmap	48	0.000	48	0.001	49	0.002	53	0.007
mprotect			17829	0.235	15309	0.351	14912	0.622
futex			192	0.002	34793	0.464	59359	1.036
madvise			1787	0.055	1923	0.095	1980	0.119
brk					27	0.001	113	0.005
munmap					2	0.000	5	0.001

16 AMD 2G core	16 processes × 1 thread		4 processes × 4 threads		2 processes × 8 threads		1 process × 16 threads	
Sys call	Number	Time	Number	Time	Number	Time	Number	Time
mmap	32	0.001	32	0.002	32	0.002	32	0.008
mprotect			11384	0.283	12457	0.344	9341	0.81
futex			122	0.002	852	0.013	18912	1.438
madvise			1595	0.086	1922	0.108	1986	0.114
brk	1011	0.020	215	0.005	99	0.002	99	0.011

Questions?

Which Method is Best?

- Geant4MT achieves linear speedup through 6 threads on 2.88 GHz Intel Xeon with 24 cores: *4 $\times$ 6-core CPUs, plus L3 cache*
- Geant4MT achieves linear speedup through 16 threads on cheaper 2.0 GHz AMD Opteron with 16 cores: *4 times 4-core CPUs with no L3 cache*
- Geant4MT plus copy-on-write *always* achieves linear speedup in every experiment.
- Geant4MT plus copy-on-write uses more memory.
- *Conclusion:* Use Geant4MT with maximum number of threads achieving linear speedup. Add copy-on-write to occupy all cores.
  1. *Example:* 24 Xeon cores: 4×6 (four Geant4MT processes employing copy-on-write; each process having 6 threads)
  2. *Example:* 16 Opteron cores: 1×16 (a single Geant4MT process having 16 threads; no copy-on-write needed)

Automatically transform Geant4

- Follow the “change list”
- Transform the original Geant4 to be thread-safe
- Example for a static variable that is not a class member

BEFORE:

```
static G4FieldTrack endTrack( ’0’);
```

AFTER:

```
static __thread G4FieldTrack *endTrack_NEW_PTR_ = 0 ;
if ( ! endTrack_NEW_PTR_ )
  endTrack_NEW_PTR_ = new G4FieldTrack (’0’) ;
G4FieldTrack &endTrack = *endTrack_NEW_PTR_;
```

Automatically transform Geant4 (cont.)

- Example for a static class member

  BEFORE:

  static G4String dirName;

  AFTER:

  static __thread G4String dirName_NEW_PTR_;

  BEFORE:

  G4String G4NuclearLevelStore::dirName("");

  AFTER:

  __thread G4String G4NuclearLevelStore::dirName_NEW_PTR_ = 0;

  G4NuclearLevelStore* G4NuclearLevelStore::GetInstance()

  {if (! dirName_NEW_PTR_)

  dirName_NEW_PTR_ = new G4String("") ;

  G4String &dirName = * dirName_NEW_PTR_;

  . . . }

Sharable Class Transformation

Redefine the references for read-write data members

```
class G4PVReplica : public G4VPhysicalVolume
{
   int g4PVReplicaObjectOrder;
   static G4PVReplicaPrivateObjectManager g4PVReplicaPrivateObjectManager;
   ...
   // G4int fcopyNoG4PVReplica;
   ...
}

#define    fcopyNoG4PVReplica
((g4PReplicaPrivateObjectManager.offset[g4PVReplicaObjectOrder]).fcopyNo)
```

Sharable Class Transformation Continue

Implement the array for all thread-private data members

```
class ReplicaPrivateObject
{
public:
   G4int fcopyNo;
};
class G4PVReplicaPrivateObjectManager
{
public:
   ReplicaPrivateObject* privateDataArray;
   int MasterAddNew()...
   void WorkerInitialization()...
   void WorkerFree()...
}
```