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

ParGeant4

- A parallel version (separate processes) of Geant4 already exists
- `examples/extended/parallel/ParN02` (and ParN04)
- Master/Worker paradigm
- Utilize TOP-C “Raw Interface”
- Event-level parallelism to simulate separate events on remote processes
- For each event, there is a corresponding seed for CLHEP random number generator
- Seeds come from a sequence of random numbers on master
- No recompilation of Geant4 libraries
- Nearly linear speedup
- *Reproducibility:* Given same initial random seed, ParGeant4 produces same result.

Goal of Thread-Parallel Geant4

- Efficiency for future many-core CPUs
- Testing and validation on today's 4-, 8- and 16-core nodes
- New implementation (independent of ParGeant4, but Leveraging that experience)
- Unlike ParGeant4, requires re-compilation of some of Geant4 source code for thread-parallel operation
- Preliminary results available based on testing on `fullCMS bench1.g4`

Initial Results

Current results:
testing on `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) (DONE)

2. Phase II: Sharing of geometry, materials, particles, production cuts
(400 MB: $\approx$ 30 MB text/code + 80 MB shared geom. + 4 $\times$ 70 MB)
(DONE, UNDERGOING VALIDATION)

3. Phase III: Sharing of data for electromagnetic physics processes
(400 MB: $\approx$ 30 MB text/code + 80 MB shared geom. + 70 MB electromagnetics physics tables + 4 $\times$ *small* (TODO)

4. Phase IV: Other physics processes (TODO)

5. Phase V: General Software Schema: new releases of sequential Geant4 drive corresponding multi-threaded releases (TODO)

Phase III: 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.
- *When to fork:* Geant4 initialization of different components can happen prior to the first event, during the first event, or during later events (lazy initialization). We chose to fork *after the first event*, which captures most of the initializations.

Summary of Data Sharing

Measured for one master process/thread and three workers.
Testing on `fullCMS bench1.g4` (electromagnetics).
Sequential running time: 6 hours

Implementation	Total Memory on master	Additional Memory per Worker	Total Memory (master + 3 workers)	Runtime
Multi-Processing (COW)	180 MB	70 MB	400 MB	$2\frac{1}{4}$ hours
Multi-Threaded (Phase I)	180 MB	140 MB	600 MB	$2\frac{1}{4}$ hours
Multi-Threaded (Phase II)	180 MB	70 MB	400 MB	$2\frac{1}{4}$ hours
Multi-Threaded (Phase III) (estimate, not completed)	180 MB	small	180 MB	$2\frac{1}{4}$ hours

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

Status and Future Plans

Status:

- 10,000 lines of Geant4 modified automatically
- 100 lines of special cases modified by hand
- months of effort to find last few bugs (tested using fullCMS)
- goal of automatically patching each new Geant4 release

Note: We have seen that Phase I is multi-threaded, but all data is thread private. Hence, no bugs. Phases II and later introduce sharing of data. Testing on fullCMS, Geant4 unit tests, and other apps reveals bugs (additional special cases to be modified by hand).

Future Plans:

- Tool for automatically finding conflicts in data shared among threads.
- Measure cache misses (better indication of true sharing)
- Accurate measure of dynamic data sharing (shared pages accessed *per second*) versus non-shared pages *per second*

Related Work: Checkpoint-Restart

- DMTCP: `http://sourceforge.net/projects/dmtcp` (free, open-source)
- DMTCP (Distributed Multi-Threaded Checkpointing) is a tool to transparently checkpointing (copying to disk) the state of a computation
- It operates transparently (no modification of the user binary).
- In the case of Multi-threaded Geant4 or Multi-Process Geant4 (Copy-On-Write), it can checkpoint Geant4 *after* initialization. Restarting from this point saves time for production runs.

Recently, we demonstrated checkpointing of runCMS. RunCMS consists of up to 2 million lines of code and up to 700 dynamic shared libraries. A typical case (checkpointing 12 minutes after startup) is:

Size: 600 MB memory image (225 MB compressed on disk)
Dynamic shared libraries in RAM: 540
Time to checkpoint: 25 seconds; Time to restart from disk: 18 seconds

Questions?

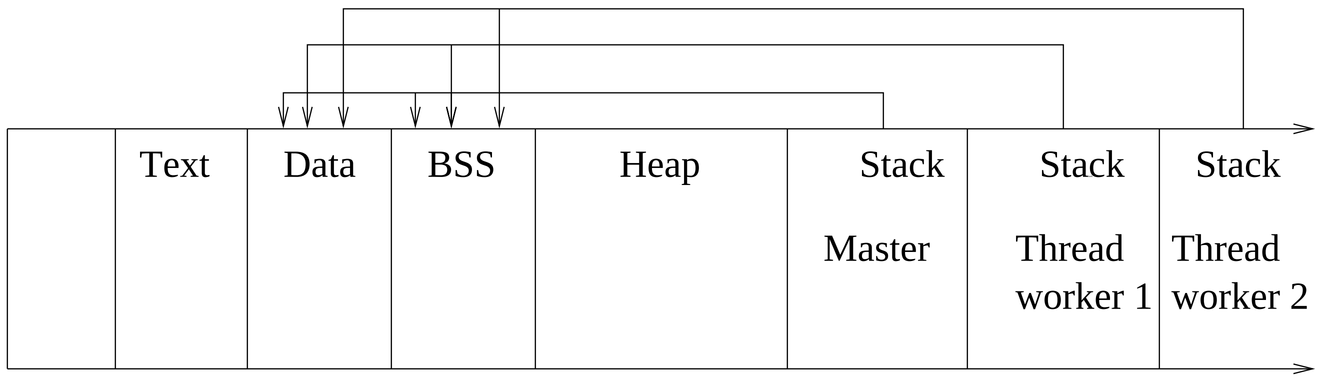
Details: Process Image Layout

- Text: This segment, also known as the code segment, holds the executable instructions of a program
- Data: This section holds all initialized data. Initialized data includes statically allocated and global data that are initialized
- BSS: This section holds uninitialized data
- Heap: This is used to grow the linear address space of a process
- Stack: This contains all the local variables that get allocated

