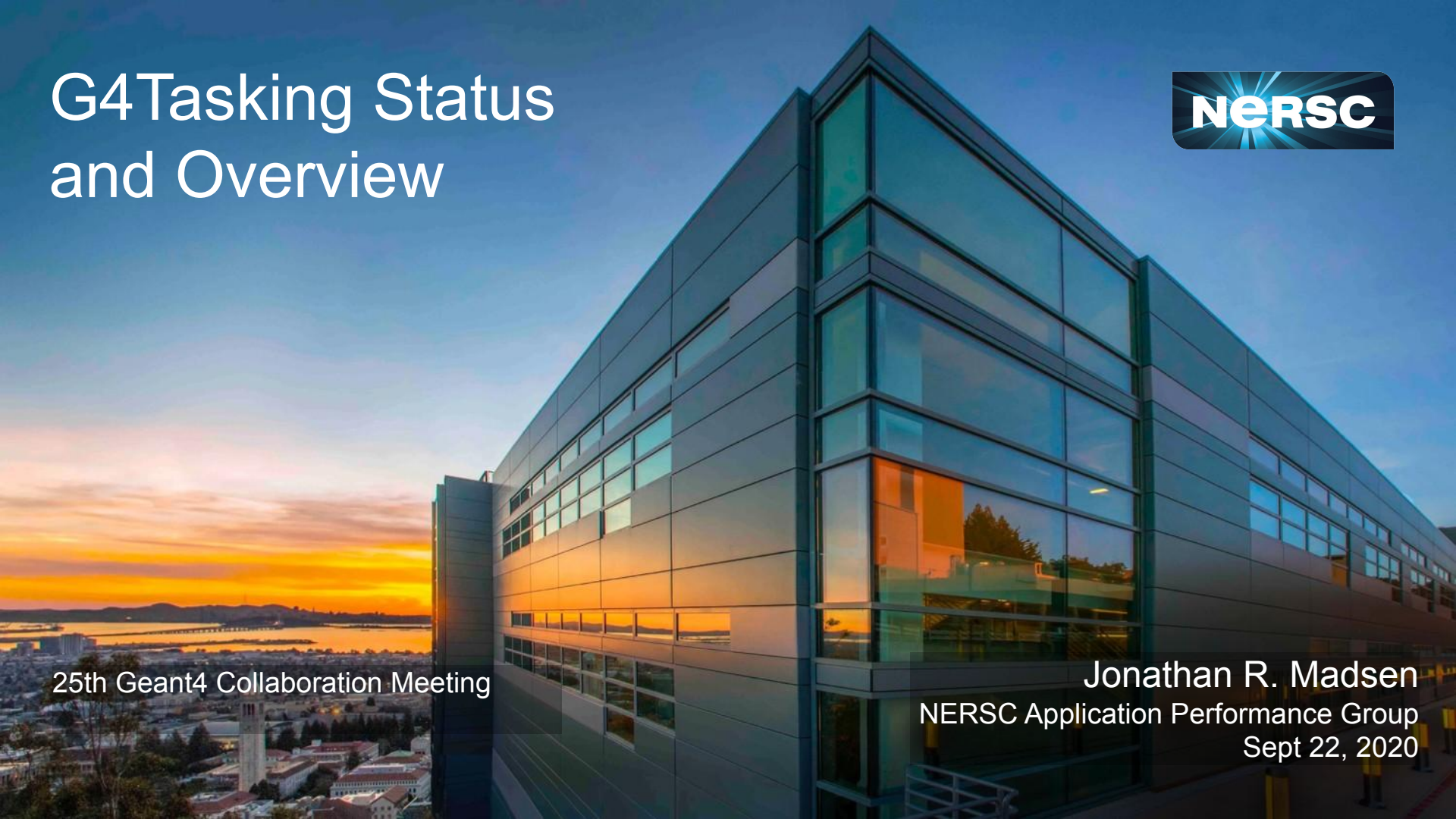
G4Tasking Status and Overview



25th Geant4 Collaboration Meeting

Jonathan R. Madsen
NERSC Application Performance Group
Sept 22, 2020

Tasking Overview

- It is important to remember that tasking is not fundamentally different than original MT
  - G4MTRunManager has primitive form of tasking
    - See G4WorkerRunManager::DoWork()
    - Essential difference is queue of functions vs. G4MTRunManager::WorkerActionRequest enum
- Standard workflow: invoke one function, get result, aggregate result, next iteration
- Tasking workflow: define aggregation functor, submit all functions to call, wait for final aggregation

