Scalable Parallel Implementation of GEANT4 Using Commodity Hardware and Task Oriented Parallel C

George Alverson, Luis Anchordoqui, Gene Cooperman, Victor Grinberg, Thomas McCauley, Steve Reucroft, Edgar Salazar and John Swain

Northeastern University
Boston, MA, USA

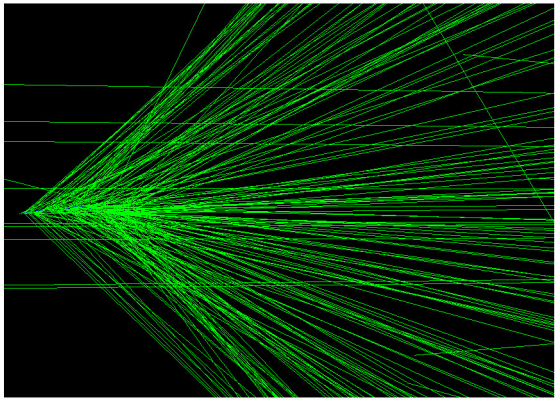
Features of TOP-C model

Primary Goals:

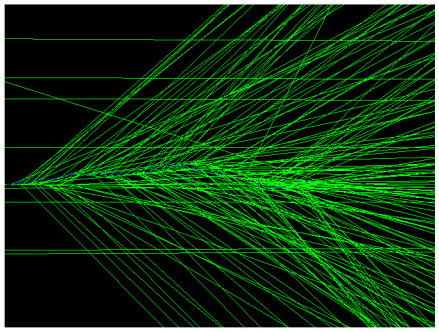
- Ease-of-programming (small number of primitives, embedded in familiar language, ports to C, LISP, GAP, Java, etc.)
- Good latency tolerance (for commodity hardware)

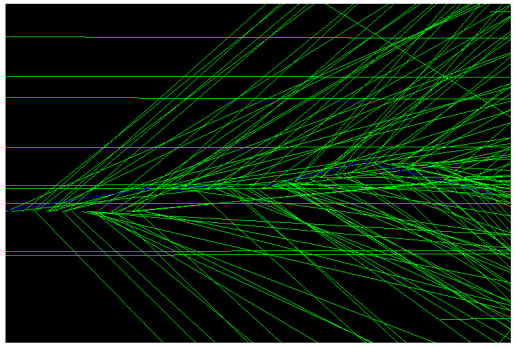
Secondary Goals:

- Re-use of "legacy" sequential code
- Simple, task-oriented programmer's model
- Natural load balancing
- Robustness (in presence of very slow or dead processes)
- Dynamic attachment of new processors
- Checkpointing
- Meta-computing on Web
- Runs on top of message-passing or shared memory model (same application code, linked with different library)
- Small model (small library of code — easily maintained and modified)



Tracking a Shower Through the Tank

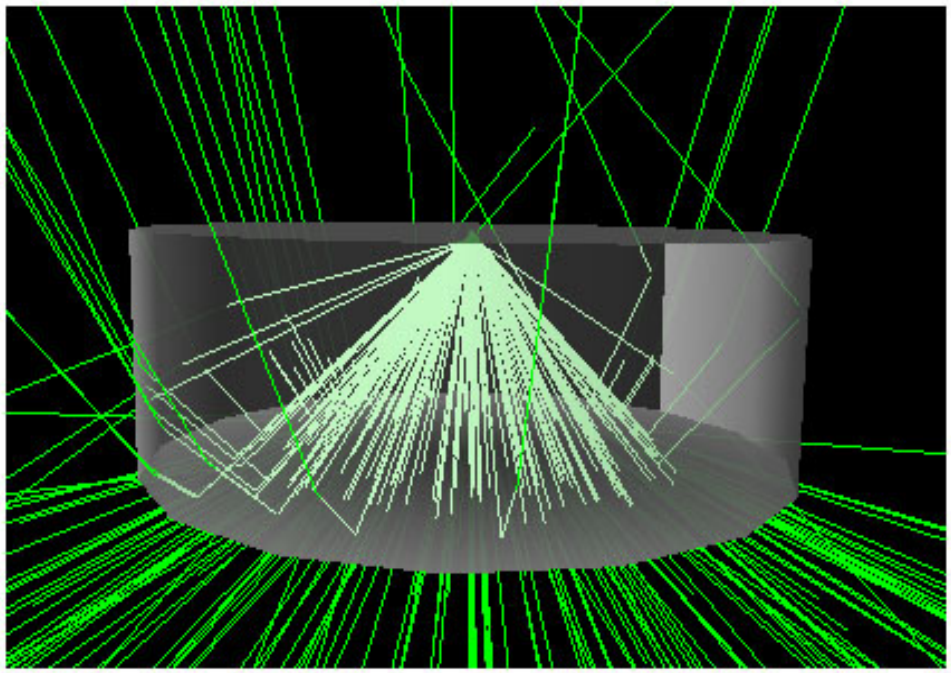




Three progressively closer views show the development of the shower of photons in an Auger detector tank produced by a 10 MeV electron (incident from the left).

