Multi-threaded Scheduling in CMS

Dr Christopher Jones
HSF Framework Group

29 April 2020

CMS's Threading Goals

Better scaling of system resources as core count increases

- Number of workflow jobs does not have to increase as core count increases
- Potential to use sites with lower available resources

More sharing between cores

- Share infrequently updated memory
  - conditions
  - I/O buffers
- Share file handles
- Share network connections

Faster processing of individual Events is NOT a goal

- CMS cares about total events/second for an entire workflow, not so much 1 job
  - A workflow processes millions of events over 10s of 1000s of jobs

Threaded Design

Run multiple transitions concurrently

- Event transitions are the most important
- Number of allowed concurrent transitions of each type set at configuration time

Within one transition run multiple modules concurrently

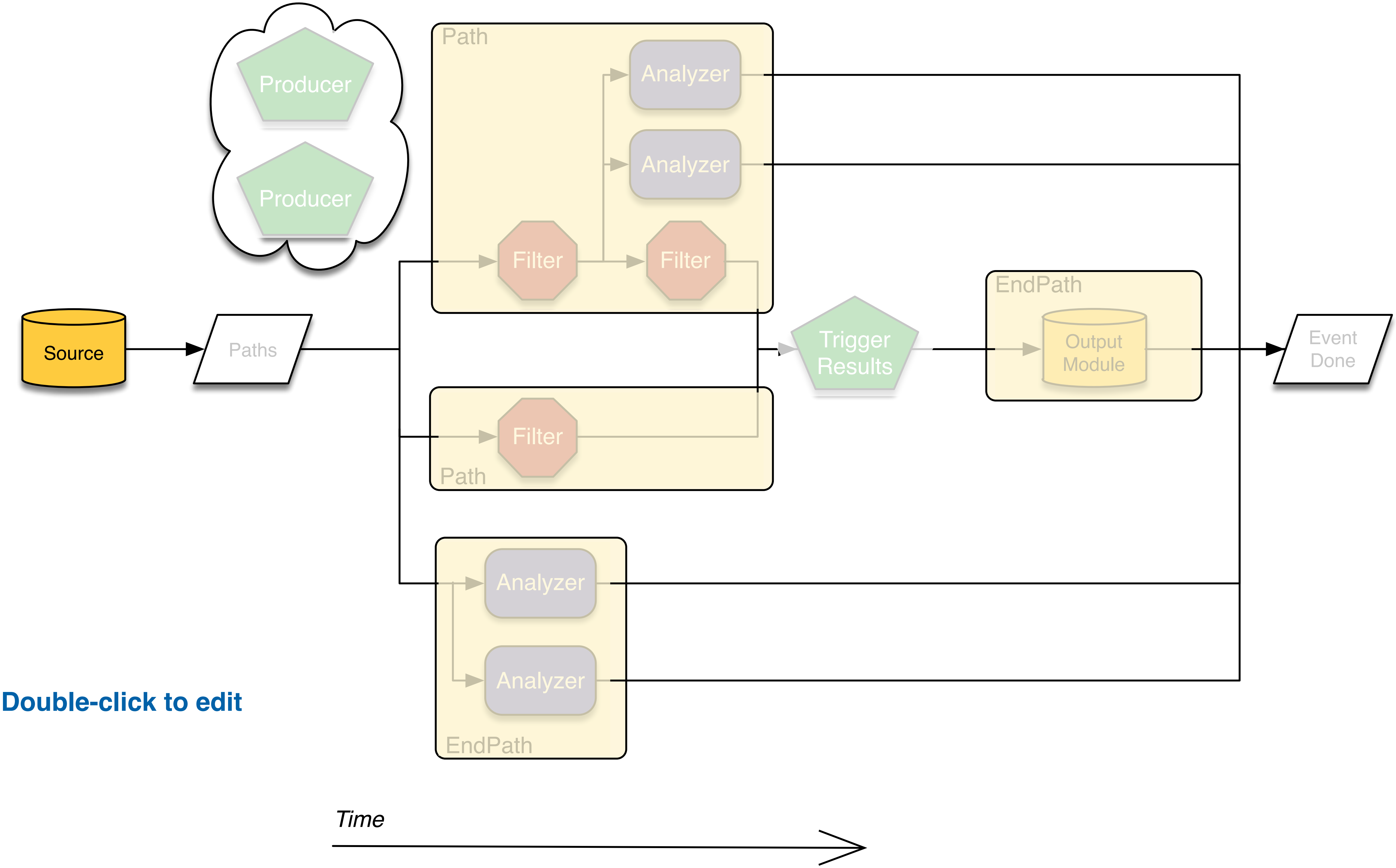
- Have to take into account module dependencies

Within one module be able to run multiple tasks concurrently

Intel’s Thread Building Blocks library used for all of the above

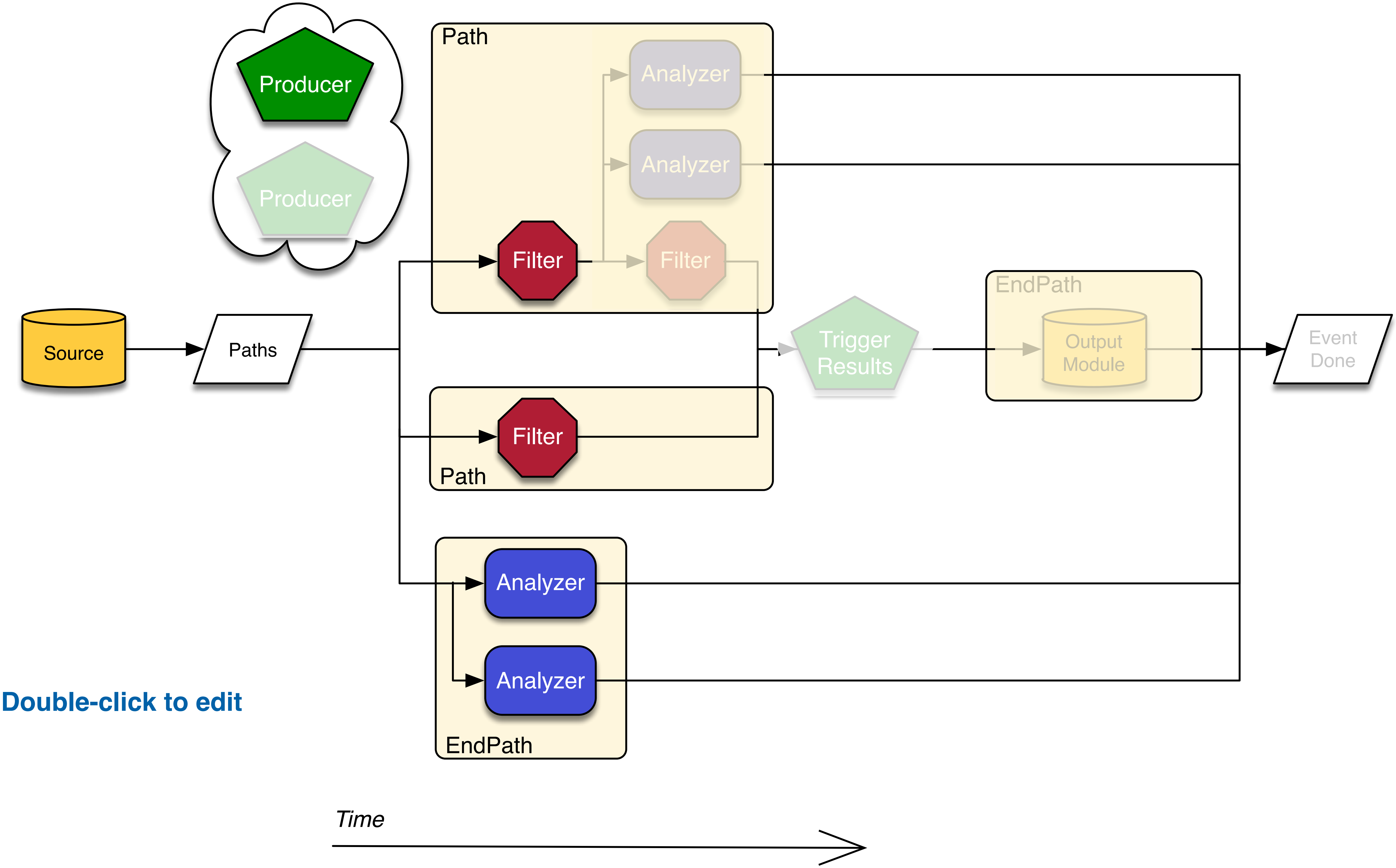
- Break down work into *tasks* and TBB can run them in parallel

