Implementation of a Multi-threaded Framework for Large-scale Scientific Applications

Transitioning CMS to a Hierarchical Threaded Framework

Elizabeth Sexton-Kennedy *FNAL*
Christopher Jones *FNAL*
Patrick Gartung *FNAL*
David Lange *LLNL*
On behalf of CMS Offline

Outline

Goals

Design

Thread Safety

Tools

Performance

Goals

Better scaling of system resources as core count increases

- Puts less burdens on existing grid sites and workflow management since one batch slot uses more cores
- Flexibility: Potential to use sites with lower available resources

Reduce latency of processing, not necessarily increasing throughput

- Prompt reconstruction can finish processing a big file faster

More sharing between cores

- Share infrequently updated memory
  1. Conditions
  2. I/O buffers
- Share file handles
- Share network connections

Minimize changes to existing framework

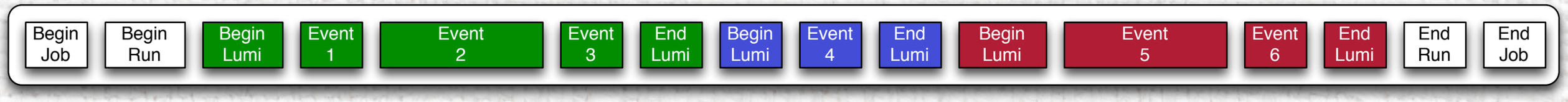
Design

CMS


Button
Center back
Straight grain of
Fold line
Center front line
Seam line
Simplicity 4700
ONE SIZE DOLL
COAT AND SLEEVE FRONT VIEW 2
½" Hem allowed
½" Hem allowed
Simplicity 4700
ONE SIZE DOLL
SCARF VIEW 1
Straight grain of fabric
Center back
Seam line

