Status of CMS' Threaded Framework

Christopher Jones *FNAL*

Overview

Status

Performance Measurements

Future Work

Conclusion

Status

Have had a working integration build (IB) since beginning of July

All threading changes were merged into main CMS release at that time

All threading problems seen in IB have been fixed

Most were problems in ROOT which are now patched

All IB validation workflows (sim, reco, etc.) are working with multiple threads

All conditions related products are now thread-safe

They can be accessed by multiple events simultaneously

Converted "easy-case" RECO modules to stream modules in July

Additional thread safety changes will allow more modules to be converted

Have been using threads in Tier0 RECO replay

Using the October release

Have begun doing scale tests on the Grid

Looking at job CPU efficiency

Performance

Performance is limited by code which must run sequentially

Causes of sequential code in CMS

Legacy modules

Modules which have not been modified to be thread friendly
Only one legacy module can run at a time

'One' modules

An instance of a 'One' module can only process 1 event at a time

Run and Lumi transitions

Must finish processing all events in a Lumi before going to next Lumi

To keep 8 cores 95% busy need 99.2% of code to run in parallel

Workflow Performance

Concentrated on getting good efficiency from RECO

Tracking group has worked to make all their modules thread efficient

DQM has been working to convert all their modules to be thread friendly

Work ongoing to be able to use parallel Geant4

Initial implementation is working

RECO Measurements

Machine

2 CPUs each with 8 Cores
AMD Opteron 6320 Core
64GB RAM

Job Configuration

RECO sequence
TTBar Monte Carlo
25ns bunch spacing
Average of 40 interactions per crossing