Processing an Event



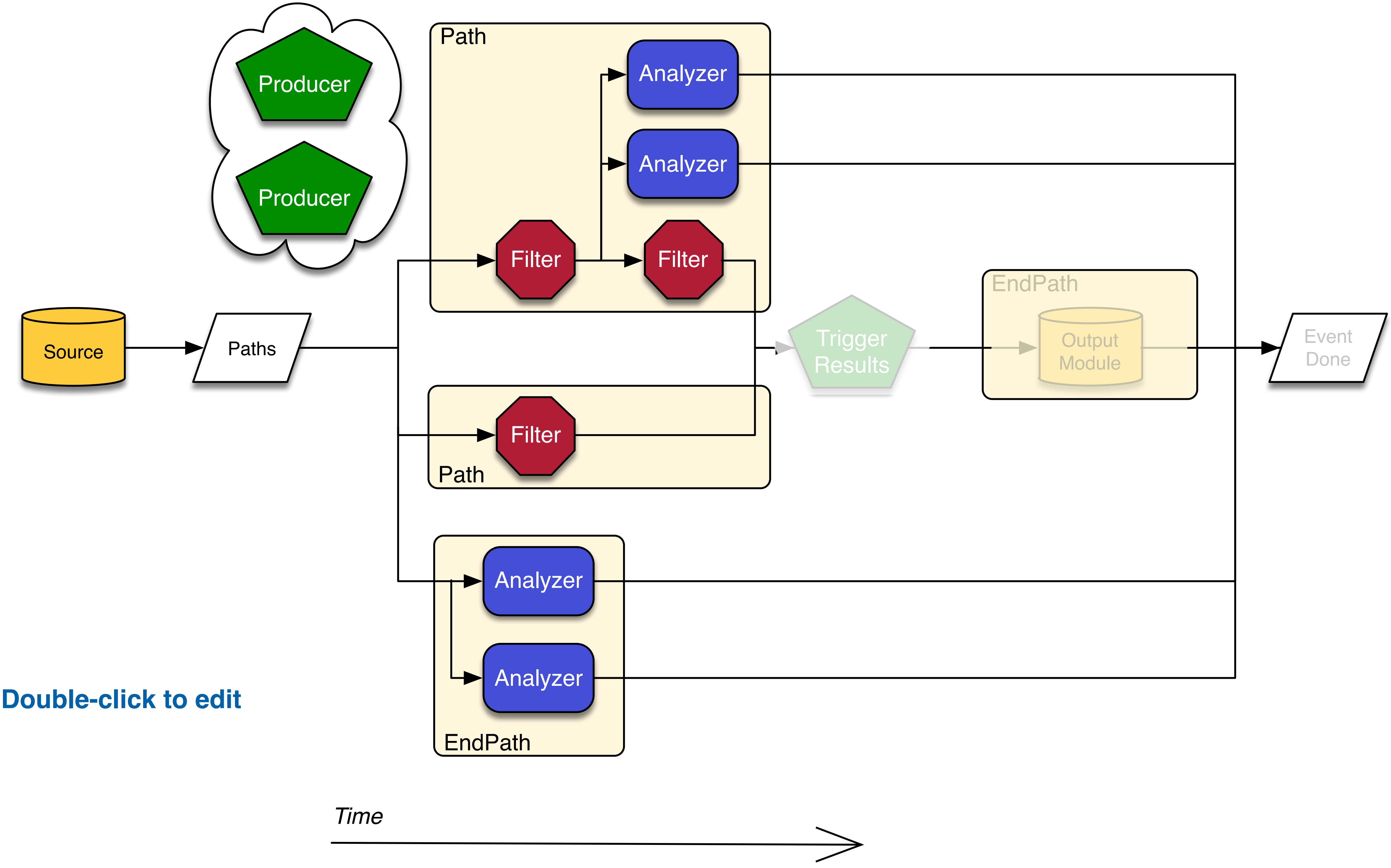
Double-click to edit

Processing an Event



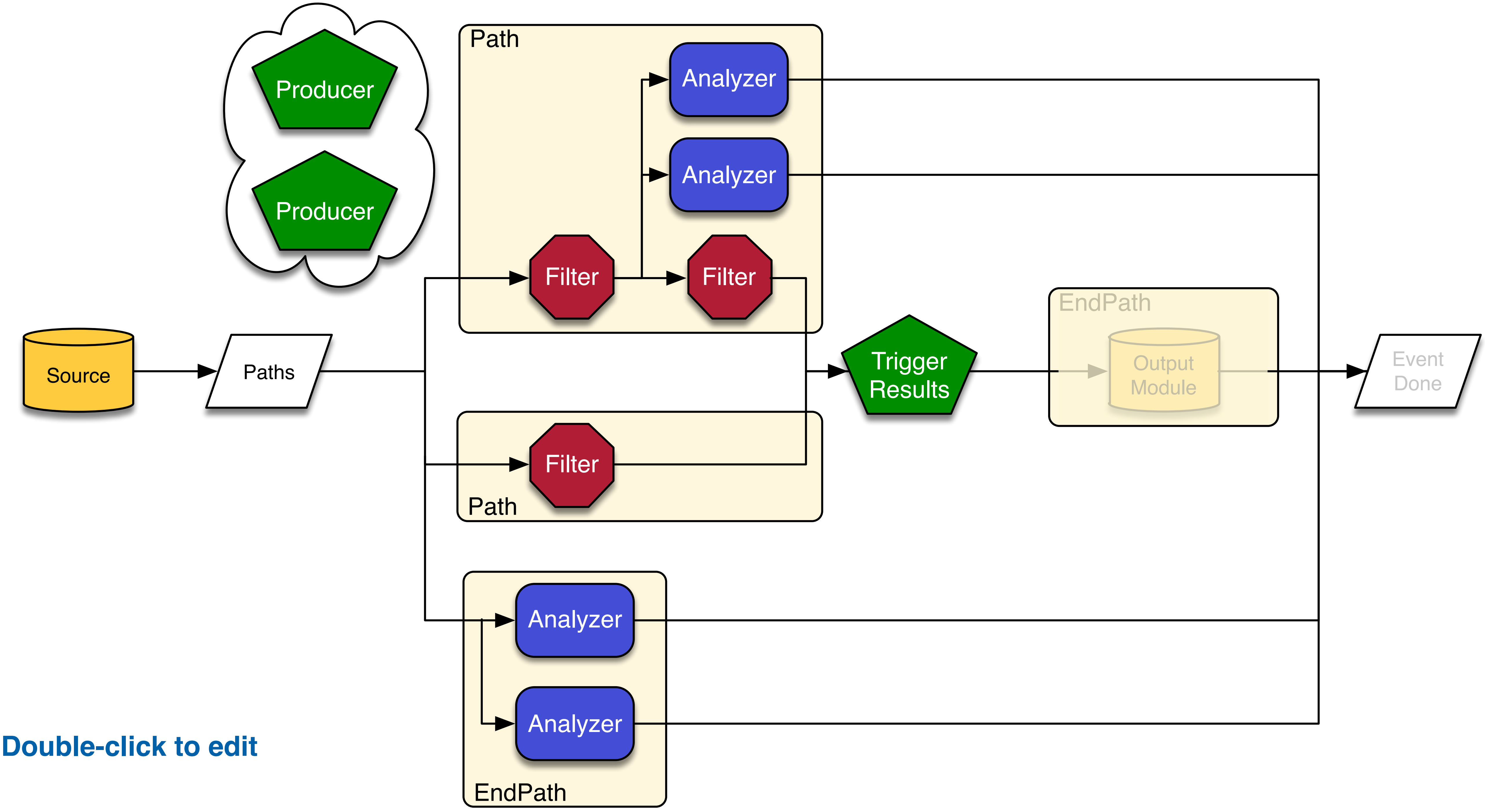
Double-click to edit

Processing an Event

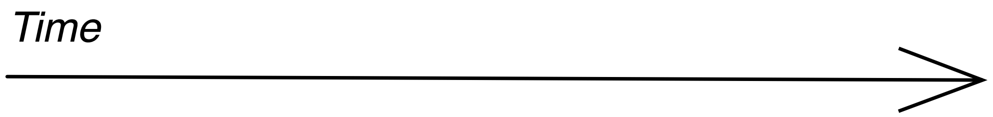


Double-click to edit

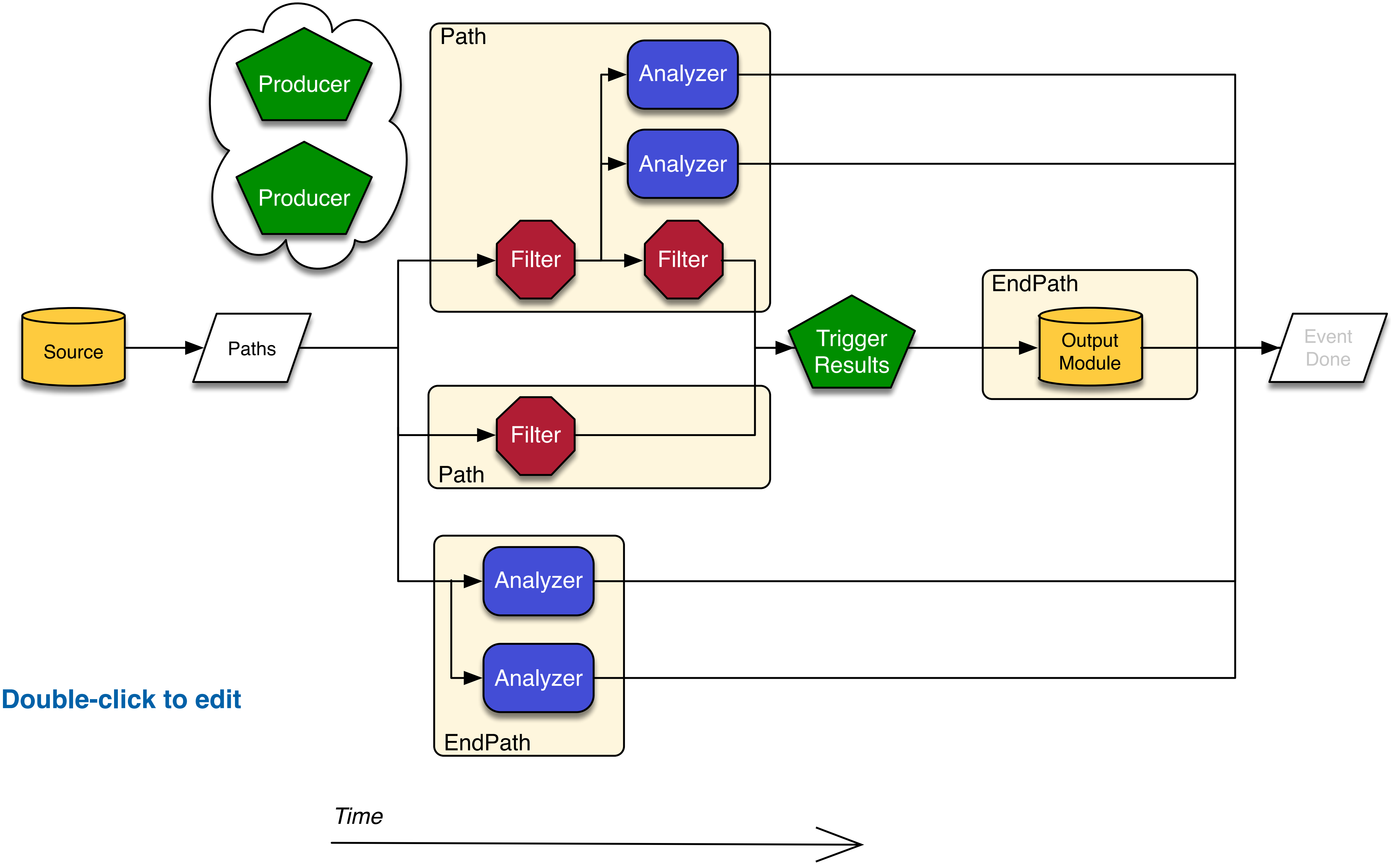
Processing an Event



Double-click to edit

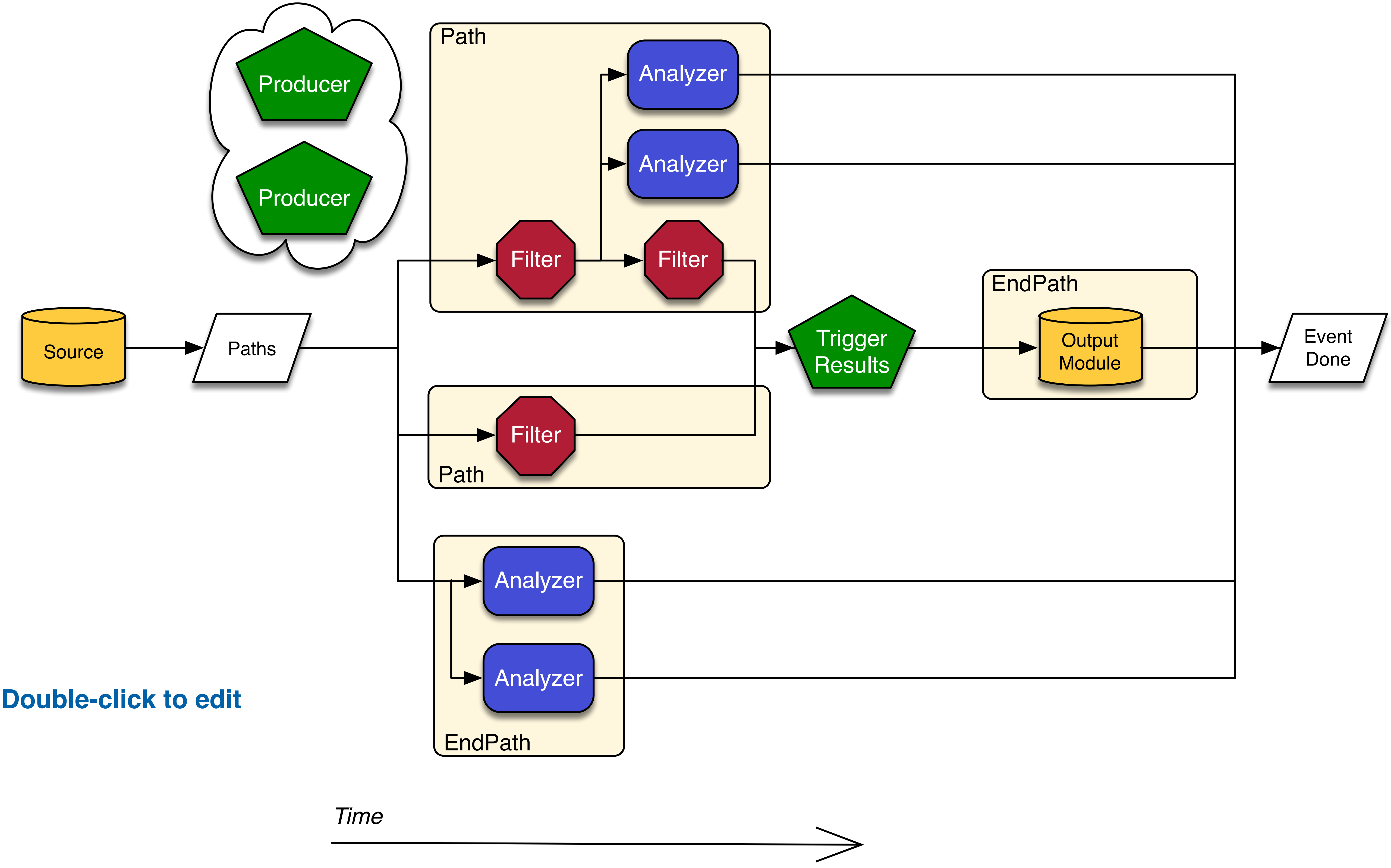


Processing an Event



Double-click to edit

Processing an Event



Double-click to edit

Glossary

Paths

- Events are filtered by Paths
- Paths hold a list of Filters

Filters

- Modules after a Filter on a Path only run if the Filter passes

Producers

- Make data products used by other Modules
- Run first time their data is requested

EndPaths

- Hold modules that want to see all Events or want to see results of Paths