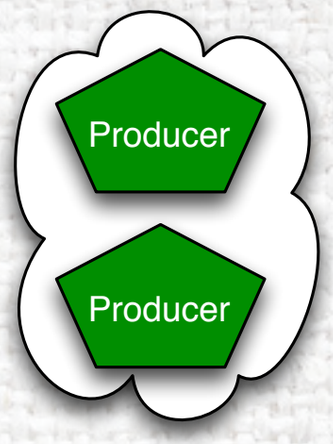
Legacy Design

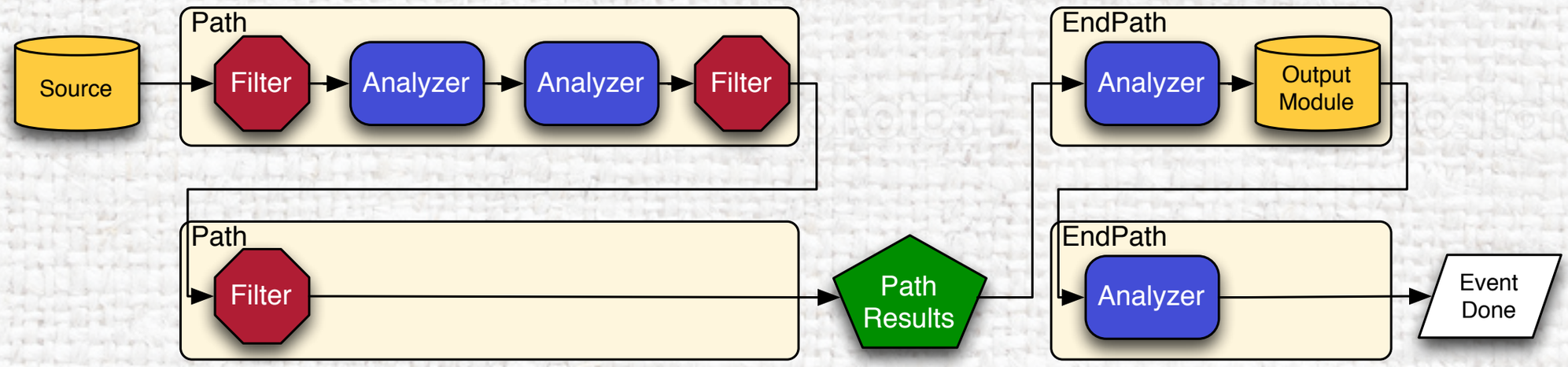
State Transitions



Event Processing

- Algorithms are encapsulated into modules





Threaded Design

Run multiple transitions, i.e. events, concurrently

- Introduces new concepts: *Global* and *Stream*

Within one event run multiple modules concurrently

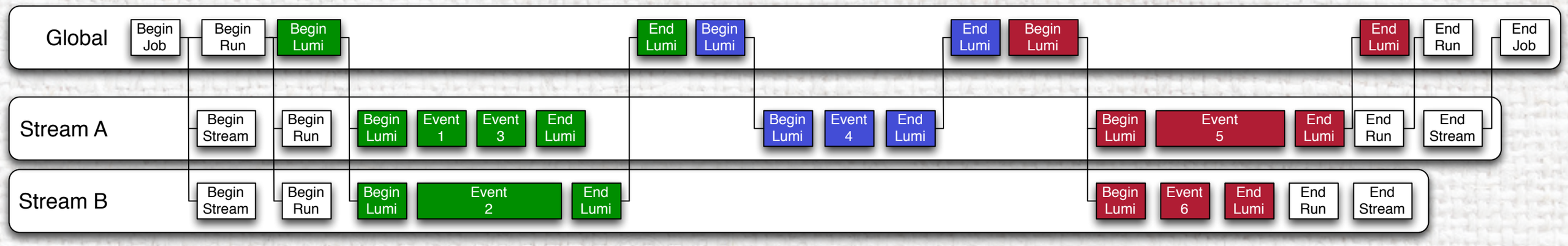
- Have to take into account module dependencies
- Want to minimize any required changes to module code

Within one module be able to run multiple tasks concurrently

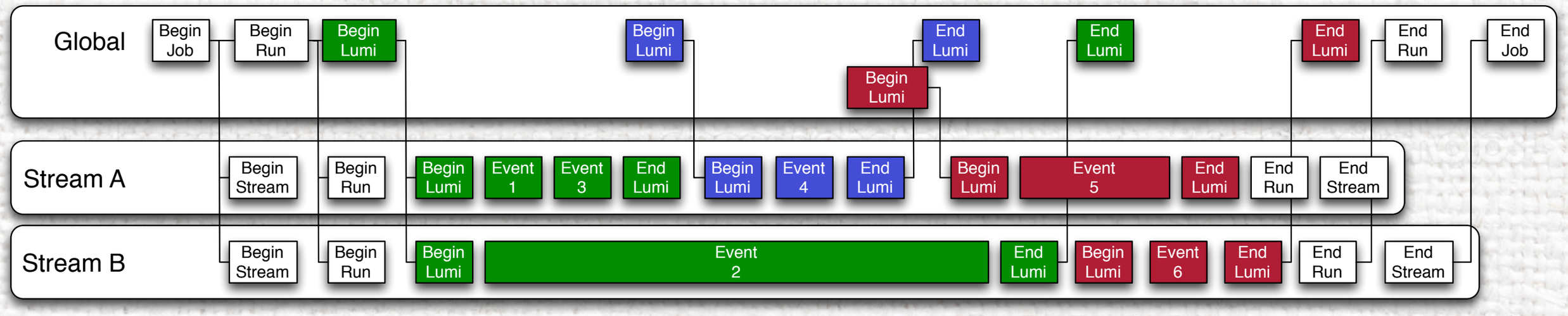
Intel's Threaded Building Blocks used for all of the above

- Break down work into 'tasks' and TBB can run the tasks in parallel
- http://threadingbuildingblocks.org

Serial Non-Event Transitions



Concurrent Transitions



Concurrent Tasks

Can use TBB directly inside a module

- TBB will handle scheduling tasks for both modules and sub-modules

TBB has some convenience functions

```
std::vector<Results> results(input.size(),Results());
tbb::parallel_for(0U,input.size(), DoWork(results) );
```

Can create own tasks for complex algorithms

```
class MyTask : public tbb::task { ... };
...
MyTask* mt = new (tbb::task::allocate_root()) MyTask;
tbb::task::spawn_root_and_wait( mt );
```

Users tasks must finish before returning from module

Thread-Safety

Thread Safety

Data Products

- Information passed from module to module
- Framework only provides 'const' access to data products
- 'const' member functions must be thread safe
  - Matches C++11 thread-safety guarantee for containers

Modules

- Majority of user defined code
- Different module varieties define different levels of thread safety
  1. Stream
  2. Global
  3. One
  4. Legacy

Stream Module

Replicate an instance of a module configuration for each Stream

- E.g. if have 8 Streams in a job will have 8 copies of a module

A Stream only processes one Event at a time

- A module copy will only be called at most once per event
- Member data does not have to be thread safe

One Stream only sees a fraction of the Events in the job

- Therefore a module copy only sees a fraction of the events
- Not a problem for most Producers and Filters

Easy to convert from Legacy to Stream interface

```
class TrackClusterRemover : public stream::Producer<> {
…};
```

Global Module

One instance of a module shared by all Streams

- One module sees all Runs, LuminosityBlocks and Events

All member functions and member data must be thread-safe

- Member functions called on each transition are 'const'
- The interface provides ways to help you with thread-safety
  1. per transition caching

```
class Counter : public global::Analyzer<StreamCache<int>> {
...
   void analyze(StreamID id, Event const& event) const {
     ++(*streamCache(id) ); }
};
```

Only use if

- Need to share as much memory across Streams as possible or Algorithm must see all Runs, LuminosityBlocks or Events

High performance OutputModules would be Global

One Module

One instance of a module shared by all Streams

- One module sees all transitions

Module instance sees only one transition at a time

- Framework guarantees the serialization
- Member data does not need to be thread-safe

Can use a resource shared across different modules

- Modules declare the use of the resource
- Framework guarantees only one module using the resource runs at a time
- Can call code which uses "static"
    1. E.g. legacy FORTRAN based MC event generators

```
class NTupleMaker : public one::Analyzer<> {
…};
```

Easy to convert from Legacy to One interface

Legacy Module

Modules which have not been ported to new interface

- Just need to recompile

Only one legacy module will run at a time

- Have to assume the modules can interfere with one another
- Performance problem

Eases code migration

Module Interface Change aka. Consumes Migration

Framework orchestration requires interface change for Modules

Up until now, Modules only needed to declared the DataProducts that they produce.