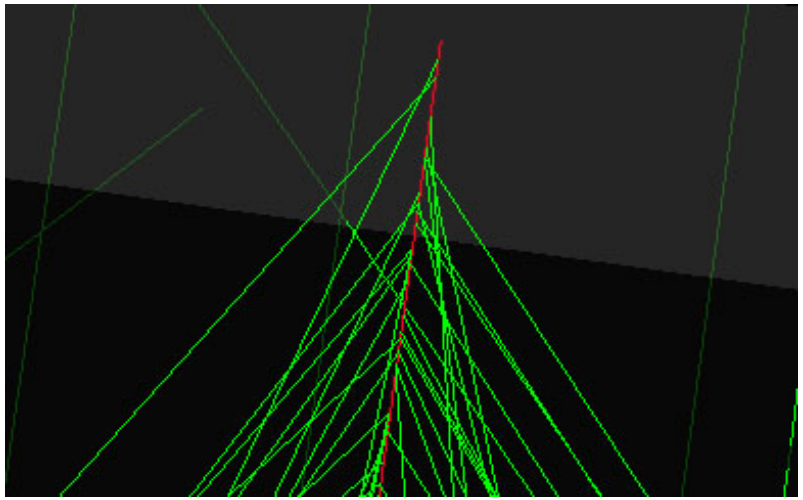
The Tank
(Surface Detector Station)

- 1.2 m in height
- 1.8 m in radius
- water filled
- triple phototube readout on top
- solar-powered radio telemetry
- 1500 tanks per location



10 MeV electron incident from top

Cherenkov photons (in green) exit or reflect from the sides of the gray (transparent) tank walls.

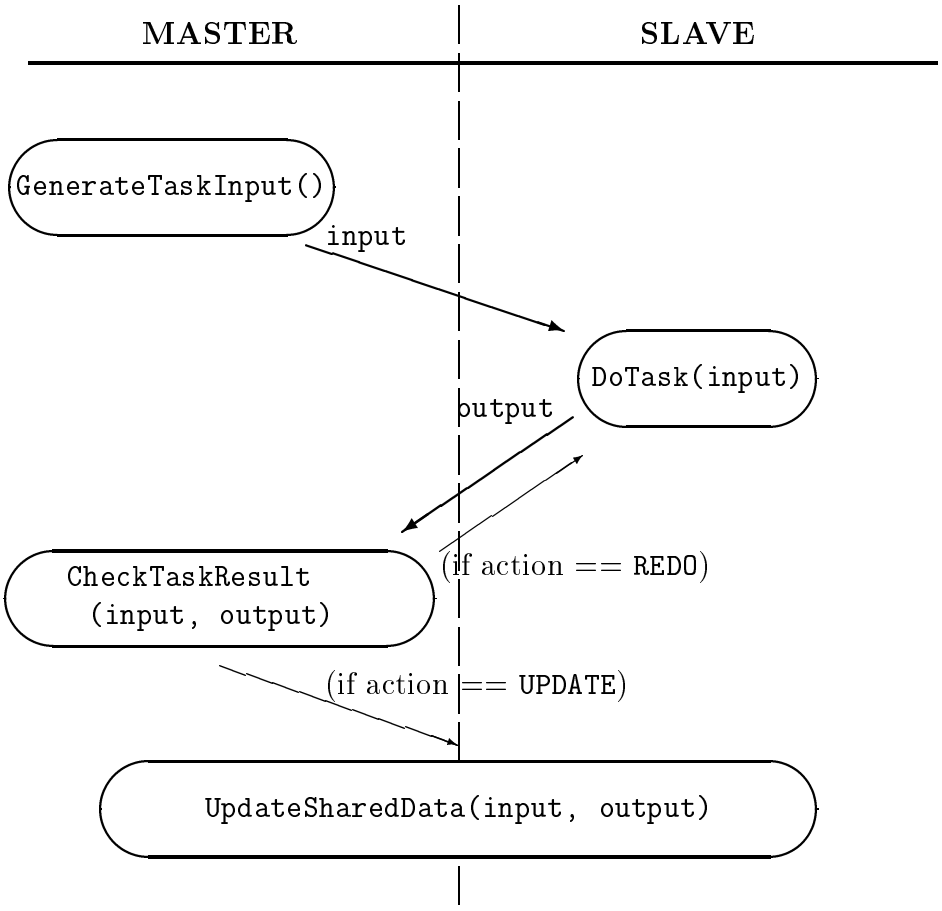


Shower Origination

Propagation Parameters

	gamma	e-	mu-	proton	neutron
Cut in range	1 mm	1 mm	1 mm	1 mm	1 mm
Cut in energy					
Air	990 eV	990 eV	48.4 keV	83.2 keV	990 eV
Water	2.9 keV	347 keV	3.39 MeV	8.95 MeV	990 eV