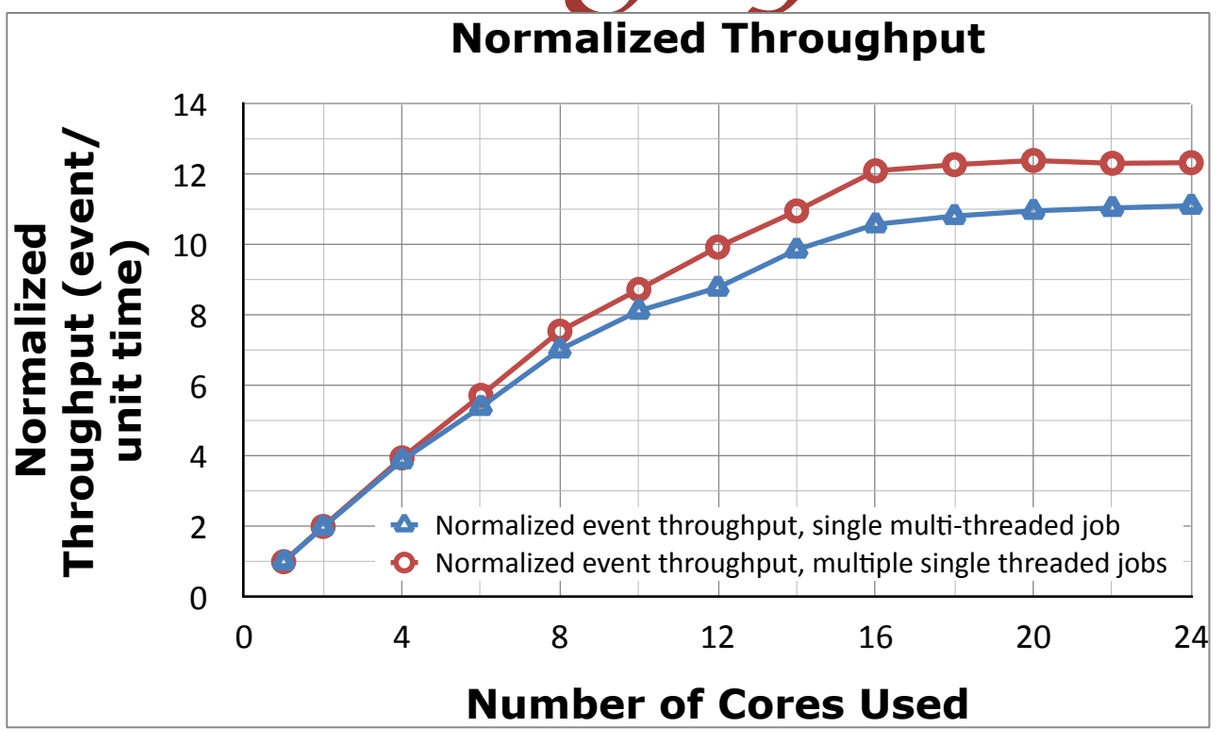
Measurement Procedure

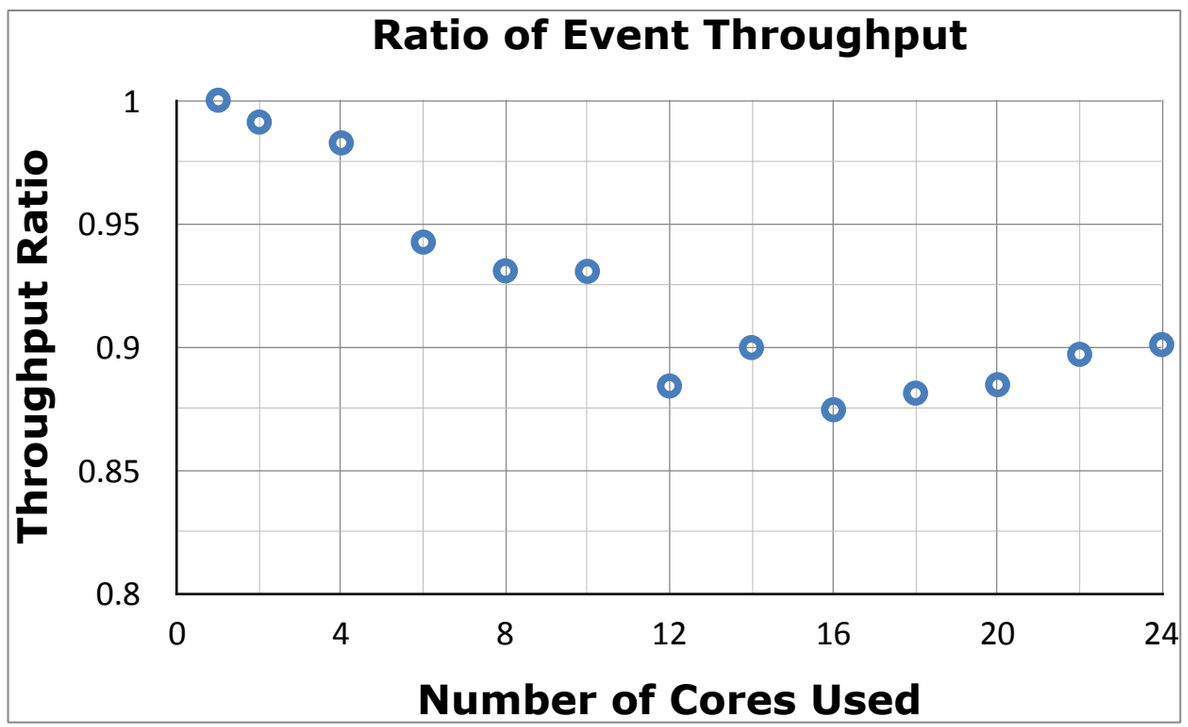
Run N single threaded jobs
Run 1 multi-threaded job using N threads