Now Modules must also declare what they consume so that can be scheduled in a way that avoids deadlocks.

This is the largest interface migration. Over 2270 modules to convert, however only about 10% are needed for the reconstruction (our primary interest).

Pushing for Concurrency

We started this migration in January of this year.

It was certainly a herding cats exercise requiring a lot of attention from our RECO conveners.

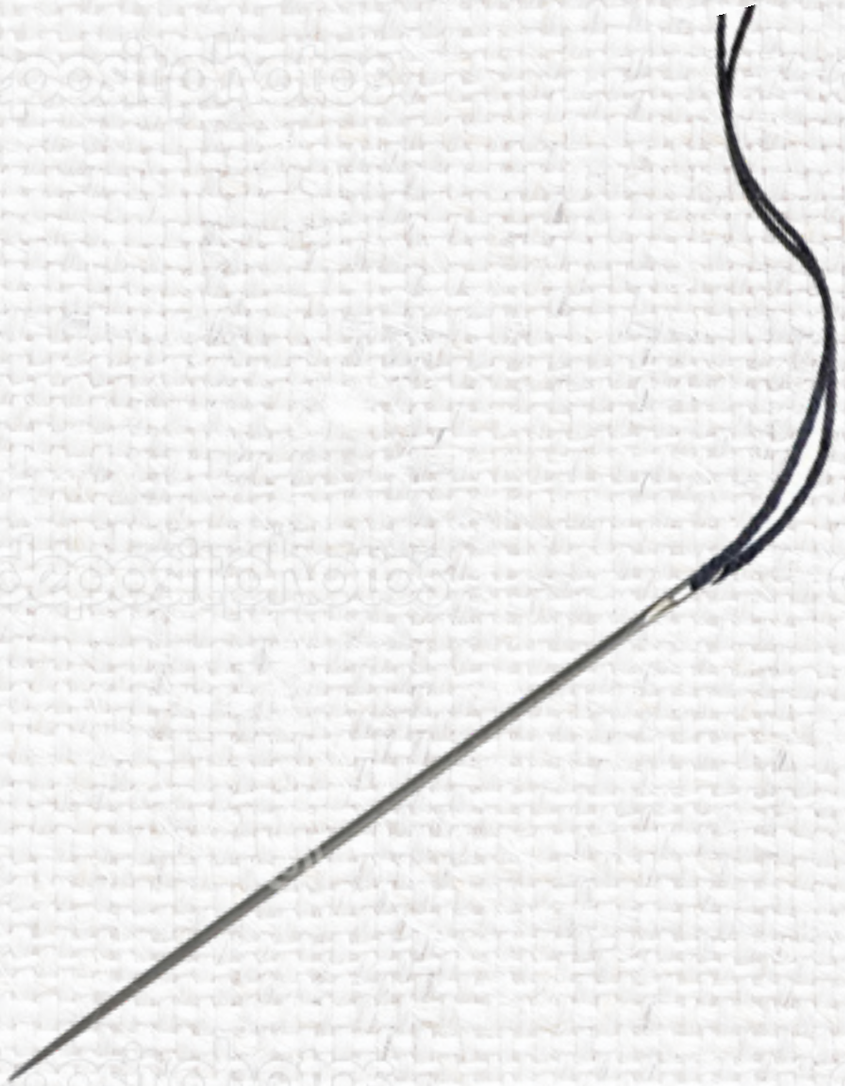
By January we had 20 conforming modules.

In June we had 200 in time for the first release of the multi-threaded framework, and now we have 230.

However this was not the largest obstacle to performance

- HEP software, required patching for concurrent use
- thread unsafe data and conditions objects
- framework policy and coding guideline violations
  1. Do not use non-const globals and non-const statics
  2. Do not 'cast away const' on Event or EventSetup data products
  3. Do use non-static module member data as long as the data object does not contain hidden global state

Tools



Tool Categories

Static code analysis

- Clang

Run time checking

- Helgrind

Static Code Analysis

CMS extended clang static analysis tool

- http://clang-analyzer.llvm.org

Types of Checkers

- Problem with const member functions of data products
- Finding statics that affect modules

Data Products Checking

Data Products are shared between modules

Only const access is allowed

We check for

- Non-const statics
- Mutable member data which is not std::atomic<>
- Member functions casting away const on member data
- Pointer member data being returned from const function
- Pointer member data being passed as non-const argument to function
    1. includes calling a non-const member function of the pointed to class

