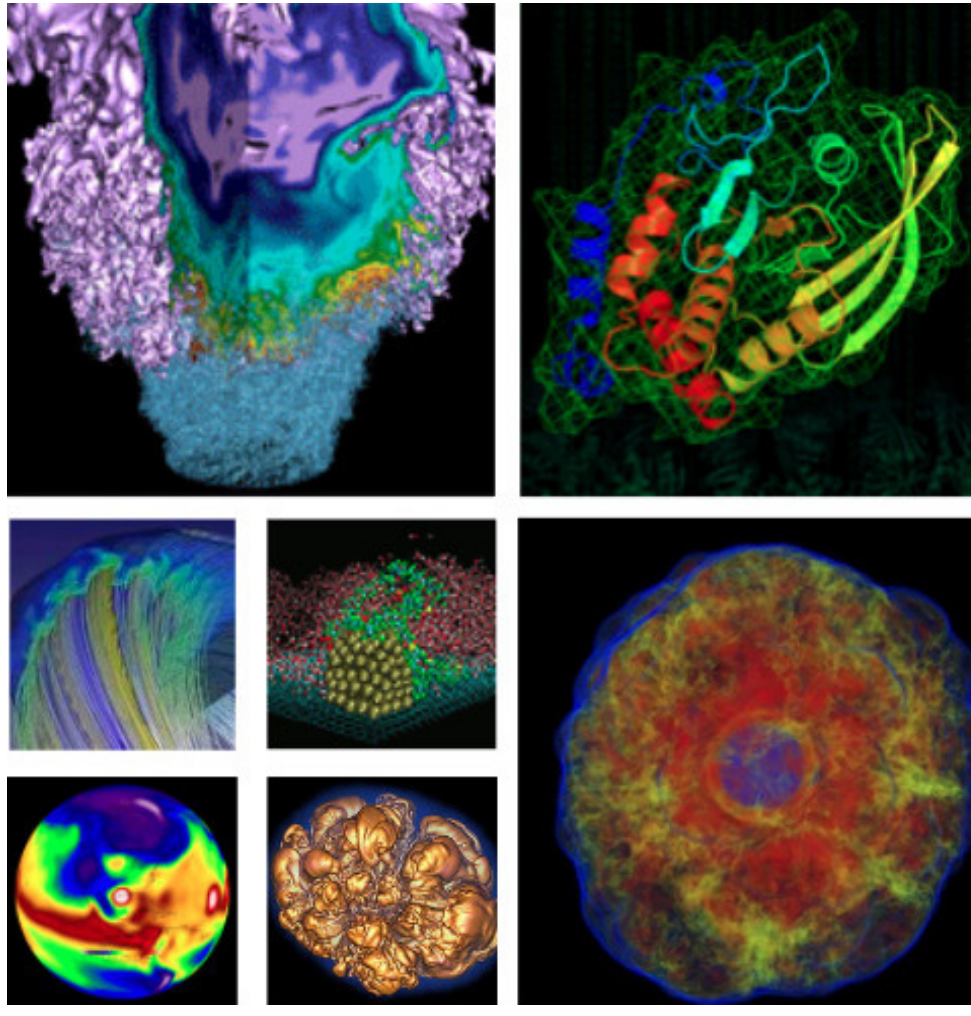
G4Tasking

Geant4 Task-based Parallelism



National Energy Research Scientific Computing Center

U.S. DEPARTMENT OF ENERGY | Office of Science

BERKELEY LAB

Jonathan R. Madsen

✉ jrmadsen@lbl.gov

National Energy Research Scientific Computing Center
Lawrence Berkeley National Laboratory