Results were shown at ACAT

Liz Sexton-Kennedy, Patrick Gartung and myself contributed to measurements

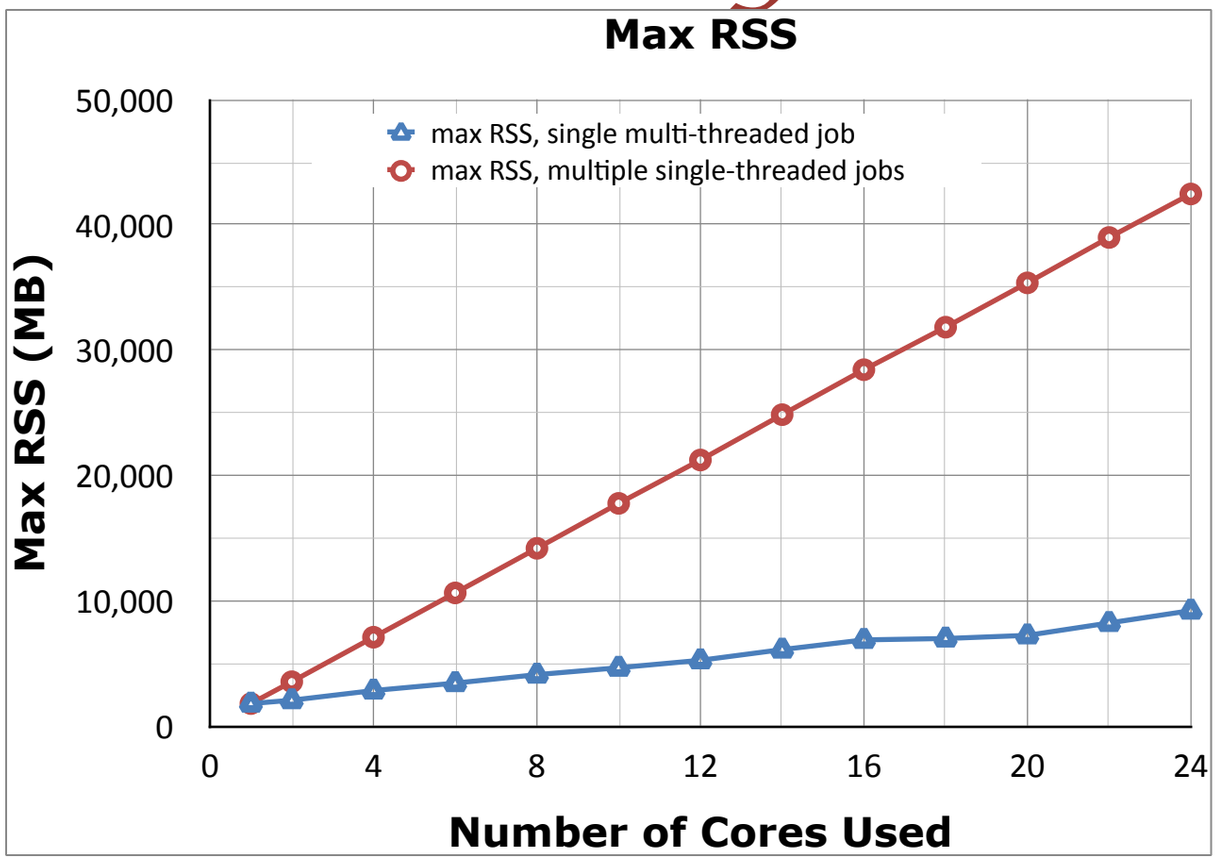
Throughput Measurement

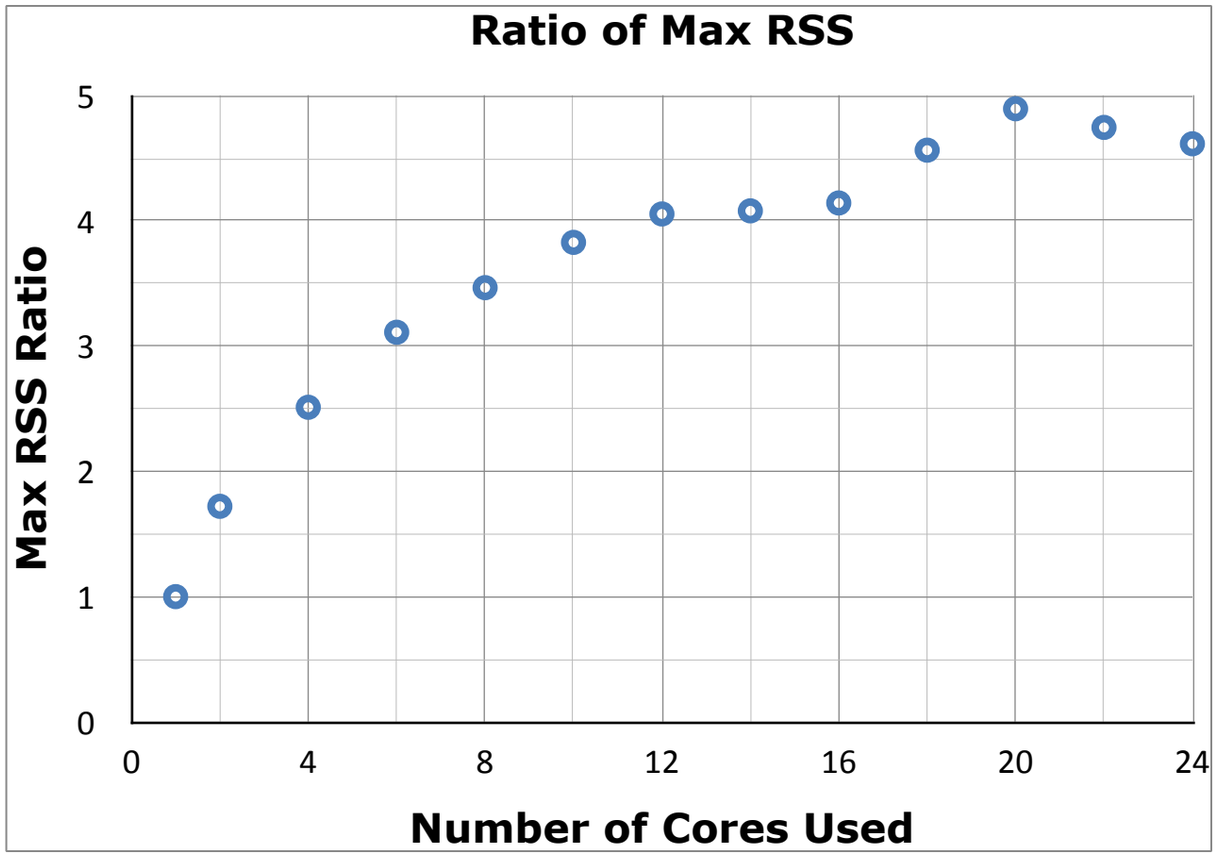