Checks done recursively on all data members which are classes

Modules & Statics

Any non-const static used by a module is shared state

Used the extended clang tool which

- Finds which functions in system interact with statics
- For each function in system, determine which other functions they call
- For a given module, see if any functions it calls ultimately reach a static

Helgrind

Tool in Valgrind suite

Searches for data races between threads

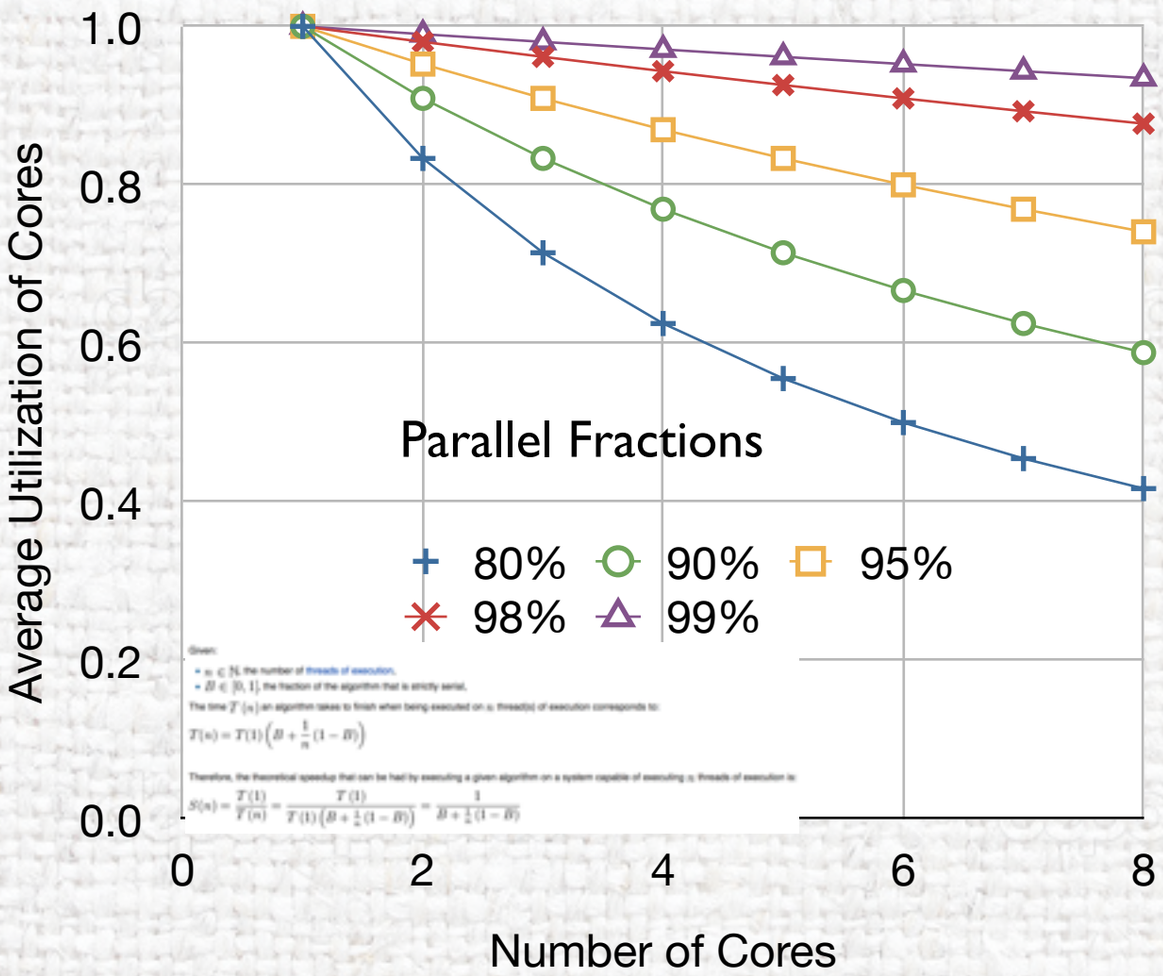
- Records memory reads/writes done by each thread
- Flags if multiple threads use same memory address and one does a write
-