August 30, 2018

- New "tasking" framework for Geant4
  - Pool of threads without a predefined call-stack
  - Tasks are essentially function calls that are placed in a queue
  - Threads in pool are idle until tasks are placed in the queue
  - When the queue is empty, threads go back to sleep
- Task-based programming benefits
  - Faster startup and shutdown vs. threads
  - Higher-level thinking
  - Better load-balancing
- Threads don't run through application-defined call-stack – disassociates call-stack from threads
- Extended on top of v10.5 (beta). Migrated to our GitLab
  - Only available to internal users
  - https://gitlab.cern.ch/jmadsen/geant4-tasking

- Intel's Threading Building Blocks (TBB) is a very popular task-based programming library
  - Many constructs: parallel_for, parallel_reduce, parallel_do, parallel_invoke, task_group, pipelines, flow-graph, scalable memory allocators
  - Excellent library but it has some drawbacks
    - Relatively large overhead
    - Fundamental dependency for multi-threading
- OpenMP has supported "tasks" since version 3.0
  - Creating a task is easy but bundling into task-groups, doing other work, and joining later is difficult
  - OpenMP tasks tend to be inefficient

- C++11 has a number of features that make generic tasking relatively easy
  - `std::packaged_task`
  - `std::future`
  - `std::promise`
  - `std::forward`
  - `std::function`/`std::bind`/lambdas
  - variadic templates
- `std::future` and `std::promise` are paired together for asynchronous execution
- A promise is fulfilled by invoking `std::promise::set_value(...)` ⇒ signal to the paired future that the work has been completed
- Invoking `std::future::wait()` or `std::future::get()` will block on calling thread (with `std::condition_variable`) until the promise is fulfilled
- A `std::packaged_task` binds the function arguments to enable generic `void packaged_task::operator()()` invocation
- A `std::packaged_task` contains a `std::promise` and provides a `std::future` and invokes `std::promise::set_value(...)` after `operator()()` invocation
- Invoke `std::packaged_task()` ⇒ sets promise ⇒ notifies future

- No threads in thread-pool? Execute on calling thread ⇒ elimination of `G4RunManager` vs. `G4MTRunManager`
- MT Visualization does not need to create own threads ⇒ instantiate own thread-pool object or use global instance in `G4RunManager`
- Sub-event parallelism
  - Tasks are lightweight – essentially function pointers
  - Every `G4Track` could be a "task"
- Concurrent CPU/GPU execution
  - Allows more than one thread-pool per thread
  - When GPU is available, we can create a separate pool of threads for submitting tasks to GPU
  - Set of threads with affinity enabled ("CPU thread-pool") and another set of threads without affinity ("GPU thread-pool") is thus possible

- `G4ThreadPool`
- `G4TaskManager`
- `G4TaskAllocator classes`
- `G4VUserTaskQueue`
- `G4VTask`
- `G4VTaskGroup`
- `G4{TBB}Task<Ret, Arg = Ret>`
- `G4{TBB}TaskGroup<Ret, Arg = Ret>`

- `G4TaskAllocator` $\Rightarrow$ `G4Allocator` that does not delete `std::shared_ptr<G4Task>`
- `G4ThreadPool` has a `G4VUserTaskQueue` instance for scheduling the tasks
- `G4UserTaskQueue` is provided as default $\Rightarrow$ fast, scalable, thread-safe, and lock-free
- `G4TaskManager` (maybe `G4TaskHelper`) is associated with a `G4ThreadPool` instance
  - Constructs task object from function pointer and arguments being passed
  - Ensures proper insertion of task into task-groups
  - Acts as a proxy for communication with G4ThreadPool
  - Provides async function for generic execution of functions

- Generic tasking with native C++ with TBB backend, if desired
- Ideal setup for performance comparison
- `G4TBBTaskGroup` calls `tbb::task_group::run(...)` and `tbb::task_group::wait()` on internal instance of `tbb::task_group`
- `G4ThreadPool` instance does not create any threads
- `G4TaskGroup` instance has internal `tbb::task_group`
- `G4ThreadPool` instance has internal `tbb::task_group` (for async usage)
- `G4TBBTask` is essentially `G4Task` with `virtual bool is_native_task()` override to return false
- Essentially, there is an extension on top of TBB that enables tasks and task-groups to return and combine data