For 8 threads we see a 93% efficiency for threaded compared to single threaded

Memory Measurement



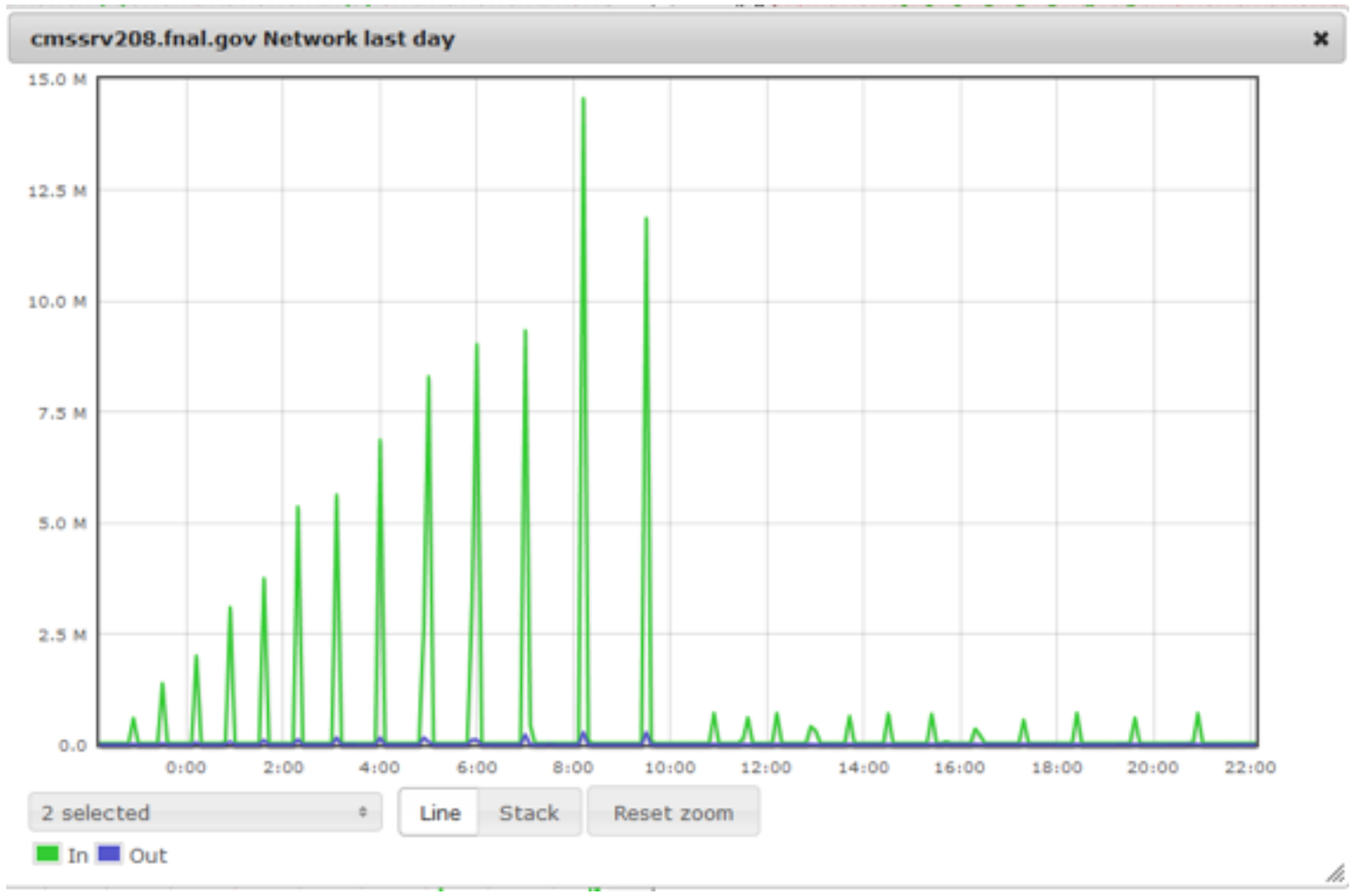


At 8 cores, single threaded is 3.5x memory than multi-threaded

8 single threaded jobs: 14.2GB

1 multi-threaded job: 4.1GB