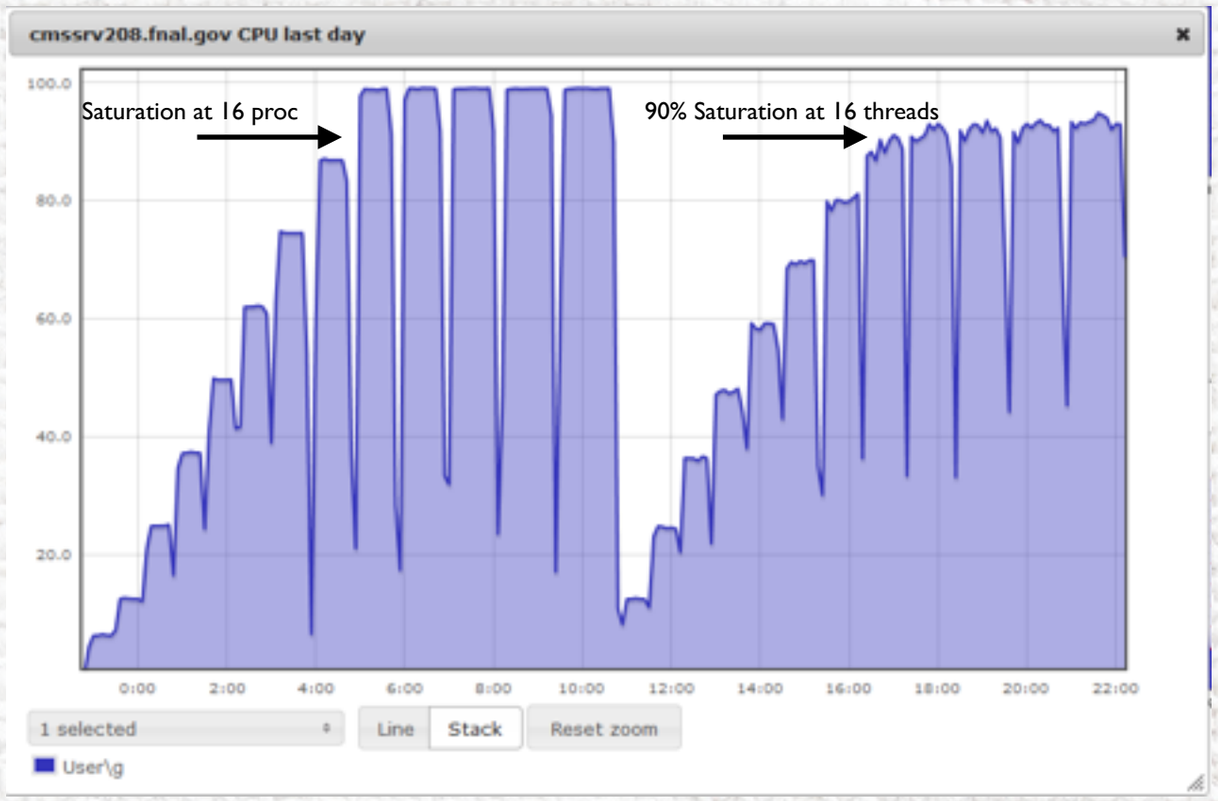
```
Possible data race during write of size 1 at 0x8D878A0 by thread #7
Locks held: none
   at 0x8E7B62C: MessageLogger::establishModule(...) (in libFWCoreMessageService.so)
   ...
   by 0x49CF6E9: EventProcessor::processEvent(unsigned int) (in libFWCoreFramework.so)

This conflicts with a previous write of size 1 by thread #2
Locks held: none
   at 0x8E7B62C: MessageLogger::establishModule(...) (in libFWCoreMessageService.so)
   ...
   by 0x49CF6E9: EventProcessor::processEvent(unsigned int) (in libFWCoreFramework.so)

Address 0x8D878A0 is 144 bytes inside a block of size 152 alloc'd
   at 0x4807A85: operator new(unsigned long) (in vgpreload_helgrind-amd64-linux.so)
   ...
```

Performance

The Amdahl Problem

To keep 8 cores 95% busy need 99.2% of our code to run in parallel
Even quick running modules will bottleneck threading
a heuristic that worked -> fix the module that is waiting to execute