Geant4 with Tasking Support

- Configure build the same as regular MT build
- Tasking is introduced in the event-loop
  - Introduces G4TaskRunManager
    - Has two modes: native mode (custom tasking) or TBB
- G4TaskRunManager creates default thread-pool
  - You can create additional thread-pools
- Limited support of custom task-queues
- Four run-manager modes now → run-manager should be created via factory function

G4RunManagerFactory

- Several overloads
  - First argument: string or enum for type of run manager
    - Strings are case-insensitive and use regex matching:
      - “^serial”, “^mt”, “^task”, “^tbb”
  - Other arguments: number of threads, fail if certain RM not available
- Run manager types
  - “default” / G4RunManagerType::Default
    - Provide parallel RM for parallel builds, serial RM for serial build
    - Env var: ‘G4RUN_MANAGER_TYPE’ to select non-default RM
  - Specifying an explicit RM → runtime error if RM not available
    - Env var: ‘G4FORCE_RUN_MANAGER_TYPE’ to override
- See also: source/tasking/README.md

Migration of Main: G4RunManagerFactory

```
#ifdef G4MULTITHREADED
#  include "G4MTRunManager.hh"
#else
#  include "G4RunManager.hh"
#endif

int main(int argc,char** argv)
{
#ifdef G4MULTITHREADED
 G4MTRunManager* runMan = new G4MTRunManager;
 runMan->SetNumberOfThreads(4);
#else
 G4RunManager* runMan = new G4RunManager;
#endif
 delete runMan;
}
```

```
#include "G4RunManagerFactory.hh"
int main(int argc, char** argv)
{
  // [Option #1] enum class G4RunManagerType:
  //   Default, Serial, MT, Tasking, TBB
  auto* runMan = G4RunManagerFactory::CreateRunManager(
     G4RunManagerType::Default, 4);

  // [Option #2] string:
  //    “default”, “serial”, “mt”, “task”, “tbb”
  auto* runMan = G4RunManagerFactory::CreateRunManager(
     "default", 4);
}
```

Using TBB Backend for Tasking

- Configure with -DGEANT4_USE_TBB=ON
- G4RunManagerFactory with G4RunManagerType::TBB
  - PTL::ThreadPool just configures tbb::global_control
- Use:
  - TBB normally in app (if desired)
  - G4TBBTaskGroup instead of G4TaskGroup
    - Syntactic sugar on top of tbb::task_group to provide capabilities of G4TaskGroup

Additional Configuration Settings

- Env var: “G4FORCE_GRAINSIZE=*N*”
  - Number of tasks, default is numEvents / poolSize
  - If grain-size == 50 and 500 events, 50 tasks of 10 events
  - Increasing the grain-size can improve load-balancing
- Env var: “G4FORCE_EVENTS_PER_TASK=*N*”
  - Alternative to setting the grain-size
  - If evts-per-task == 10 and 500 events, 50 tasks of 10 events
- Each task implies reseeding the RNG (i.e. **/run/eventModulo**)

Known Issues (1/3)

- Issue:
  - Previous run managers:
    - Fake run init + beam-on, run init, do event loop, terminate run
  - Tasking run manager:
    - Fake run init + maybe beam-on, maybe start run, one or more event loops, maybe terminate run but only if all event loops are done and run was started
- Result:
  - Some miscellaneous issues when # events < # threads
    - E.g. a thread which did not start a run tries to terminate a run