Modules

- Filters, Producers, Analyzers, and OutputModules are all Modules
- Modules can run concurrently

Scheduling

No process wide scheduling is done

- All decisions are done on each thread individually

Based on four items

- TBB's task scheduling
- Prefetching
- Module threading types
- Serial task queues

TBB Task Model

Pre-declare how many threads should be used

For each thread, there is a work queue

task::spawn adds a task to the queue for the thread that called spawn

tasks are pulled from the work queue in Last In First Out order

task::enqueue puts tasks on a shared queue

If a queue is empty, the thread will

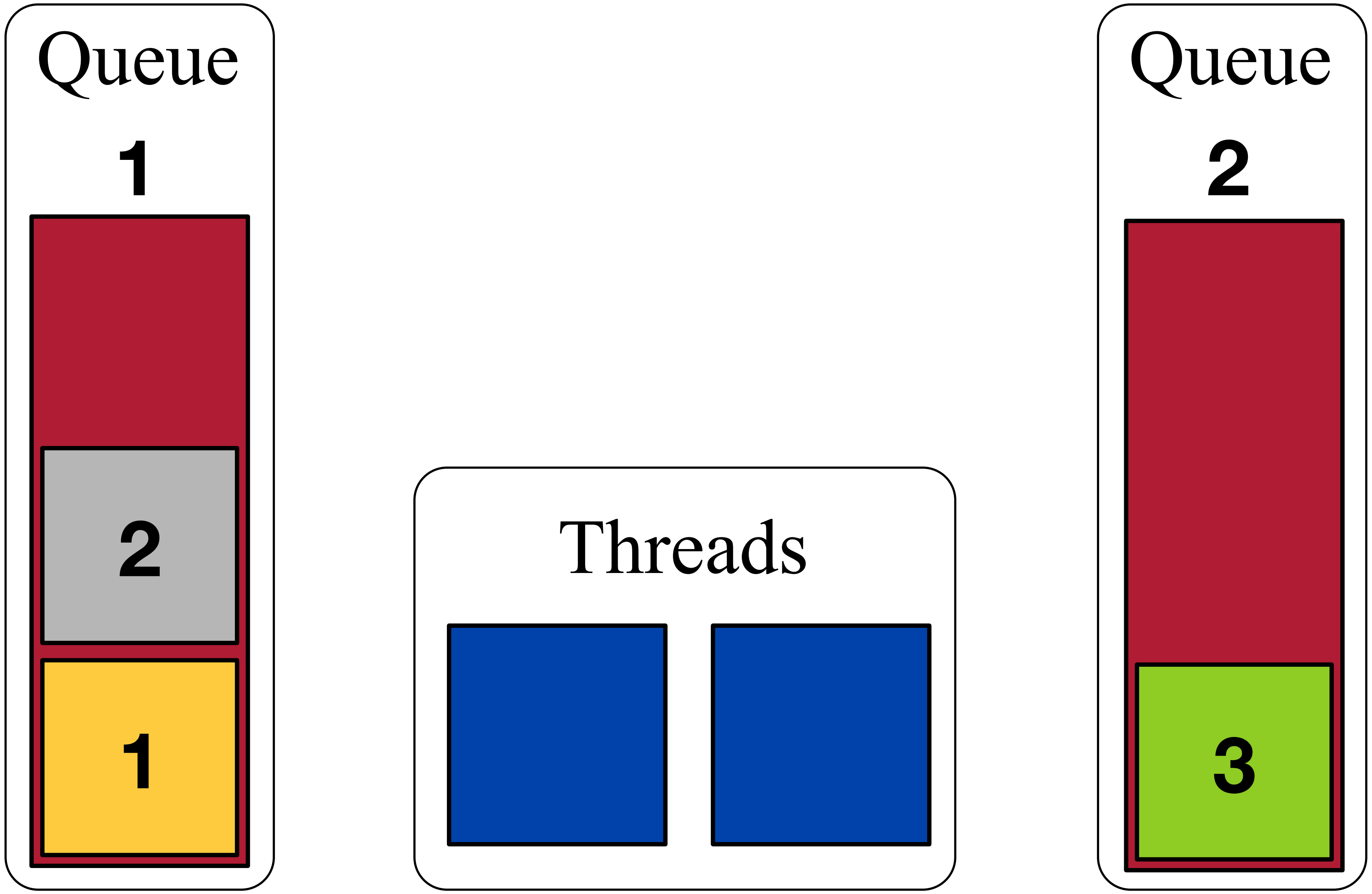
- See if a task is on the shared queue and if so take the oldest one, else
- Steal oldest task from another thread's queue