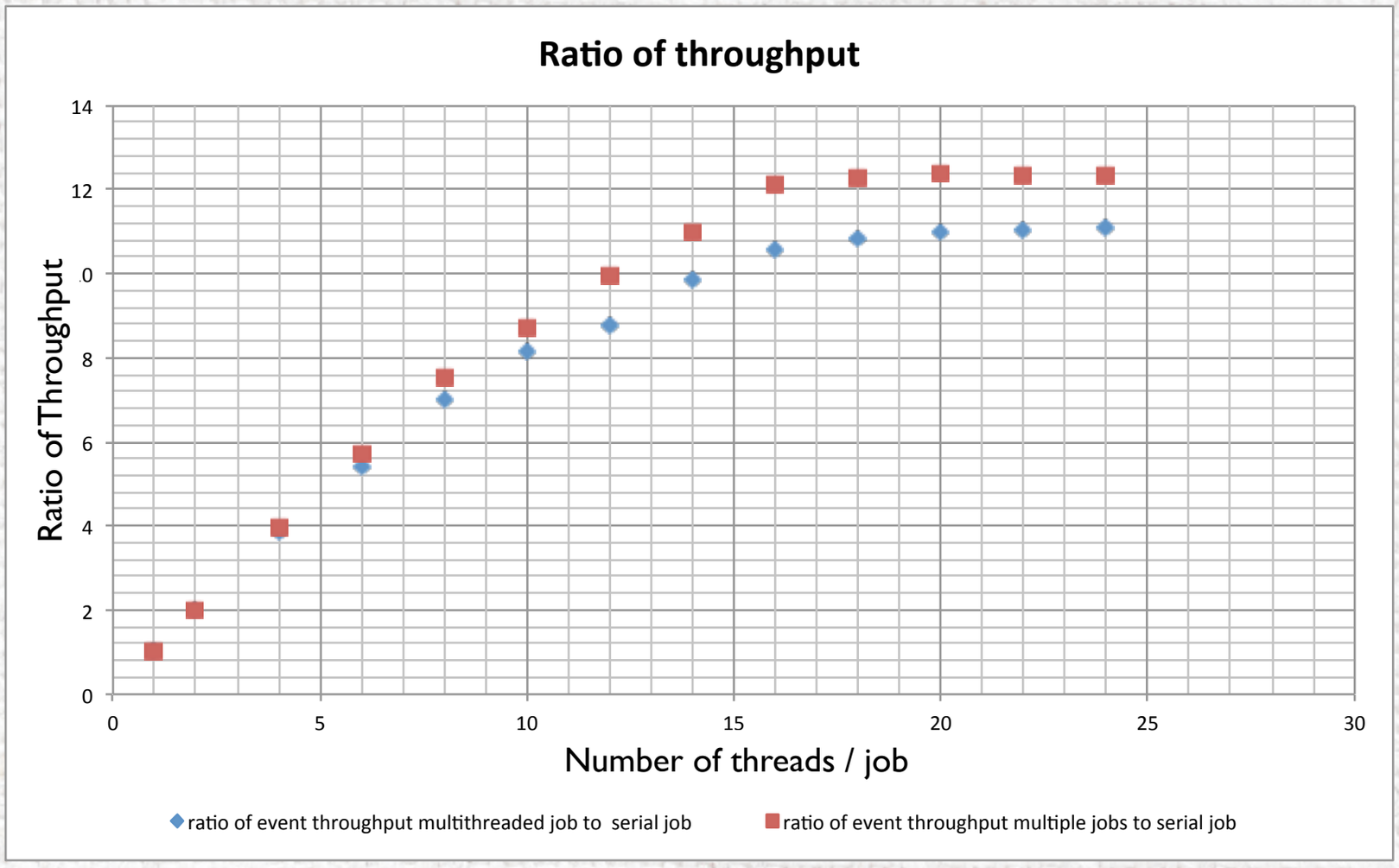




Given the 90% saturation at 16 cores we calculate that 99.3% of the reconstruction code is run in parallel.

Throughput Performance

Our benchmark for performance is reconstruction of a t-tbar MC sample with 25ns bunch spacing with average 40 interactions per crossing… most difficult problem for Run 2