- `G4TaskGroup` is a template: `<typename _Ret, typename _Arg>`
  - `_Ret`: the return type from `join()` function
  - `_Arg`: return type of functions wrapped by tasks (defaults to `_Ret`)
  - Improvement over TBB $\Rightarrow$ all arguments and return types are void
  - Pass function to `G4TaskGroup` constructor specifying how to combine/store tasks

```
G4TaskGroup<void> void_task_group;

G4TaskGroup<int>
    int_task_group(
        [] (int& ref, int i) { return ref += i; });

G4TaskGroup<long, int>
    long_task_group(
        [] (long& ref, const int& i) { return ref += i; });

G4TaskGroup<list<int>, int>
    list_task_group(
        [] (list<int>& ref, int i) { ref.push_back(i); return ref; });
```

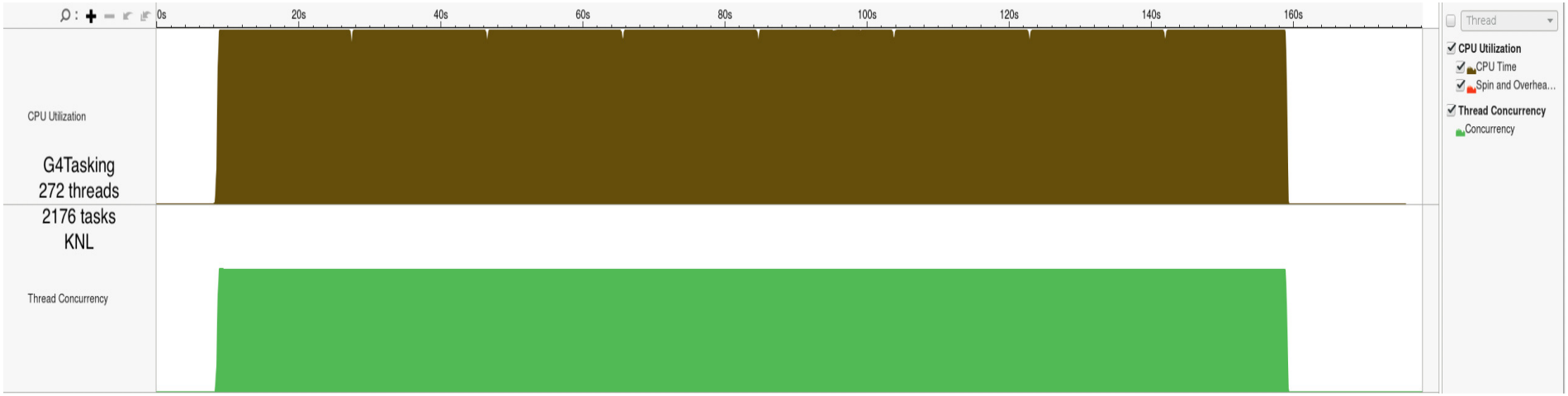
```cpp
// function being called
int fibonacci(const G4int& n)
{ return (n < 2) ? 1 : fibonacci(n-2) + fibonacci(n-1); }

// function specifying how to combine parallel results
auto join_func = [] (G4int& ref, G4int i) { ref += i; return ref; };

// bundle tasks into groups
G4TaskGroup<G4int> task_group(join_func);

// task_manager handles wrapping function into task
for(G4int i = 0; i < 8; ++i)
    task_manager->exec(task_group, fibonacci, 43);

// async interface
G4Future<G4int> fib = task_manager->async<G4int>(fibonacci, 43);

// ... continue working on CPU if desired, including generating more work ...
// block until all tasks in group have finished
int answer = task_group.join();
assert(answer == fib.get() * 8);
```

```cpp
typedef std::vector<double> Vector;

// sum double results
auto sum_join = [] (double& ref, double i) { ref += i; return ref; };
// store double results in a vector
auto vec_join = [] (Vector& vec, double i) { vec.push_back(i); return vec; };

G4TaskGroup<double> sum_task_grp(sum_join); // Construct the task-groups with
G4TaskGroup<Vector, double> vec_task_grp(vec_join); // their join function

// ... add tasks to task-groups ...

double joined_sum = sum_task_grp.join(); // Wait until finished and
Vector joined_vec = vec_task_grp.join(); // return combined result
```