A task can explicitly return a new task that is to be run next

- Guaranteed to run on the same thread

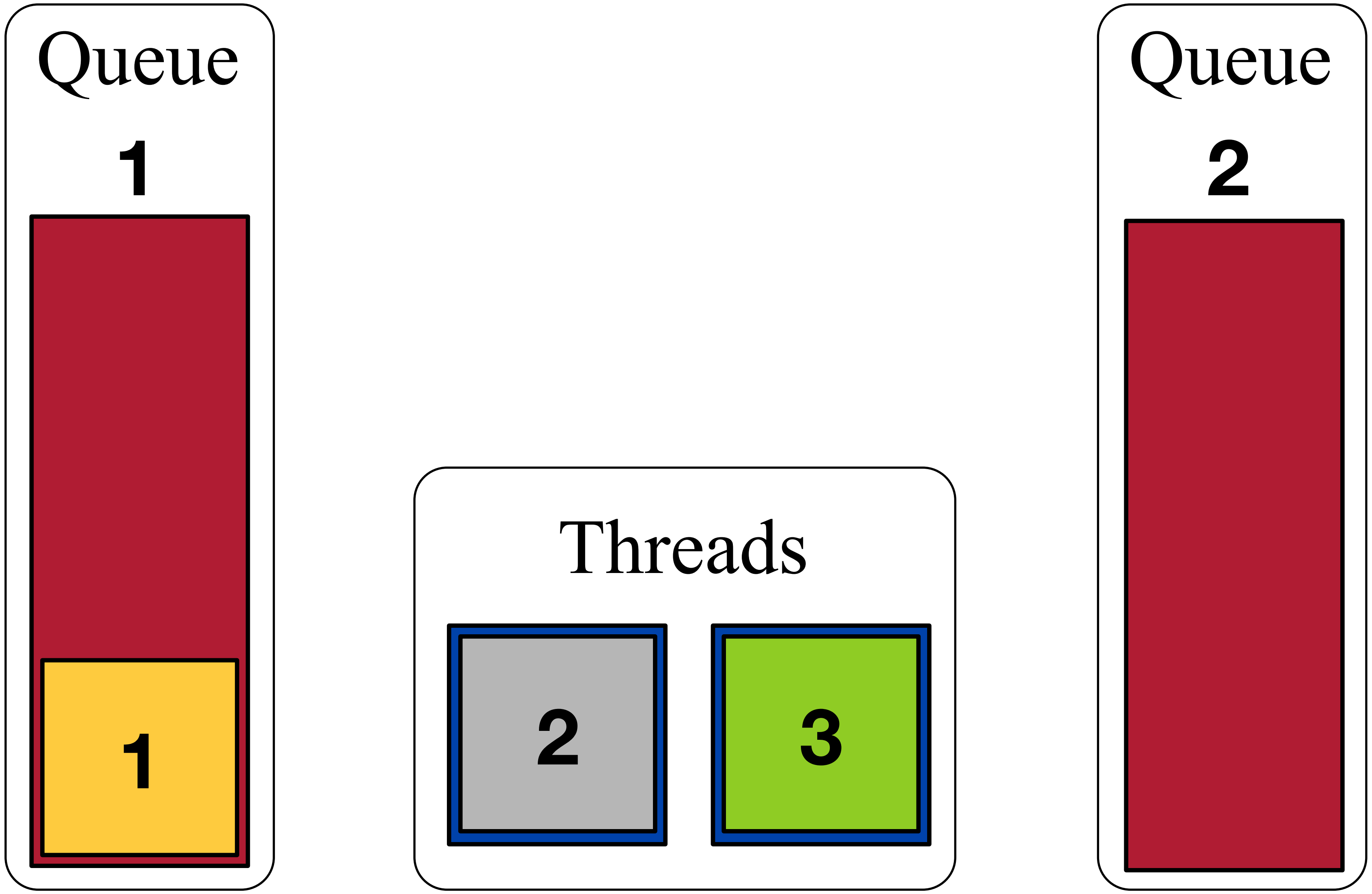
TBB Task Model

Tasks are pulled in Last In First Out order



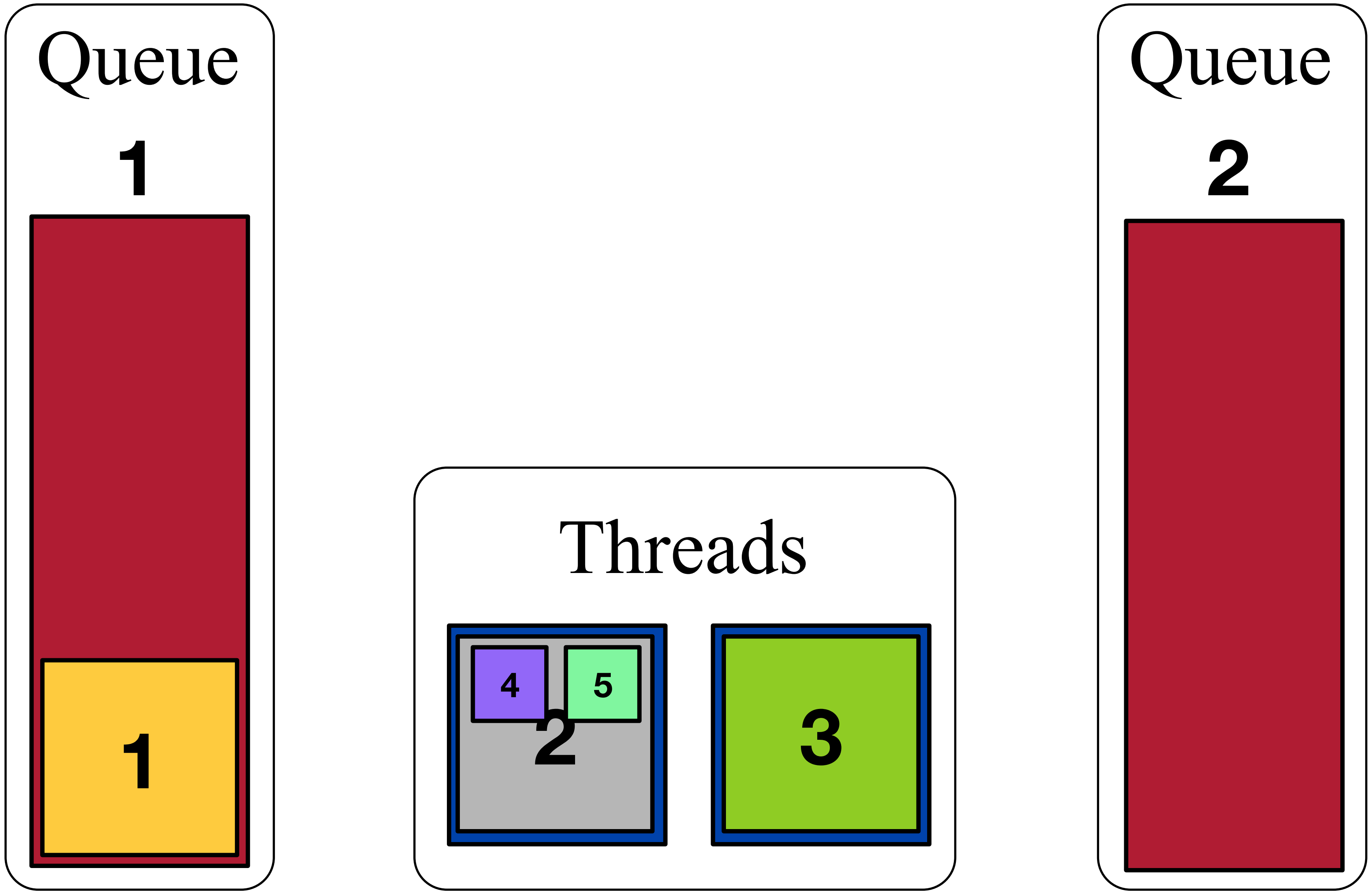
TBB Task Model

Tasks are pulled in Last In First Out order



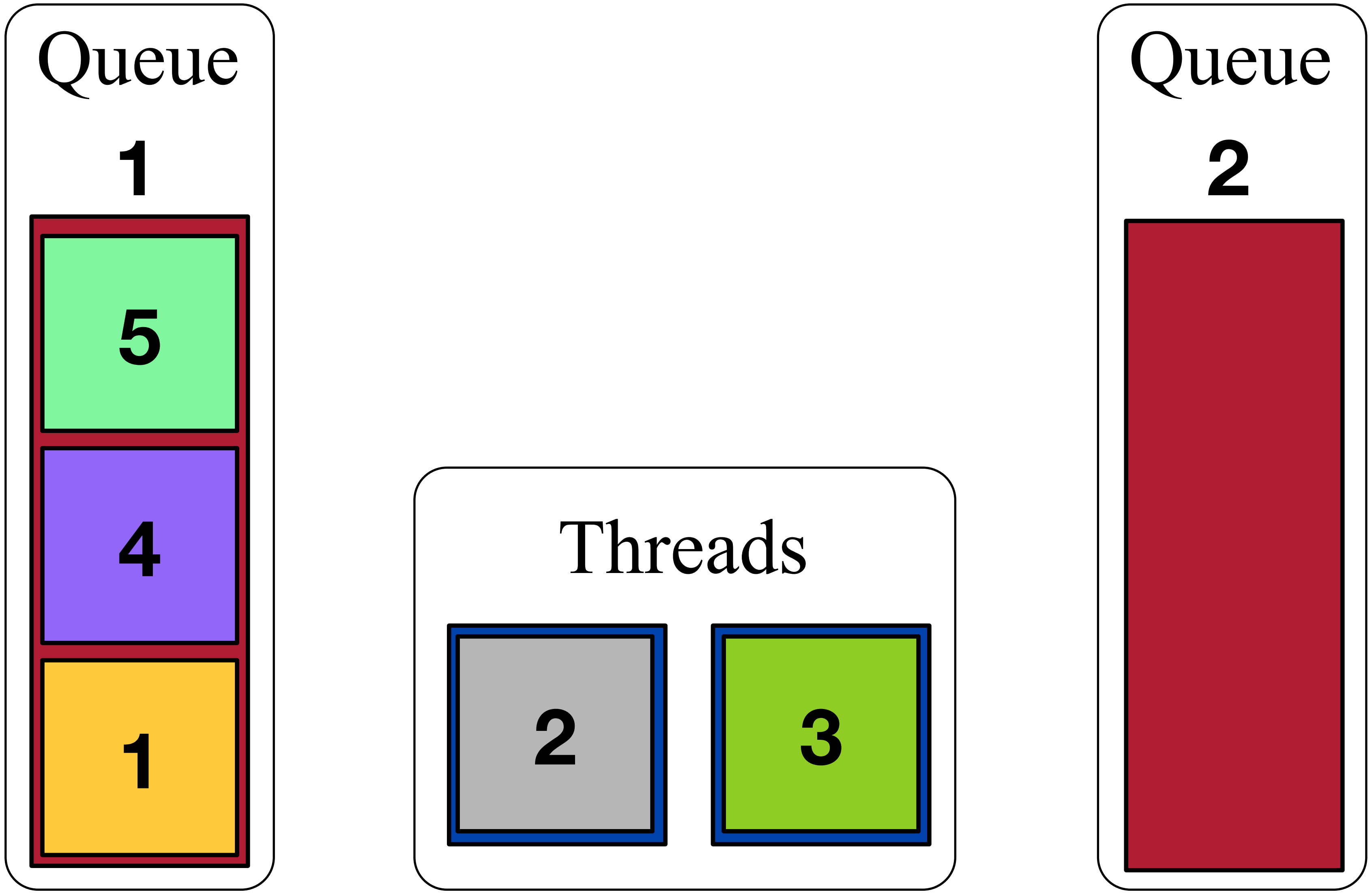
TBB Task Model

Spawned tasks go into same queue as creating task