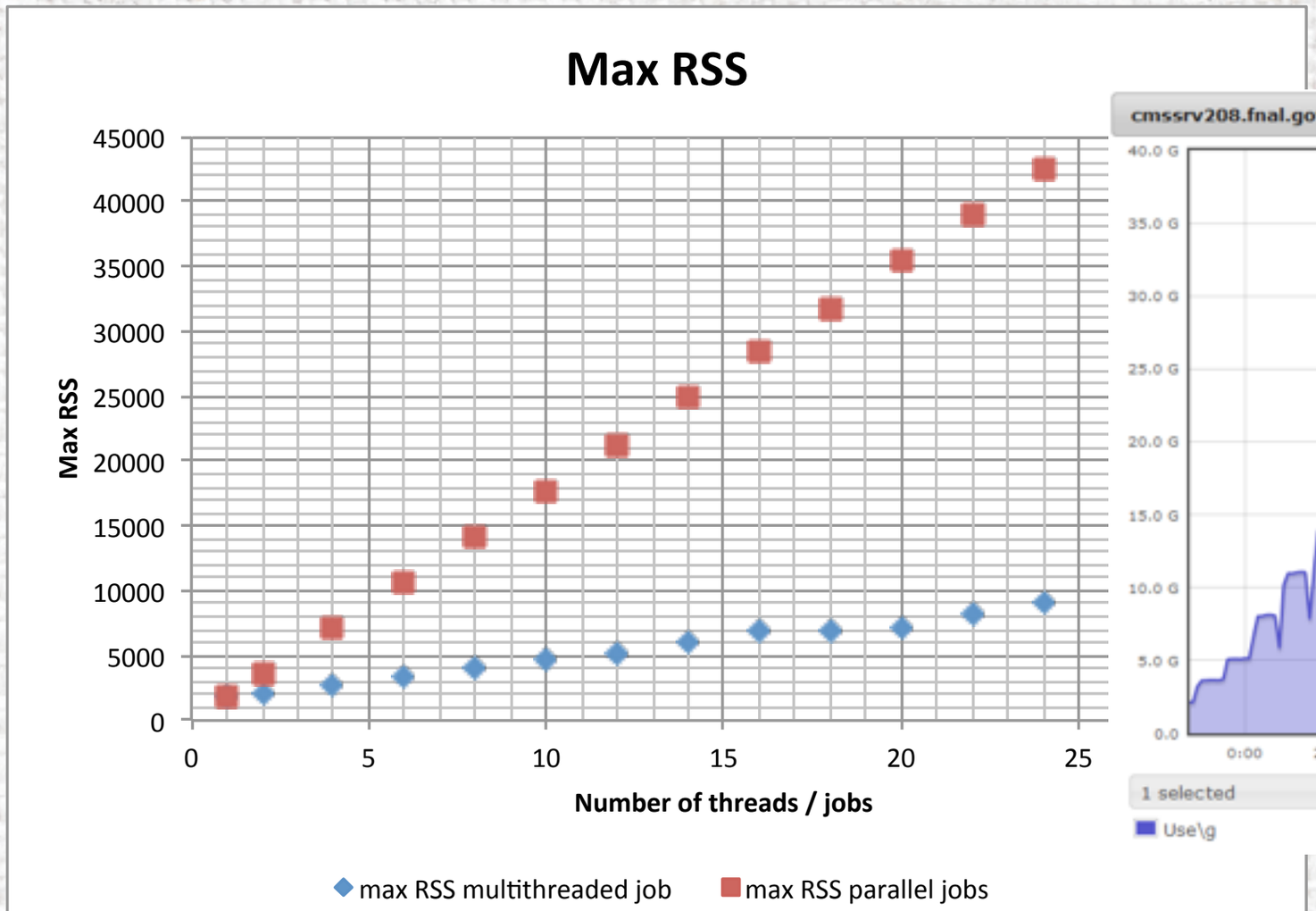


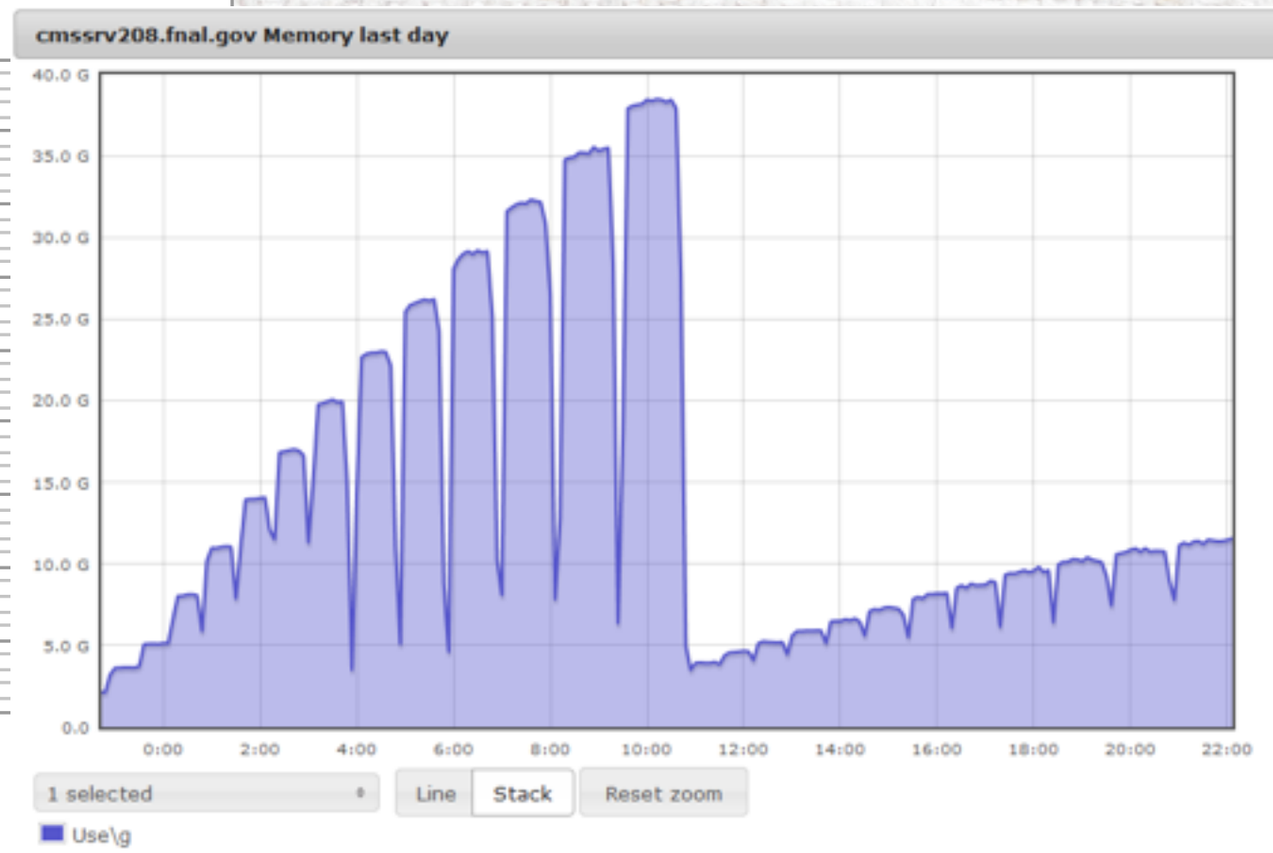
The above compares running n simultaneous processes to n simultaneous streams.