Comparison of Event Loop Processing

```
G4bool cond = ConfirmBeamOnCondition();
if(cond)
{
  numberOfEventToBeProcessed = n_event;
  numberOfEventProcessed     = 0;
  ConstructScoringWorlds();
  RunInitialization();
  DoEventLoop(n_event, macroFile, n_select);
  RunTermination();
}
```

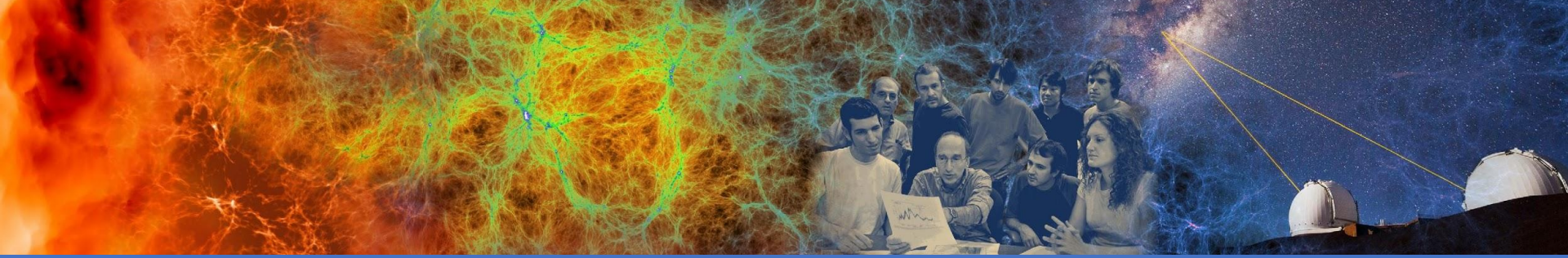
```
G4bool newRun                    = false;
const G4Run* run                 = mrm->GetCurrentRun();
G4ThreadLocalStatic G4int runId = -1;
if(run && run->GetRunID() != runId)
{
  runId  = run->GetRunID();
  newRun = true;
  if(runId > 0) ProcessUI();
}

if(newRun)
{
  G4bool cond = ConfirmBeamOnCondition();
  if(cond)
  {
    ConstructScoringWorlds();
    RunInitialization();
  }
}
DoEventLoop(nevts, macro, numSelect);
```

Known Issues (2/3)

- Issue:
  - Tasking isn't designed for "execute this function on all threads"
    - This ties into aforementioned issue(s)
    - TBB, in particular, does not make this easy
      - No direct access to thread creation/destruction
- Result:
  - Native is more stable than TBB
  - TBB is more prone to issues:
    - With multiple runs
    - Seg-faults when G4RunManager instance is destroyed
      - TLS are still "alive" after we have "terminated workers"

Known Issues (3/3)

- Issue:
  - G4MTRunManager / worker RM / kernel / etc. weren't designed for inheritance like G4RunManager, etc.
    - Explicit calls to static G4MTRunManager functions
    - Kernel creation via switch statements
- Result:
  - Changing geometries / physics after initialization is WIP
    - Current PR appears to solve most of issues, however



Mini-Tasking Tutorial

Tasking Basics

1. Thread-Pool (TP)
   - Handles backend implementation
     - Native PTL or TBB
   - Can create additional TPs in app
   - Can submit tasks directly to TP but recommend using task-groups
2. Task-Groups
   - G4TaskGroup → submits to native or executes directly
   - G4TBBTaskGroup → submits to TBB (if avail) or submits to native

- Task-groups implicitly use the TP created by G4TaskRunManager
- Task-groups provide (optional) thread-safe aggregation of return values from multiple tasks
  - Analogous to built-in parallel-reduce
  - Aggregation specified through "join" functor

G4TaskGroup Template Class

- **G4TaskGroup<T, U = T>**
  - T is aggregated type, U is type returned from tasks
- G4TaskGroup<void>
  - i.e. G4TaskGroup<void, void>
  - Performs no aggregation
- G4TaskGroup<vector<int>, int>
  - Tasks return integers, join function returns vector of integers
- G4TaskGroup<void, double>
  - Performs no aggregation but join function processes double