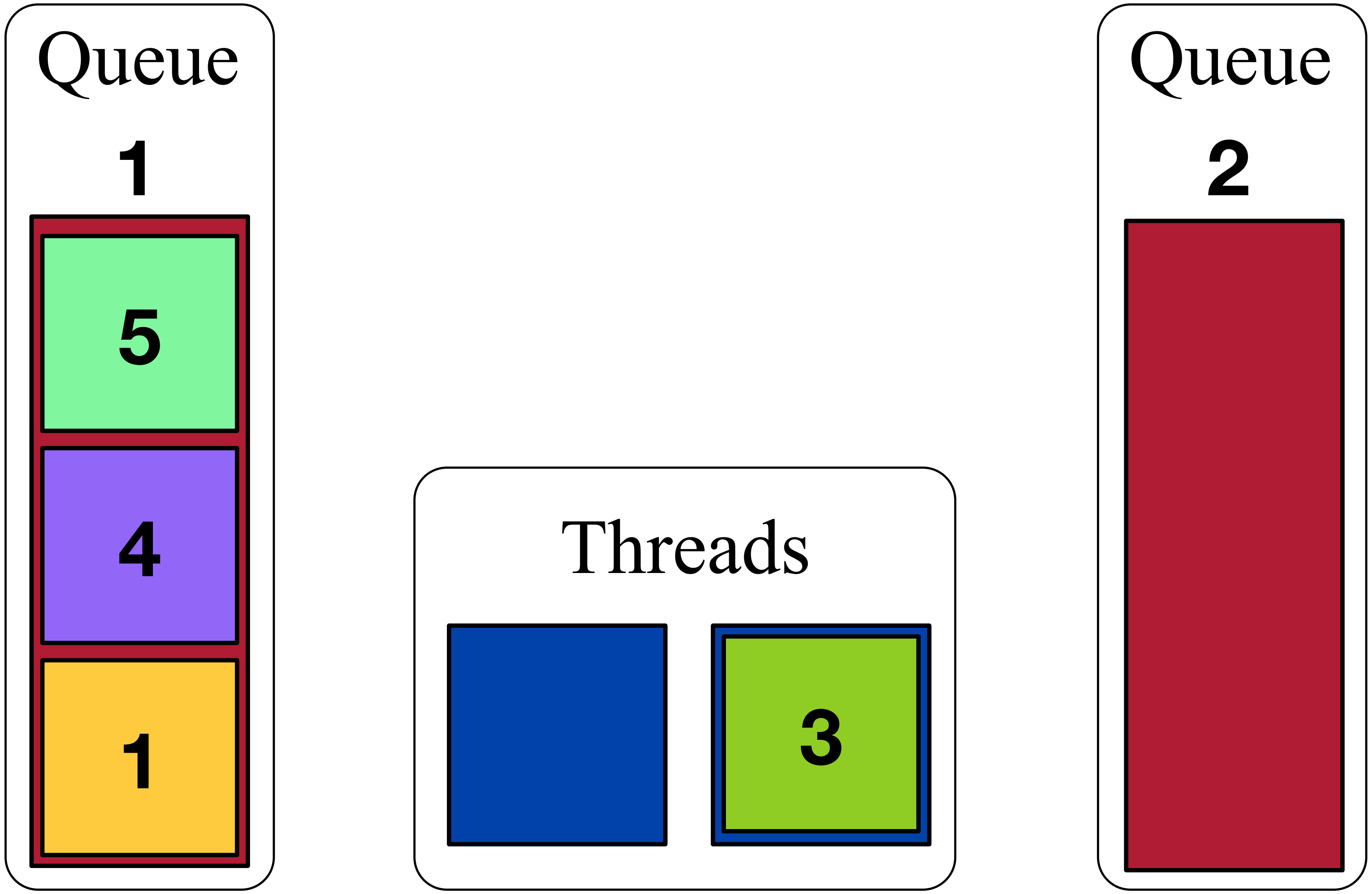
TBB Task Model

Spawned tasks go into same queue as creating task



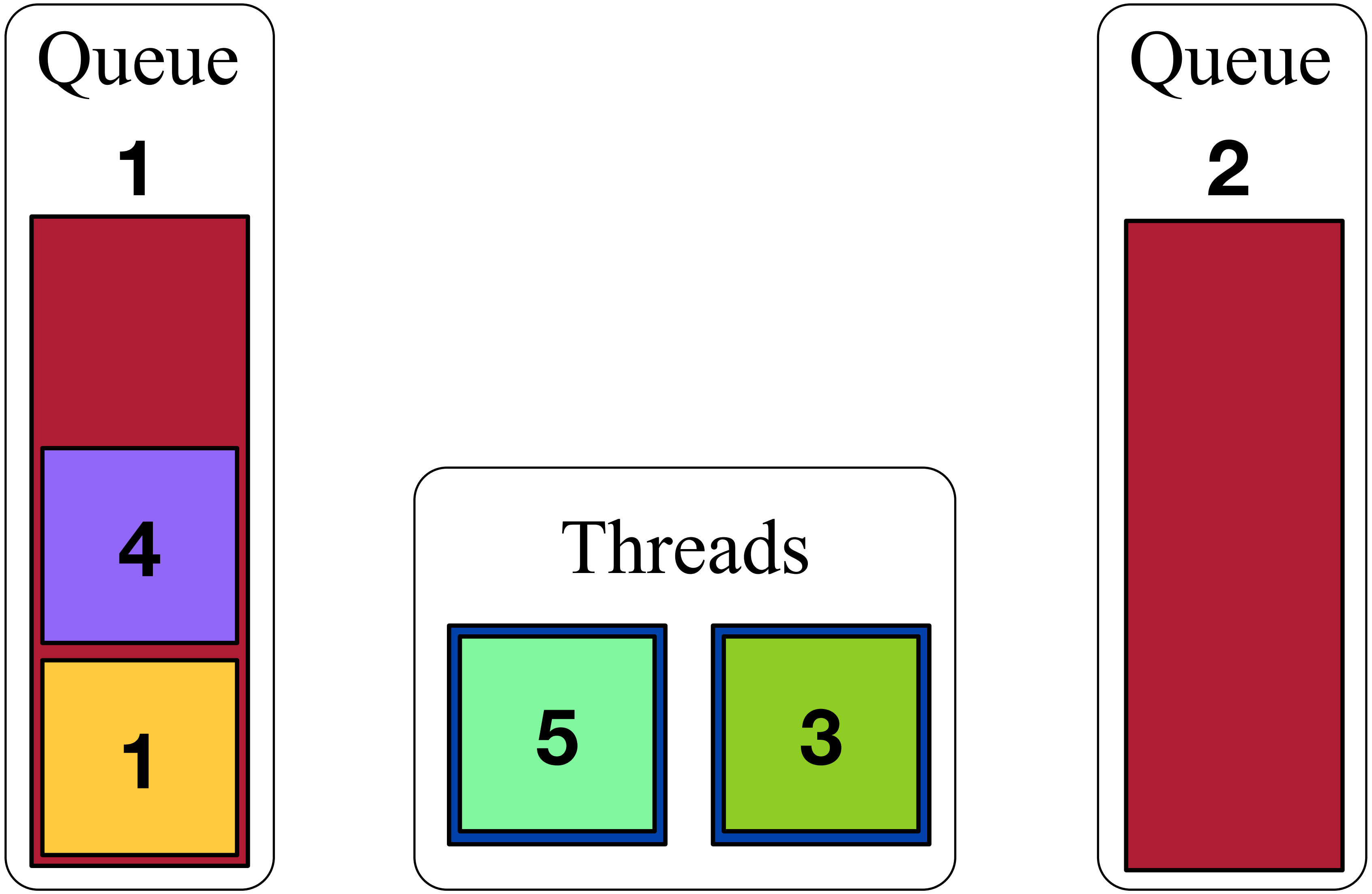
TBB Task Model

Spawned tasks go into same queue as creating task



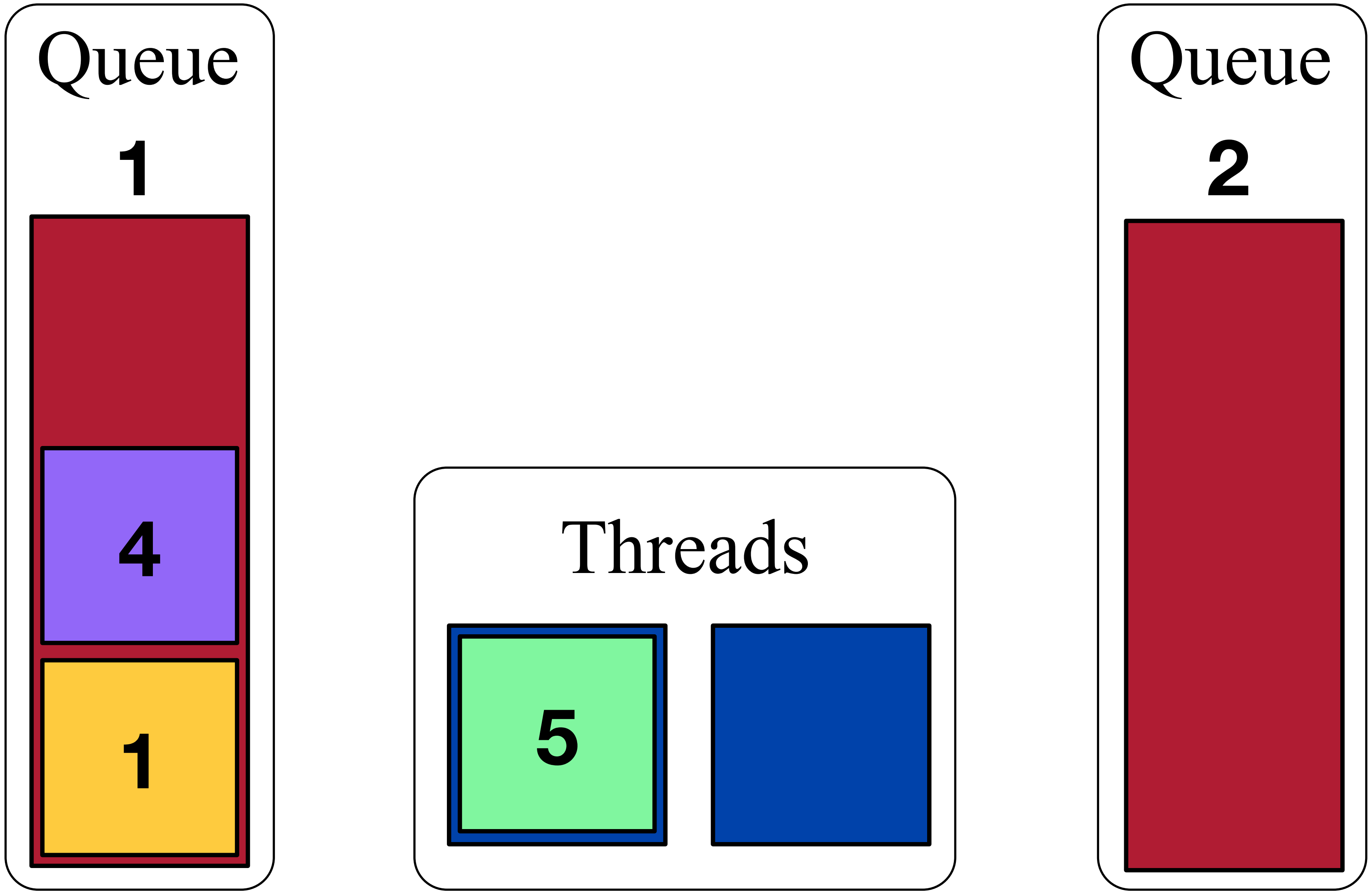
TBB Task Model

Spawned tasks go into same queue as creating task



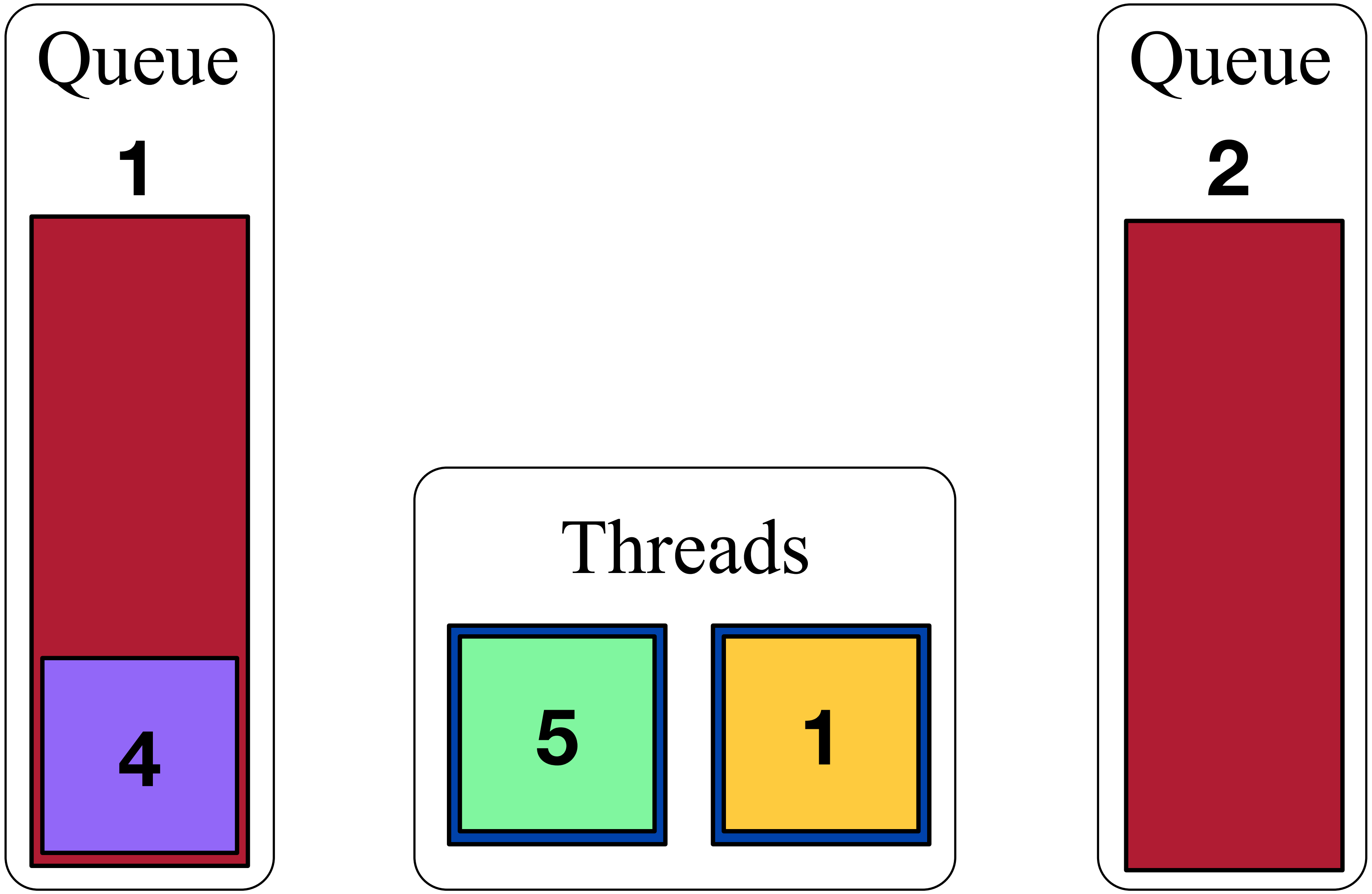
TBB Task Model

An empty thread queue steals oldest task from another queue