Process master and workers:

	Text	Data	BSS	Heap	Stack
	Text	Data	BSS	Heap	Stack
	Text	Data	BSS	Heap	Stack

Details: Thread master and workers



ALTERNATIVE: Child created by forking from master process

- In Linux, child processes are given the same resources as their parents (including the address space). A child process does not duplicate the parent's resources and instead shares physical pages with its parent until one of them tries to write to a page. At that time, a copy of the page is made and assigned to the writing process. (copy-on-write)
- Disadvantage: Little opportunity for collaboration among multiple child processes; sharing of data via operating system is at a coarse level: share entire page of RAM or nothing; Memory bus from cache to RAM becomes a bottleneck.
- With multi-core CPUs, worker threads collaborate to access the same memory at the same time; fewer bottlenecks to RAM; collaborating threads is a goal of future work

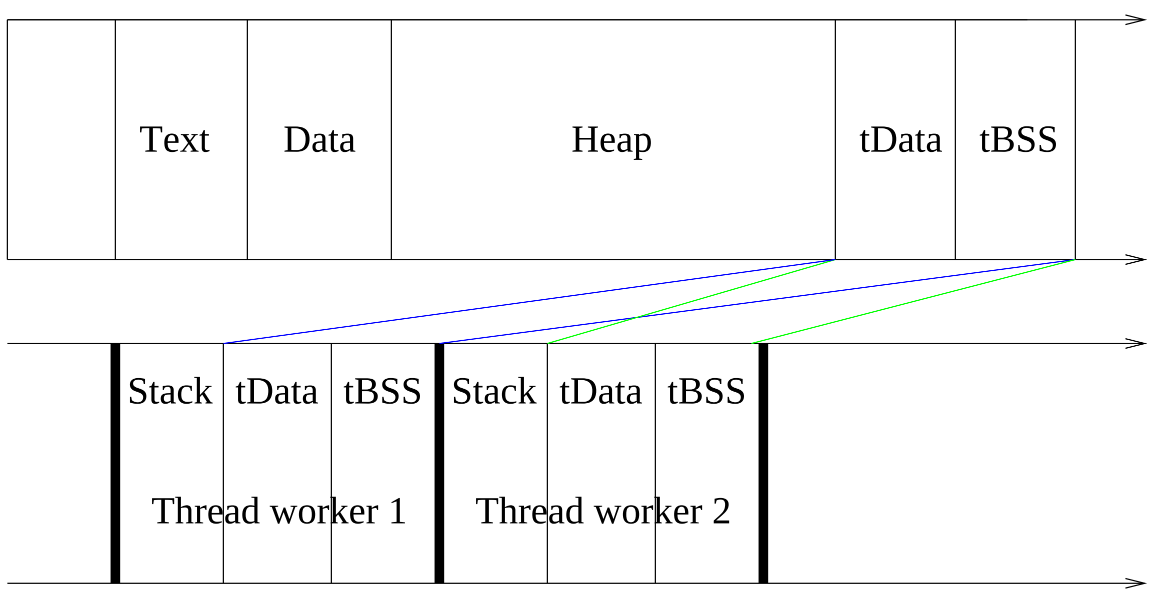
Details: Thread local storage (TLS): An example

```
#include <stdio.h>
#include <pthread.h>
__thread int gvar = 0; //int gvar = 0;
void *increase(void *)
{   gvar++;
    printf("Value in child thread: %d\n", gvar);
}
int main(int argc, char* argv[])
{  pthread_t tid;
    printf("Value in main thread: %d\n", gvar);
    pthread_create( &tid, NULL, increase, NULL );
    pthread_join(tid, NULL);
    printf("Value in main thread: %d\n", gvar);
    return 0;
}
```

```
Value in main thread:  0
```

Details: The usage of thread local storage (TLS)

- tData and tBSS segments



- statically initialized thread data does not support dynamic initialization
- “`static __thread int i = j;`” is not correct when j is a variable

Details: Thread local storage needed in 3 cases

- static class members

```
class G4StateManager
{
   ...
   static G4StateManager* theStateManager;
}
```

- static variables

```
G4double G4Navigator::ComputeStep(···
  static G4int sNavCScalls=0;
  ...
}
```

- global variables

```
G4Allocator< G4NavigationLevel> aNavigationLevelAllocator;
G4Allocator< G4NavigationLevelRep> aNavigLevelRepAllocator;
```

How to initialize a TLS variable dynamically

Fun(int j)	Fun(int j)
{	{
static int i = j;	static __thread int* i_NEW_PTR_ = 0;
	if (! i_NEW_PTR_) {
	i_NEW_PTR_ = new int;
	*i_NEW_PTR_ = j;
	}
	int &i = *i_NEW_PTR_;
i++;	i++;
return i;	return i;
}	}

The rule to add "__thread"

- Use a pointer whose name is new
- The initial value of the pointer is NULL
- Allocate space for the pointer only when the value of the pointer is NULL. Then assign the value with the original right side
- Refer to the value of the pointer using the original name
- This guarantees each variable is initialized once and only once

Questions?