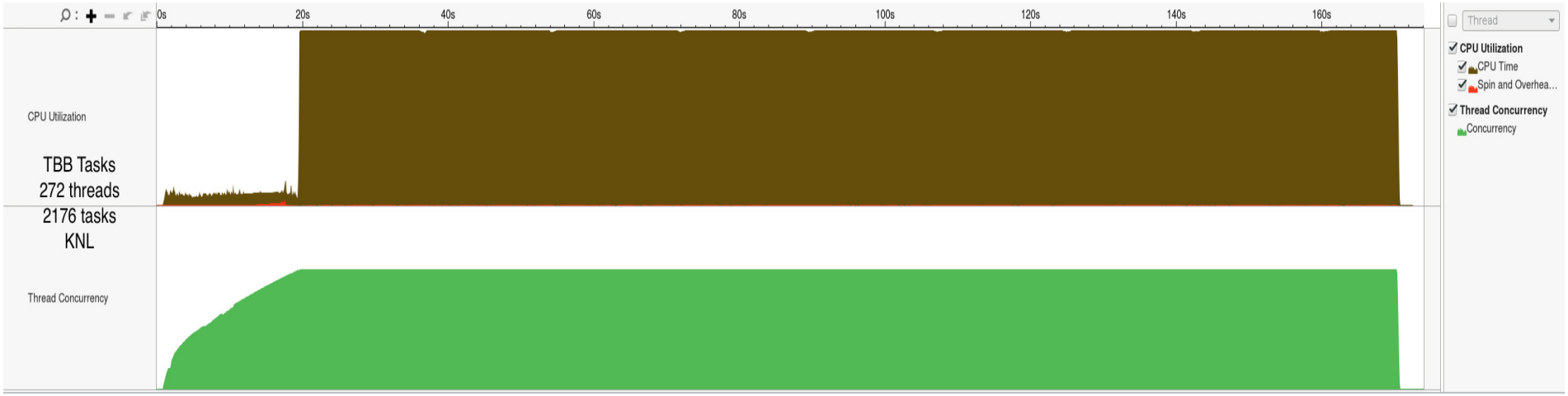
- Composed of N + 1 sub-queues where N ≡ worker threads
- Sub-queues accessed in lock-free manner

```
inline bool G4TaskSubQueue::AcquireClaim()
{
    bool is_avail = m_available.load(std::memory_order_relaxed);
    if(!is_avail) { return false; }
    return m_available.compare_exchange_strong(is_avail, false,
            std::memory_order_release, std::memory_order_relaxed);
}
// loop for finding work in sub-queue
for(G4int i = 0; i < (m_workers + 1); ++i)
{
    // get subqueue, note use "n" instead of "i"
    G4TaskSubQueue* task_subq = m_subqueues[n];
    if(task_subq->AcquireClaim())
    {   // if i == GetThreadBin(), pop from top
        // if i != GetThreadBin(), pop from bottom (work-stealing)
        G4VTaskPtr task = task_subq->PopTask(i == GetThreadBin());
        task_subq->ReleaseClaim();
        if(task.get())
            return task;
    }
}
```