TBB Task Model

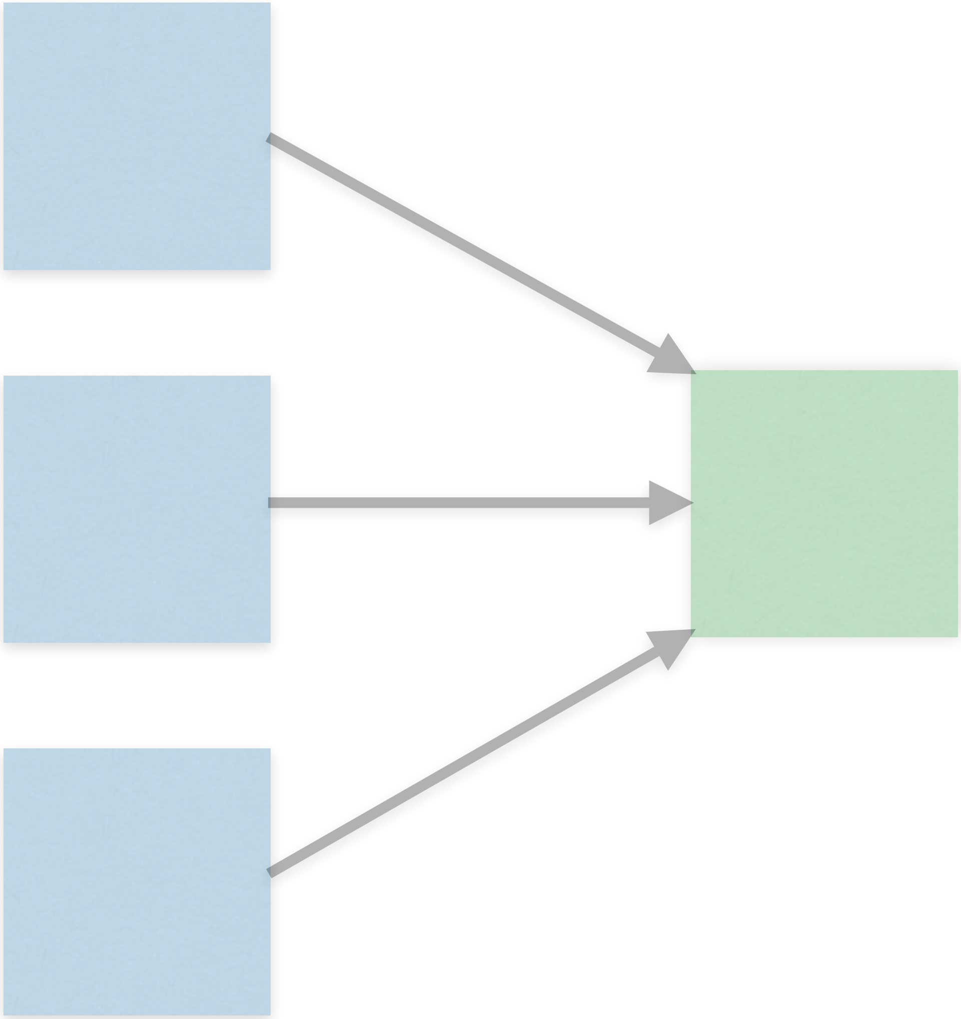
An empty thread queue steals oldest task from another queue



Data Prefetching

Data for modules are prefetched asynchronously

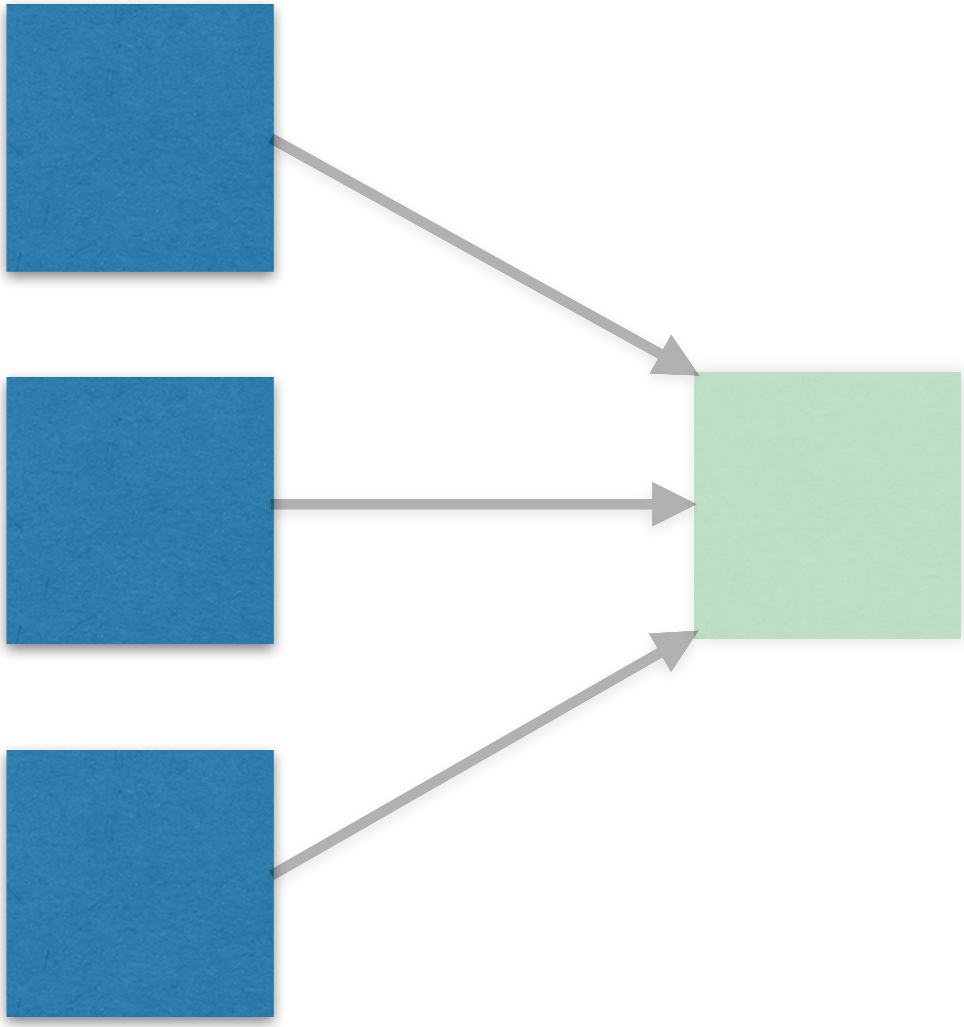
- Done when framework decides a module should be run for that Event
- Provides a large number of tasks for TBB to schedule
- Module starts after all prefetches have finished