TaskGroup Example

```
// Target Function
G4Event* ProcessOneEvent(G4int event_id)
{
  auto rm = GetRunManager();
  auto em = GetEventManager();
  G4Event* evt = rm->GenerateEvent(event_id);
  em->ProcessOneEvent(evt);
  return evt;
}

// Analysis Function
void AnalyzeEvent(G4Event* evt)
{
   GetRunManager()->AnalyzeEvent(evt);
}
```

```
// Serial Execution
void ProcessEvents(G4int num_events)
{
  for(int i = 0; i < num_events; ++i)
  {
    G4Event* evt = ProcessOneEvent(i);
    AnalyzeEvent(evt);
  }
}
```

TaskGroup Example

```
// Target Function
G4Event* ProcessOneEvent(G4int event_id)
{
  auto rm = GetRunManager();
  auto em = GetEventManager();
  G4Event* evt = rm->GenerateEvent(event_id);
  em->ProcessOneEvent(evt);
  return evt;
}

// Analysis Function
void AnalyzeEvent(G4Event* evt)
{
   GetRunManager()->AnalyzeEvent(evt);
}
```

```
// Tasking Execution
void ProcessEvents(G4int num_events)
{
  using taskg_t = G4TaskGroup<void, G4Event*>;

  taskg_t tg(&AnalyzeEvent);

  for(int i = 0; i < num_events; ++i)
    tg.run(&ProcessOneEvent, i);

  tg.join();
}
```

TaskGroup Example

```
using taskg_t = G4TaskGroup<void, G4Event*>;
```

```
// Serial
void ProcessEvents(G4int num_events)
{
  for(int i = 0; i < num_events; ++i)
  {
    G4Event* evt = ProcessOneEvent(i);
    AnalyzeEvent(evt);
  }
}
```

```
// Tasking
void ProcessEvents(G4int num_events)
{
  taskg_t tg(&AnalyzeEvent);
  for(int i = 0; i < num_events; ++i)
    tg.run(&ProcessOneEvent, i);
  tg.join();
}
```

Tasking Gotchas

- Reference types in function parameters
  - Recent updates make copy of arguments when task is created
    - Referencing a variable created on stack can lead to corrupted parameters when function is actually invoked → stack variable will likely be out-of-scope
  - Lambda capture semantics can be used (carefully) to avoid expensive copies

```
void foo(int& val, int incr) { val += incr; }
void get_foo(TaskGroup<void>& tg)
{
   int val = 0;           // by the time foo(val, 1) is called, code will (probably)
   tg.run(&foo, val, 1);  // have returned from get_foo → val is invalid reference
}
```

Tasking Gotchas

- Using join() in recursively created task-groups
  - Each recursive execution results in +1 thread waiting for other tasks to finish processing → may result in deadlock

```
int fib(int n)
{
   if(n < 2) return n;
   auto _join = [](int& t, int v) { return t += v; };
   TaskGroup<int> tg(_join);
   tg.run(&fib, n - 1);
   tg.run(&fib, n - 2);
   // this WILL deadlock when # recursive calls
   // is equal to the number of threads in pool
   return tg.join();
}
```

```
int fib(int n)
{
   if(n < 2) return n;
   uint64_t        x, y;
   TaskGroup<void> tg;
   tg.run([&]() { x = fib(n - 1); });
   tg.run([&]() { y = fib(n - 2); });
   // this WILL NOT deadlock
   g.wait();
   return x + y;
}
```

Geant4 Example

- See: examples/extended/parallel/ThreadsafeScorers
  - TSRunAction.cc has example of processing results with a task-group inside a task-group
- Parallelizing this post-processing is simplified by C++ lambdas
  - Don’t waste time creating unnecessary member functions
  - Split the work into mini-functions
  - Capture any shared data by reference or value
  - Convert “private” data into parameters for lambdas