Life Cycle of A Task



TOP-C Programmer's Model

DISTRIBUTED FREE:

ftp://ftp.ccs.neu.edu/pub/people/gene/topc/

Concept 1: The Task

- Virtual star topology: single master process, many slave processes
- Task executed on a single slave process: *never interrupted*
- For a given *task input* and a given state of *shared data*, task produces a unique *task output*, independently of which slave process was invoked

Concept 2: The Shared Data

- **Definition:** Global, shared data, readable by all routines in all processes, but writable only by `UpdateSharedData()`
- Lazy updates: `UpdateSharedData()` called by TOP-C on a given slave *after* current task is complete and *before* next task

Concept 3: The Action

A task results in one of four actions:

- NO_ACTION: /* do nothing */;
- UPDATE: call UpdateSharedData(input, output);
  [on master and all slaves]
- REDO: Ask *same* slave to repeat calculation
  (executed only after *all* pending calls to UpdateSharedData)
  (Slave may have cached information from first computation)
- CONTINUATION: ;

Invocation of TOP-C

- Write application source code including:
  1. initialization of values of global variables (including shared data);
  2. definition of four TOP-C application functions: `GenerateTaskInput`, `DoTask`, `CheckTaskResult`, `UpdateSharedData`;
  3. invocation of parallelism:
     `MasterSlave(GenerateTaskInput, DoTask, CheckTaskResult, UpdateSharedData);`
- Build application binary:
  1. Compile application source using TOP-C include file; and
  2. Link application object file using TOP-C library.
  - **NOTE:** same source code can be re-compiled for sequential, message-passing, SMP, DSM, and other architectures
- Write `procgroup` file on master specifying:
  1. number of slave processes;
  2. which computer to use (hostname or Internet number) for each slave; and
  3. location of binary on slave computer
- Execute binary on master:
  This automatically invokes `MasterSlave()` in TOP-C library, which invokes MPI or other parallel library to start slave processes
- **NOTE:** Source code uses *SPMD style:* Master and all slave process execute identical code until they reach `MasterSlave()`

CheckTaskResult(): The Heart of a Parallel Algorithm

```
TOPC_ACTION CheckTaskResult( void *input, void *output )
{ if (output == NULL) return NO_ACTION;
  if (! is_up_to_date()) return REDO;
  return UPDATE; }
```

NOTE: There are only two library functions for the parallel programmer to know about: `master_slave()` and `is_up_to_date()`

- Strategy 1: Define task so most outputs result in `NO_ACTION`
  1. TRIVIAL PARALLELISM: collect results in private variable of master; Report results at end of execution
  2. SEARCH: Most search branches fail, eliminate those cases
- Strategy 2:
  1. Define `DoTask()` to cache partial results in private global variable, `partsOfTask`, in slave process.
  2. Define `UpdateSharedData()` to record in private global variable, `partsOfSharedDataModified`, which parts of the shared data were modified.
  3. Modify `DoTask()` to recognize when called in `REDO` action, and to use those portions of `partsOfTask` not affected by the changed shared data, as recorded in `partsOfShareDatayModified`.
- Strategy 3: Collect task outputs from multiple slaves, and merge on master. Then modify task input and output on master before calling `UpdateSharedData()`

Parallelizing Legacy Software: Geant4 Experience

100,000 lines of C++ code with STL for simulating particle showers.

1. The use of `.icc` (include) files to isolate our code from the original Geant4 code.
2. Collecting the code of the inner loop in a separate routine, `DoTask()`, whose input was a primary particl track, and whose output was the primary and its secondary particle.
3. Marhsalling and unmarshalling the C++ objects for particle tracks. (`gdb`, a symbolic debugger, and `etags`, an emacs-compatible code browser used to inspect internals)
4. Adding `TOPC_init()`, `TOPC_submit_task_input()`, etc.; Tested on marshalled particle tracks being sent across the network.
5. Finally, adding `CheckTaskResult()`, which inspected the task output, and added the secondary tracks to the Geant4 stack, for later processing by other slave processes.

Lessons from Parallelizing Geant4

1. Parallelization/Distributed computing is easier when application writers provide marshalling routines
2. TOP-C is economical: Geant4 stack, with its potentially large space requirements, ersides only on master
3. Network latency (approx. 10 ms – 100 ms) is an issue. However, TOP-C allows tasks to be bundled (*agglomerated*) together to amortize network latency over fewer messages.
4. Porting to shared memory under TOP-C is trival: replace distributed memory TOP-C library by shared memory TOP-C library

DISTRIBUTED FREE:

ftp://ftp.ccs.neu.edu/pub/people/gene/topc/