Data Prefetching

Data for modules are prefetched asynchronously

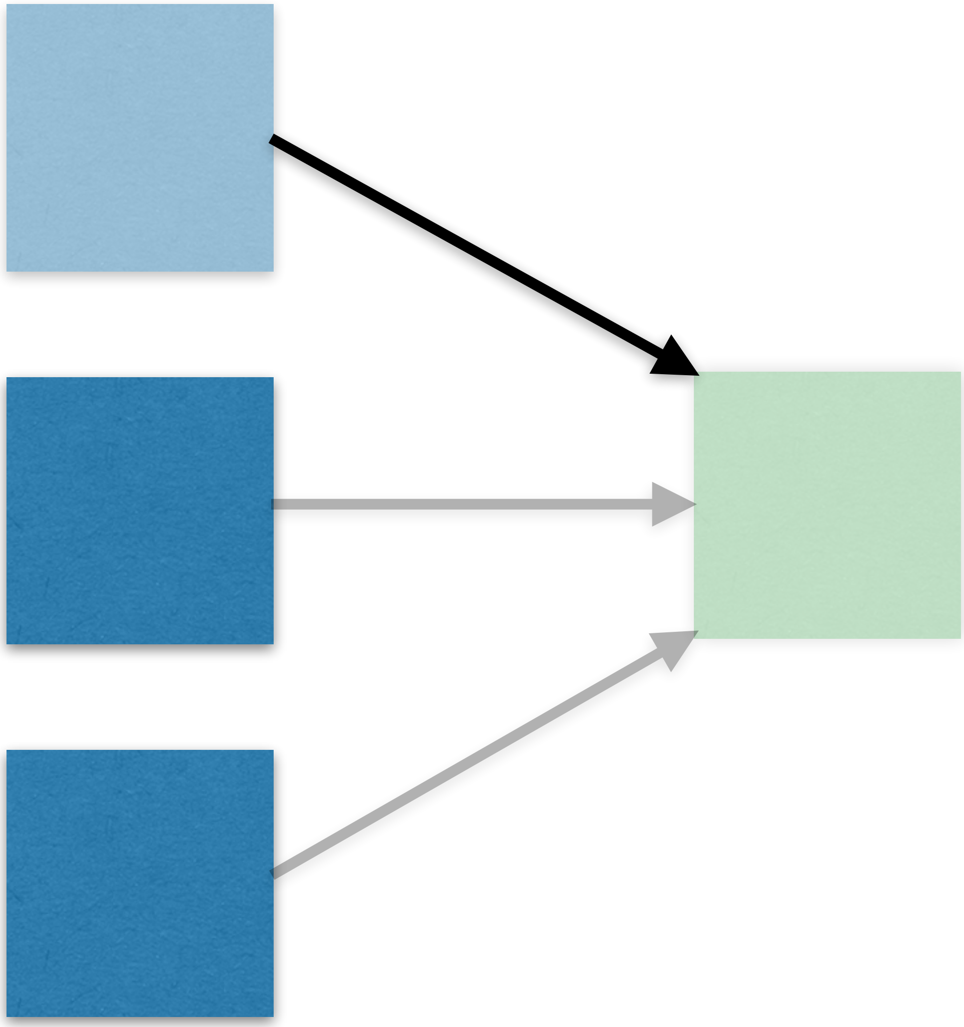
- Done when framework decides a module should be run for that Event
- Provides a large number of tasks for TBB to schedule
- Module starts after all prefetches have finished



Data Prefetching

Data for modules are prefetched asynchronously

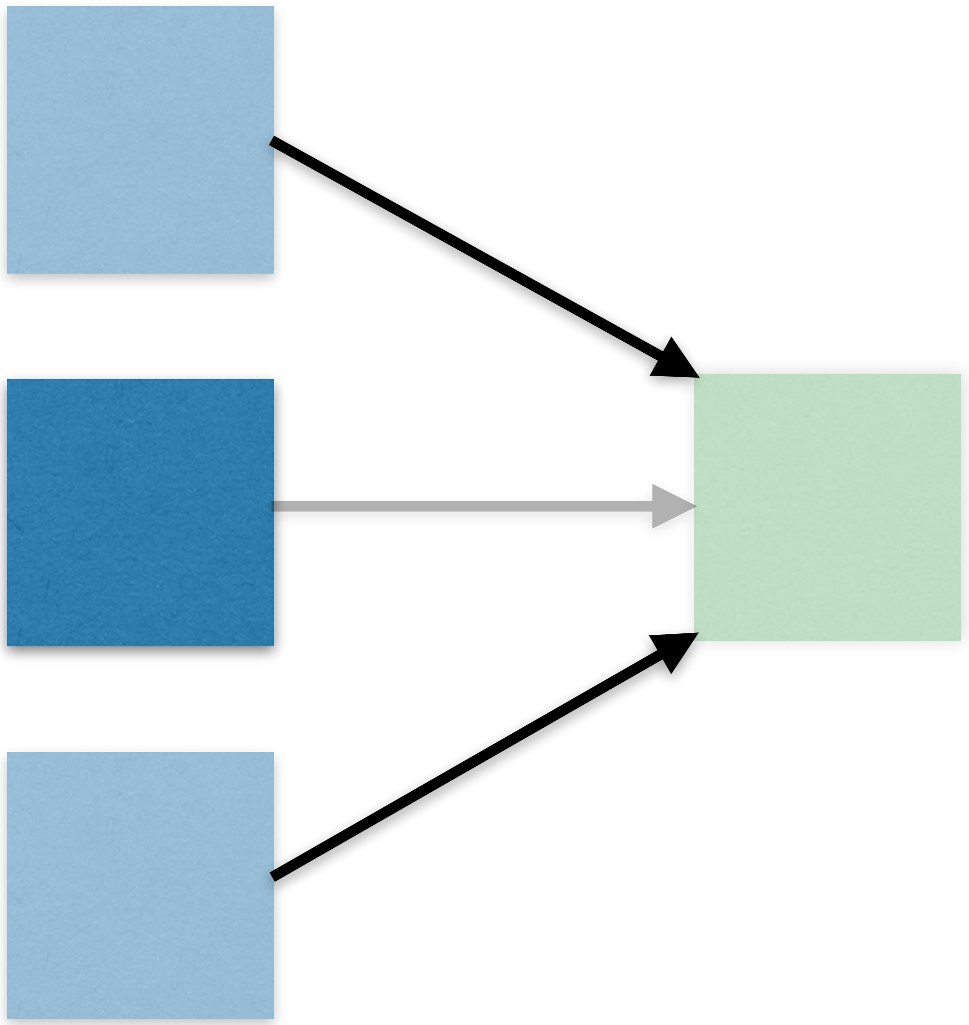
- Done when framework decides a module should be run for that Event
- Provides a large number of tasks for TBB to schedule
- Module starts after all prefetches have finished



Data Prefetching

Data for modules are prefetched asynchronously

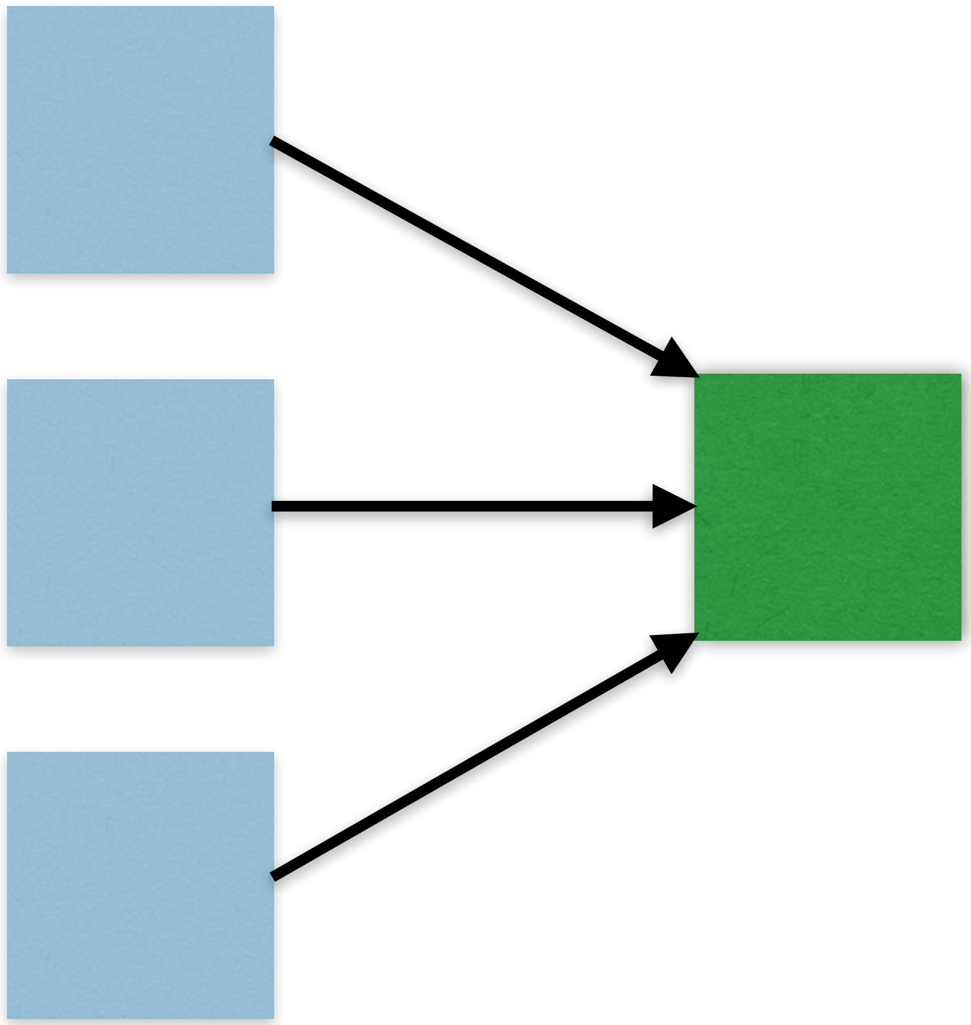
- Done when framework decides a module should be run for that Event
- Provides a large number of tasks for TBB to schedule
- Module starts after all prefetches have finished



Data Prefetching

Data for modules are prefetched asynchronously

- Done when framework decides a module should be run for that Event
- Provides a large number of tasks for TBB to schedule
- Module starts after all prefetches have finished



Module Threading Types

Modules are implemented based on threading types

Re-entrant

- Multiple events can simultaneously run the same instance of a module

Replicated

- Each concurrent event has its own copy of a module
- Since number of concurrent events is set at configuration the modules are made early

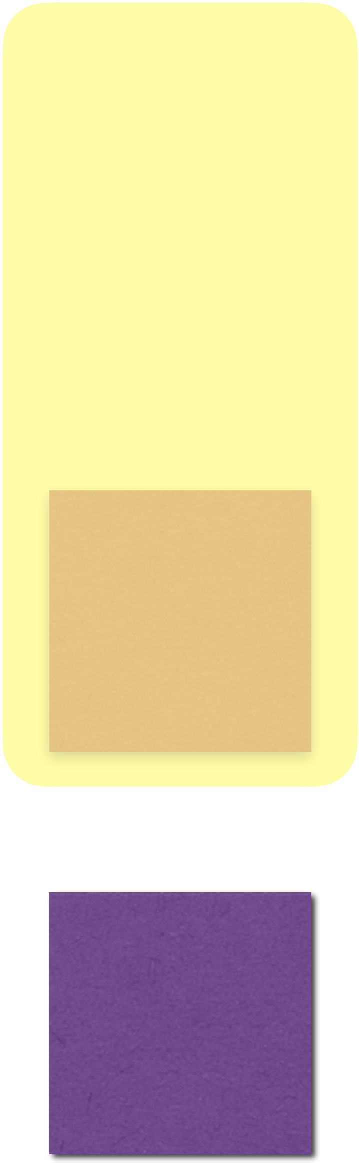
One

- Only one instance of the module
- Only one event at a time can interact with the module
- Cross event scheduling handled by a serial task queue