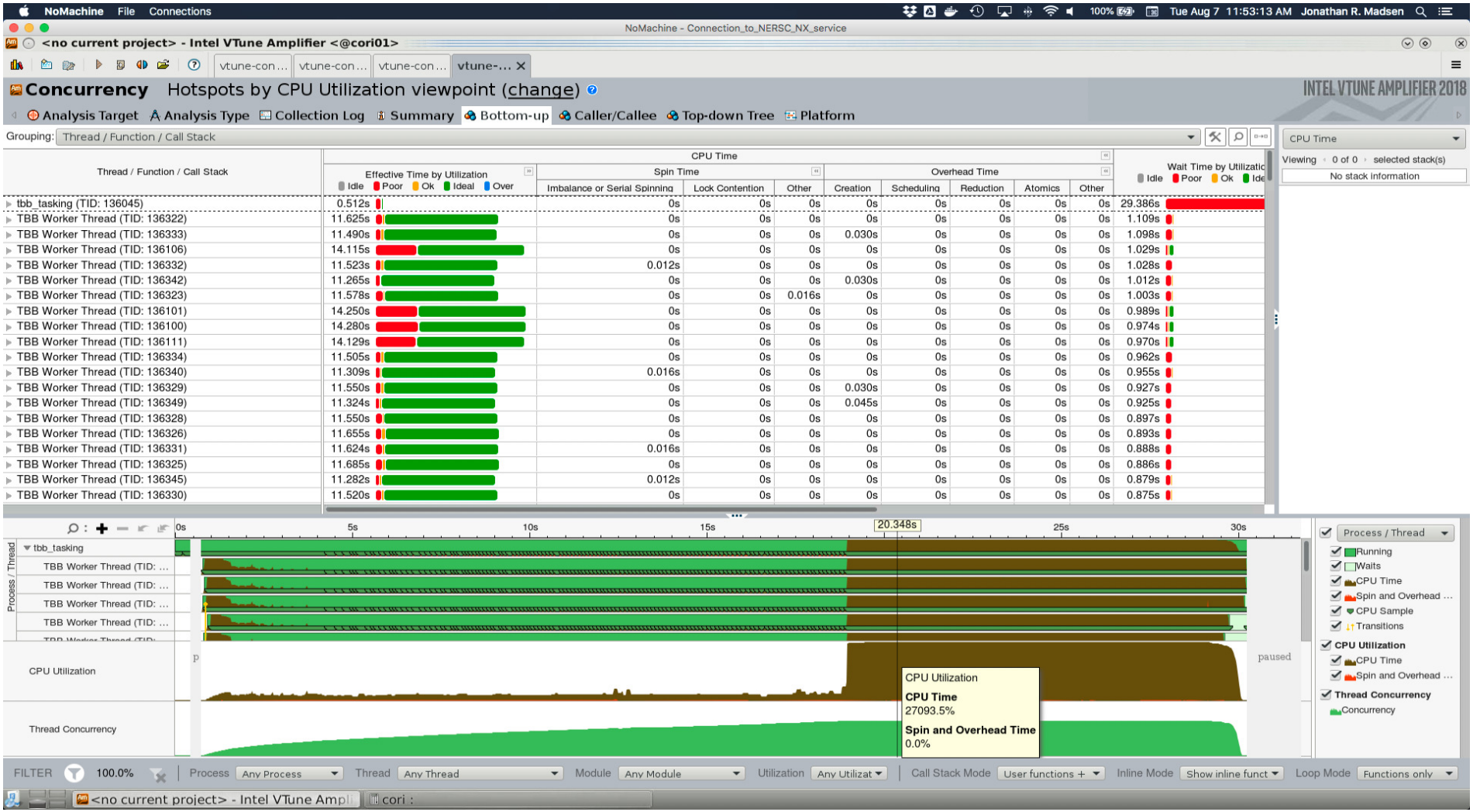
- Geant4 Tasking



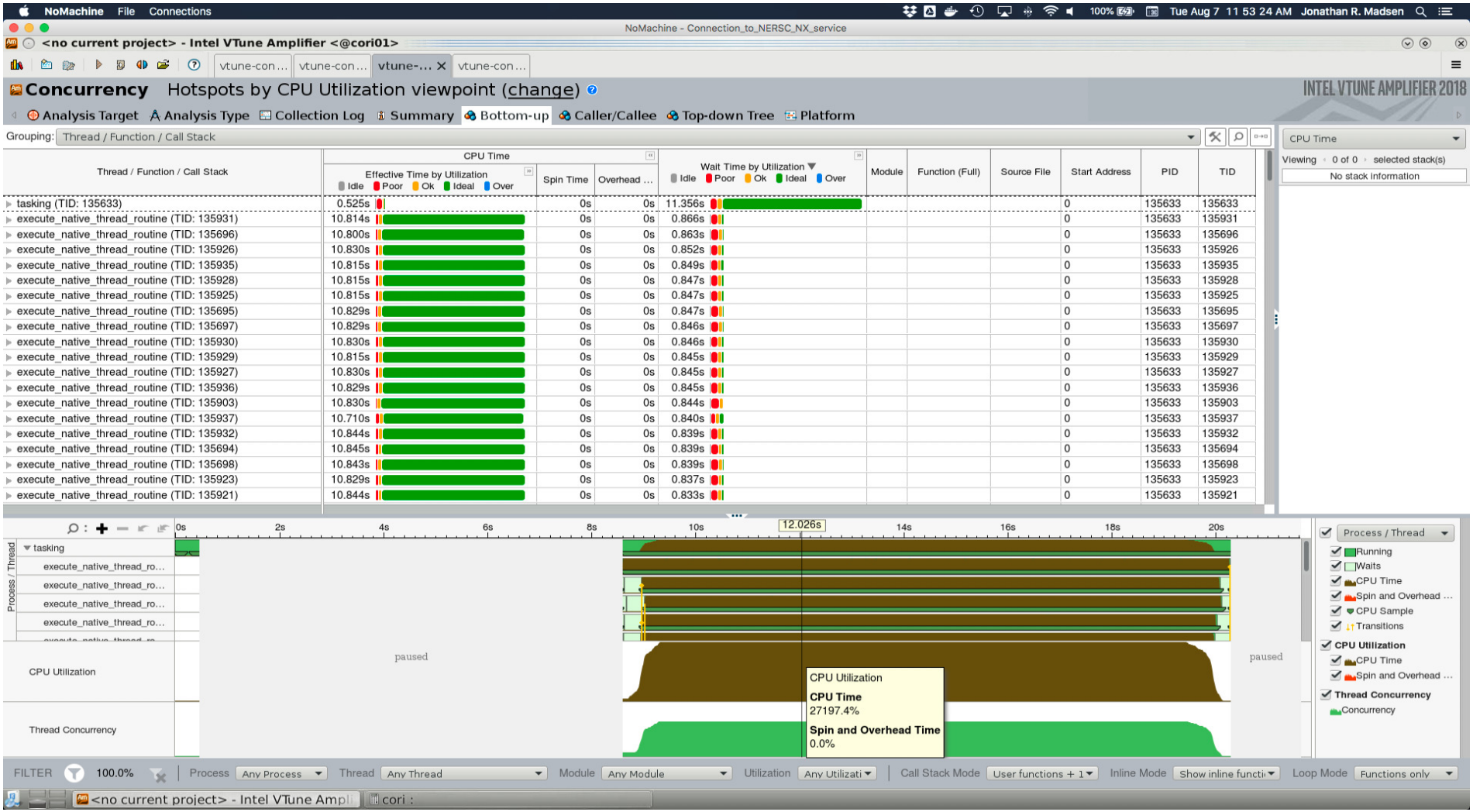
- TBB Tasking



TBB Tasking


NoMachine File Connections
NoMachine - Connection_to_NERSC_NX_service
<no current project> - Intel VTune Amplifier <@cori01>
Concurrency Hotspots by CPU Utilization viewpoint (change)
INTEL VTUNE AMPLIFIER 2018
Analysis Target Analysis Type Collection Log Summary Bottom-up Caller/Callee Top-down Tree Platform
Grouping: Thread / Function / Call Stack
Thread / Function / Call Stack
CPU Time
Effective Time by Utilization
Idle Poor Ok Ideal Over
Spin Time
Imbalance or Serial Spinning
Lock Contention
Other
Overhead Time
Creation
Scheduling
Reduction
Atomics
Other
Wait Time by Utilizatic
tbb_tasking (TID: 136045) 0.512s 0s 0s 0s 0s 0s 0s 0s 0s 29.386s
TBB Worker Thread (TID: 136322) 11.625s 0s 0s 0s 0s 0s 0s 0s 0s 1.109s
TBB Worker Thread (TID: 136333) 11.490s 0s 0s 0s 0.030s 0s 0s 0s 0s 1.098s
TBB Worker Thread (TID: 136106) 14.115s 0s 0s 0s 0s 0s 0s 0s 0s 1.029s