Good scaling up to 8 on this 16 core, 64 GByte AMD machine.

Memory Performance

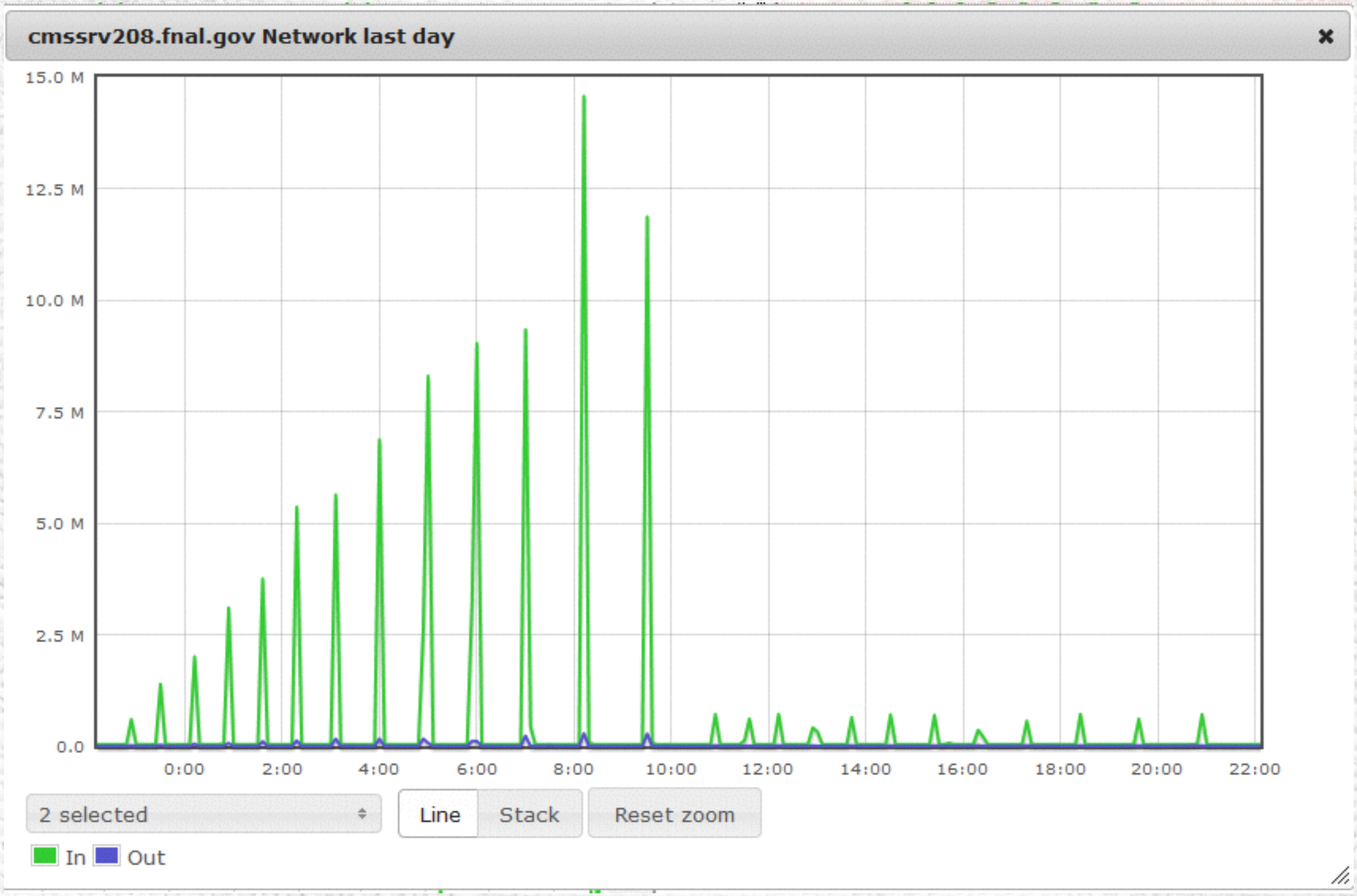
The big win is in memory consumption and network load.





Network Load

The big win is in memory consumption and network load.



Conclusion

CMS has moved to a multi-threaded framework

The design allows many different levels of concurrency

- Events, modules and sub-module

Thread-unsafe code is allowed via 'One' module variety

- Framework guarantees serialization

Tools to find thread-safety issues have been developed

First performance results show that 99.3% of our reconstruction application can run in parallel

- memory consumption is no longer a problem
- network load is way down