Network Measurement



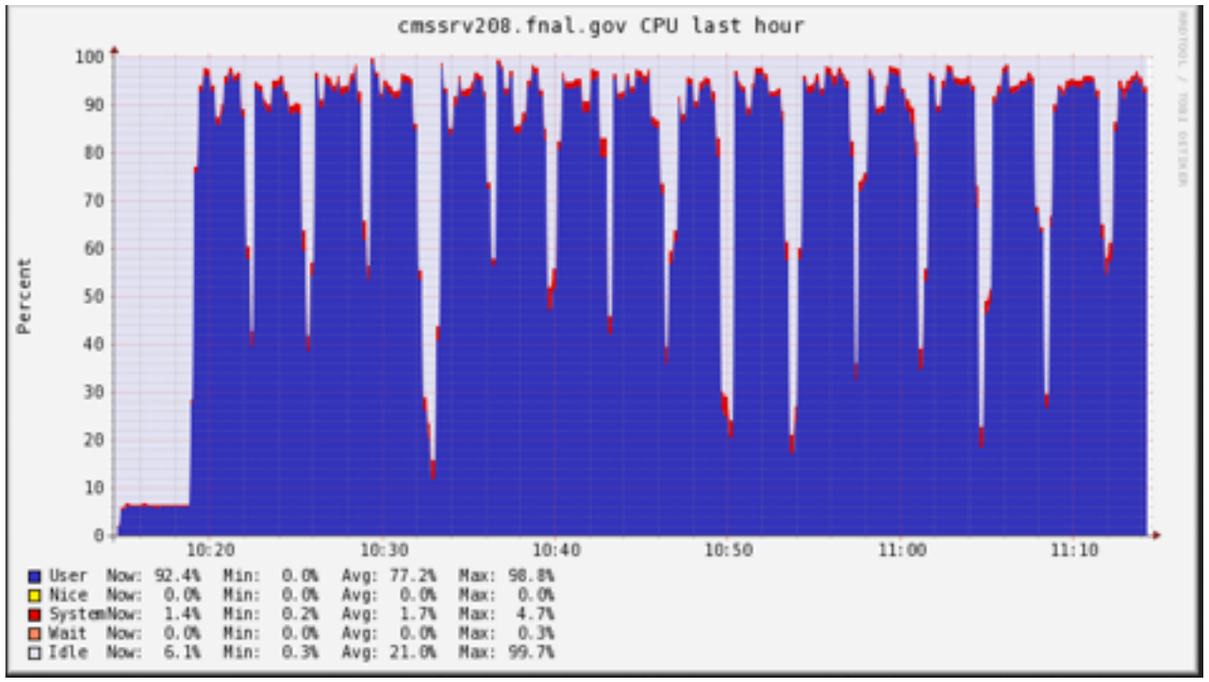
Captured Network Usage on the Machine

1st half are the single-threaded jobs going from 1to 24 simultaneous jobs
2nd half are the multi-threaded jobs going from 1 to 24 threads