TBB Worker Thread (TID: 136332) 11.523s 0.012s 0s 0s 0s 0s 0s 0s 0s 1.028s
TBB Worker Thread (TID: 136342) 11.265s 0s 0s 0s 0.030s 0s 0s 0s 0s 1.012s
TBB Worker Thread (TID: 136323) 11.578s 0s 0s 0.016s 0s 0s 0s 0s 0s 1.003s
TBB Worker Thread (TID: 136101) 14.250s 0s 0s 0s 0s 0s 0s 0s 0s 0.989s
TBB Worker Thread (TID: 136100) 14.280s 0s 0s 0s 0s 0s 0s 0s 0s 0.974s
TBB Worker Thread (TID: 136111) 14.129s 0s 0s 0s 0s 0s 0s 0s 0s 0.970s
TBB Worker Thread (TID: 136334) 11.505s 0s 0s 0s 0s 0s 0s 0s 0s 0.962s
TBB Worker Thread (TID: 136340) 11.309s 0.016s 0s 0s 0s 0s 0s 0s 0s 0.955s
TBB Worker Thread (TID: 136329) 11.550s 0s 0s 0s 0.030s 0s 0s 0s 0s 0.927s
TBB Worker Thread (TID: 136349) 11.324s 0s 0s 0s 0.045s 0s 0s 0s 0s 0.925s
TBB Worker Thread (TID: 136328) 11.550s 0s 0s 0s 0s 0s 0s 0s 0s 0.897s
TBB Worker Thread (TID: 136326) 11.655s 0s 0s 0s 0s 0s 0s 0s 0s 0.893s
TBB Worker Thread (TID: 136331) 11.624s 0.016s 0s 0s 0s 0s 0s 0s 0s 0.888s
TBB Worker Thread (TID: 136325) 11.685s 0s 0s 0s 0s 0s 0s 0s 0s 0.886s
TBB Worker Thread (TID: 136345) 11.282s 0.012s 0s 0s 0s 0s 0s 0s 0s 0.879s
TBB Worker Thread (TID: 136330) 11.520s 0s 0s 0s 0s 0s 0s 0s 0s 0.875s
CPU Time
Viewing 0 of 0 selected stack(s)
No stack information
0s 5s 10s 15s 20.348s 25s 30s
Process / Thread
tbb_tasking
TBB Worker Thread (TID: ...
CPU Utilization
Thread Concurrency
paused
CPU Utilization
CPU Time
27093.5%
Spin and Overhead Time
0.0%
Process / Thread
Running
Waits
CPU Time
Spin and Overhead ...
CPU Sample
Transitions
CPU Utilization
CPU Time
Spin and Overhead ...
Thread Concurrency
Concurrency
FILTER 100.0% Process Any Process Thread Any Thread Module Any Module Utilization Any Utilizat Call Stack Mode User functions + Inline Mode Show inline funct Loop Mode Functions only

Geant4 Tasking



- Limitation for **recursive** task-groups in fibonacci tests
  - Serial compute time: ~ 9.5 seconds
  - Investigating but doubt we will ever encounter this situation in Geant4