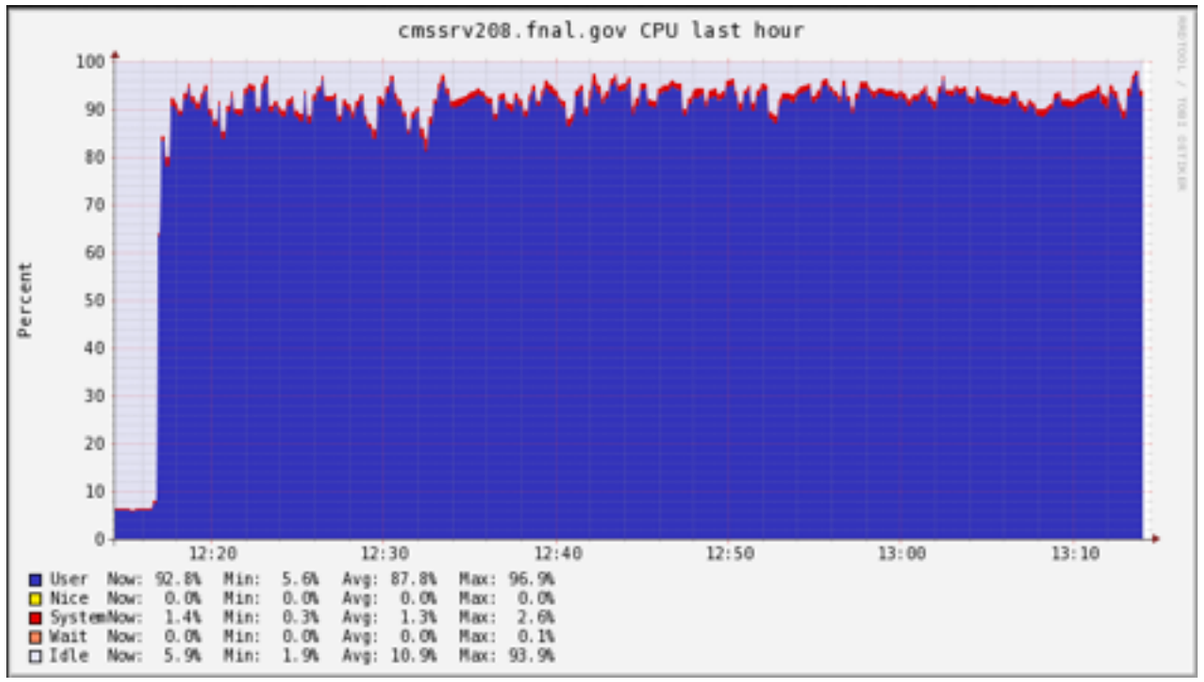
What is seen is related to conditions

Small LuminosityBlock

100 Events per LuminosityBlock



All Events in 1 LuminosityBlock



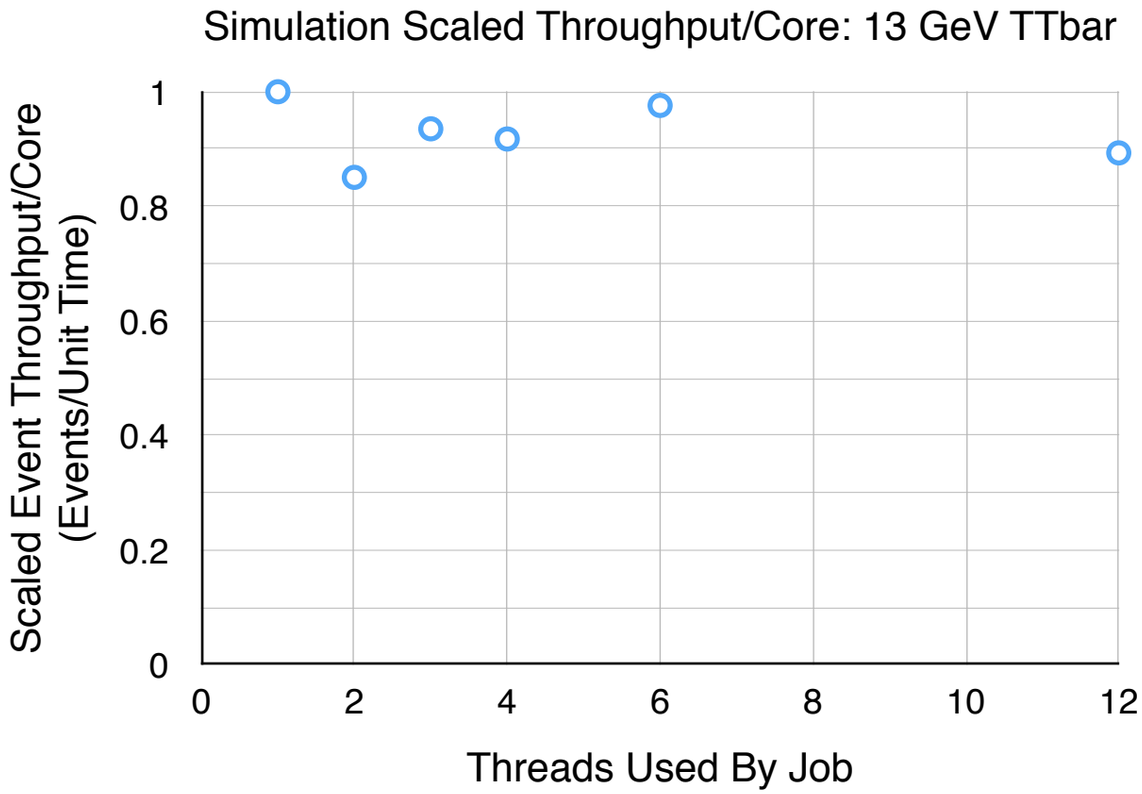
Monitored CPU utilization

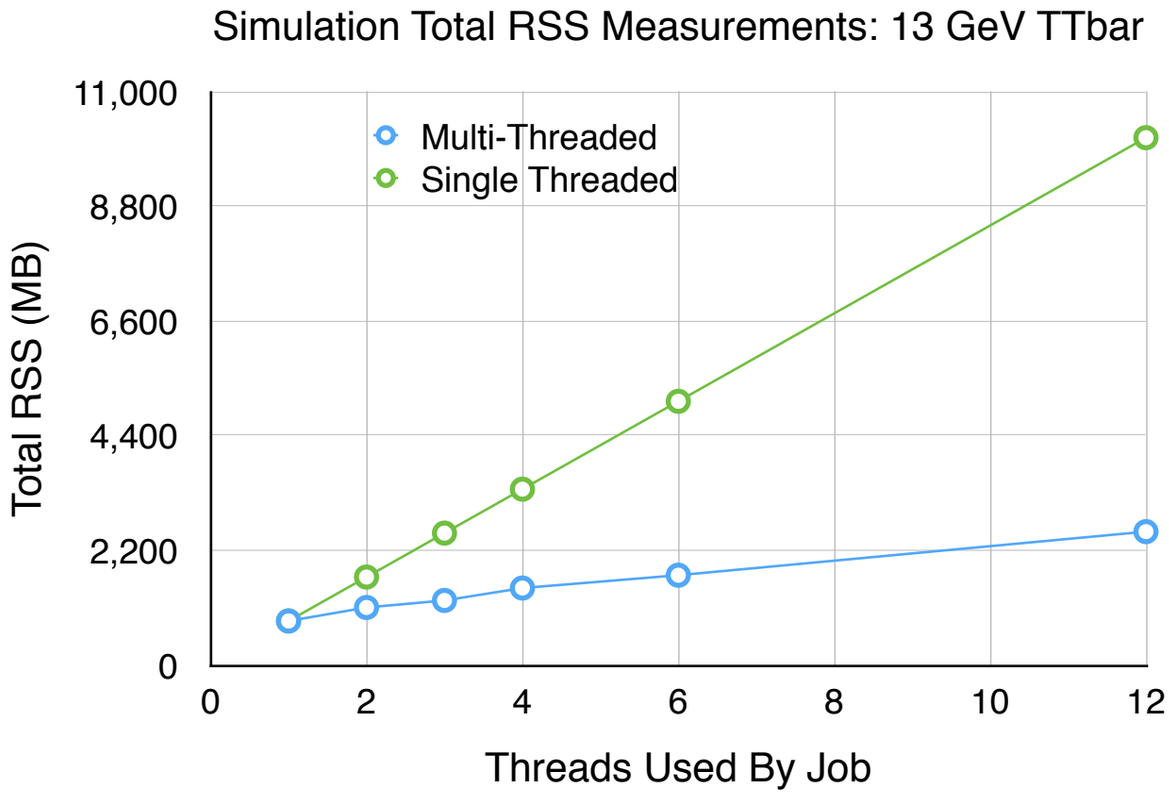
Can see end LuminosityBlock synchronization affects efficiency

Fewer events in a LuminosityBlock means more serialization

Previous measurements done with 1 LuminosityBlock

GEN-SIM Measurements





Brand new results from Simulations group

Used 12 core machine where N jobs * N threads == 12 for all measurements

Throughput per core very flat

Shows good scaling to 12 cores

Memory consumption much less than single threaded jobs

1 jobs with 12 threads < 2.6GB

12 jobs each with 1 thread > 10 GB

~200MB memory increase per thread

Future Work

Use tasks instead of mutex to control non-stream modules

- Now a non-stream module can block the use of a thread
  - Switching to tasks would allow other work to happen on the thread
- Still would only have 1 module run per Event
  - Event data products not thread safe yet
- Would allow modules to internally use their own tasks
  - i.e. sub-module level parallelism
- Requires 'consumes' migration to be finished
  - modules must now register what data they will 'consume'
  - 'hard' 1% are left
- Needed for further threading changes

Run multiple modules per event

- Requires switching to task base modules (see previous bullet)
- Framework could schedule around non-stream based modules
  - Increased CPU efficiency
- Mitigates some of the LuminosityBlock synchronization efficiency problem
  - Last events in LuminosityBlock get to use the 'freed' threads which ran finished events
- Can be used to decrease memory used by a job but keep CPU efficiency
  - Amount of memory used in a job is dependent on # of events processed simultaneously
  - Use # threads > # events takes less memory

Future Work Continued

Run multiple simultaneous LuminosityBlocks/Runs

API of Framework is built to accommodate this

Requires no module in job needing to see 1 Lumi/Run at a time

- no legacy in job
- no 'one' module in job where module says it needs to see Lumis or Runs

Conclusion

High pileup SIM & RECO jobs have good CPU efficiency for 8 Cores

Present efficiency is good enough for CMS' Run 2 needs

Future work will allow even higher CPU efficiency