Serial Task Queue

A serial task queue guards a shared resource

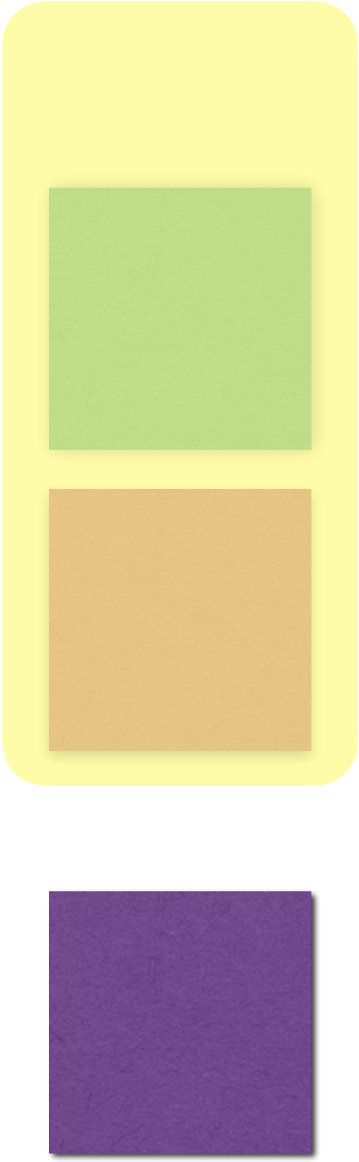
- Modules needing the resource have their *to run* task placed in the appropriate queue
- When a task from the queue finishes, it automatically spawns the next task in the queue



Serial Task Queue

A serial task queue guards a shared resource

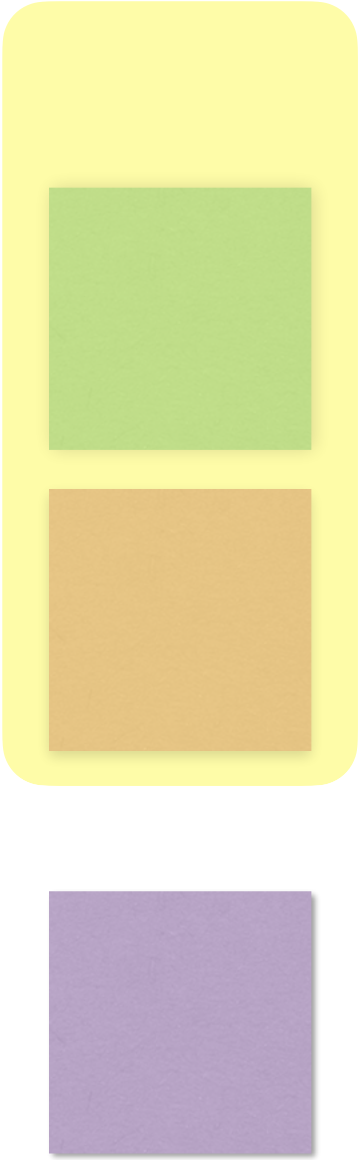
- Modules needing the resource have their *to run* task placed in the appropriate queue
- When a task from the queue finishes, it automatically spawns the next task in the queue



Serial Task Queue

A serial task queue guards a shared resource

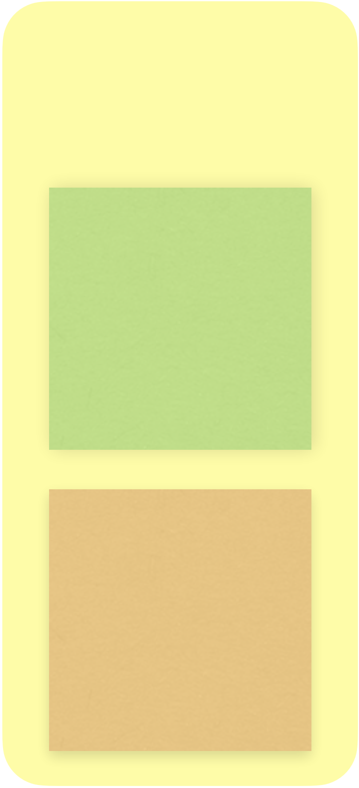
- Modules needing the resource have their *to run* task placed in the appropriate queue
- When a task from the queue finishes, it automatically spawns the next task in the queue



Serial Task Queue

A serial task queue guards a shared resource

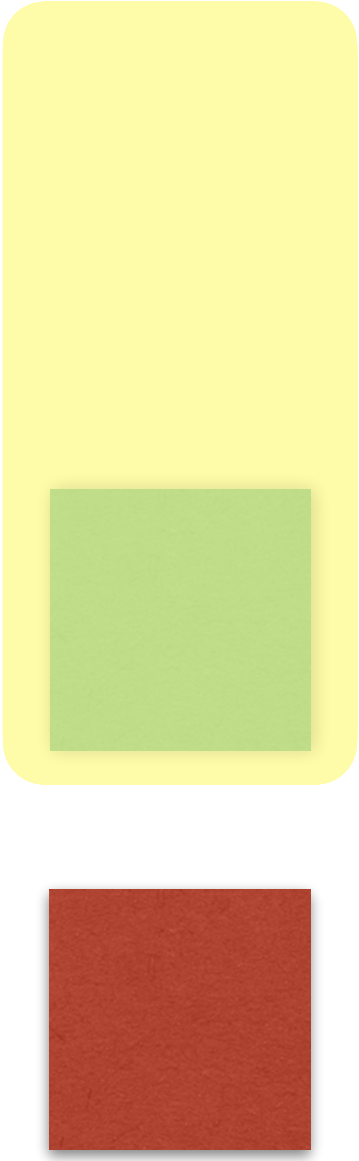
- Modules needing the resource have their *to run* task placed in the appropriate queue
- When a task from the queue finishes, it automatically spawns the next task in the queue



Serial Task Queue

A serial task queue guards a shared resource

- Modules needing the resource have their *to run* task placed in the appropriate queue
- When a task from the queue finishes, it automatically spawns the next task in the queue



Processing Stalls

If had no shared resources, CMS's threading design scales perfectly

- If have as many threads as concurrent events
- Each thread would always have a tbb task to run for a given thread
- No cross thread communication needed

Sharing resource across concurrently running events can cause stalls

- All reads from a ROOT file must be serialized
- Writing to a ROOT output file must be serialized
  - NOTE: can write to different ROOT output files simultaneously
- *One* modules are shared across events but are not thread-safe

Mitigate stalls by having many more scheduled tasks than threads

- Running Paths/EndPaths concurrently creates many tasks
- Prefetching data products creates many tasks
- Having concurrent events creates many tasks
- Using parallel algorithms within a module can create many tasks

Example of Stall Mitigation

Simple job configuration

- 2 threads
- 2 concurrent events
- 2 *One* OutputModules (**A** and **B**) both on same EndPath

During event processing loop

- Both threads put requests to run **A** and **B** into their respective serial task queues
- The first thread to add a task to a serial task queue will have the task spawned to TBB
- A task for both **A** and **B** will be available to TBB
  - Could be for the same event or for different events depending on exact timing of calls
- Both tasks will be run
  - Either because they were spawned in different threads or one thread stole from the other
- When a tasks finish, it will run the other task waiting in the queue
- At all times both threads are busy

Stall Mitigation Performance Measurement

Machine for testing

- Westmere-EP L5640 CPU with 6 cores x 2 hyper-threads

Compared Reconstruction jobs

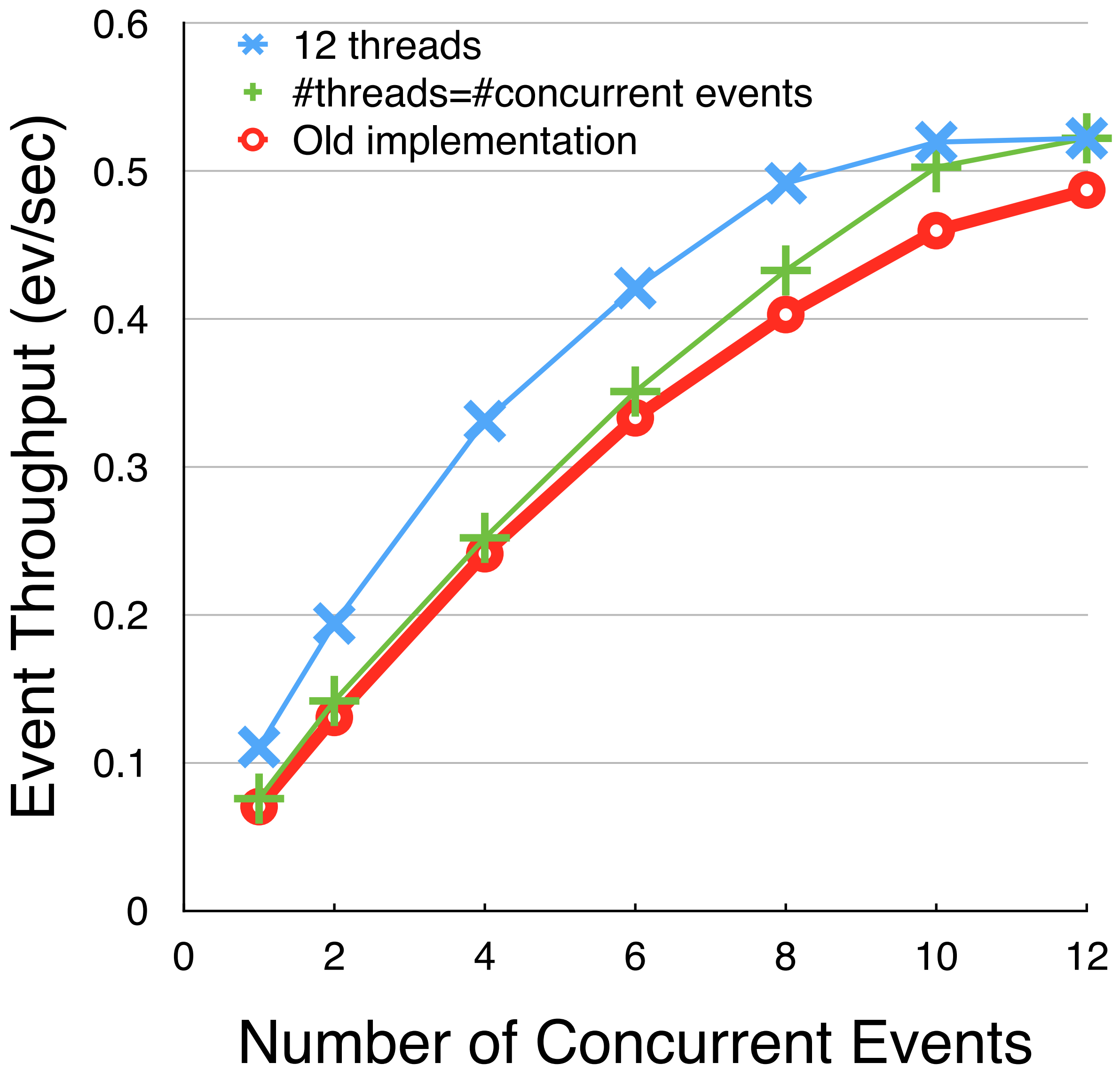
- Old *one-thread-per-event* implementation which runs modules in a fixed order
  - used mutex to guard shared resources
- Full system with number of threads == number of concurrent events
- Full system with number of threads == 12 (machines max)

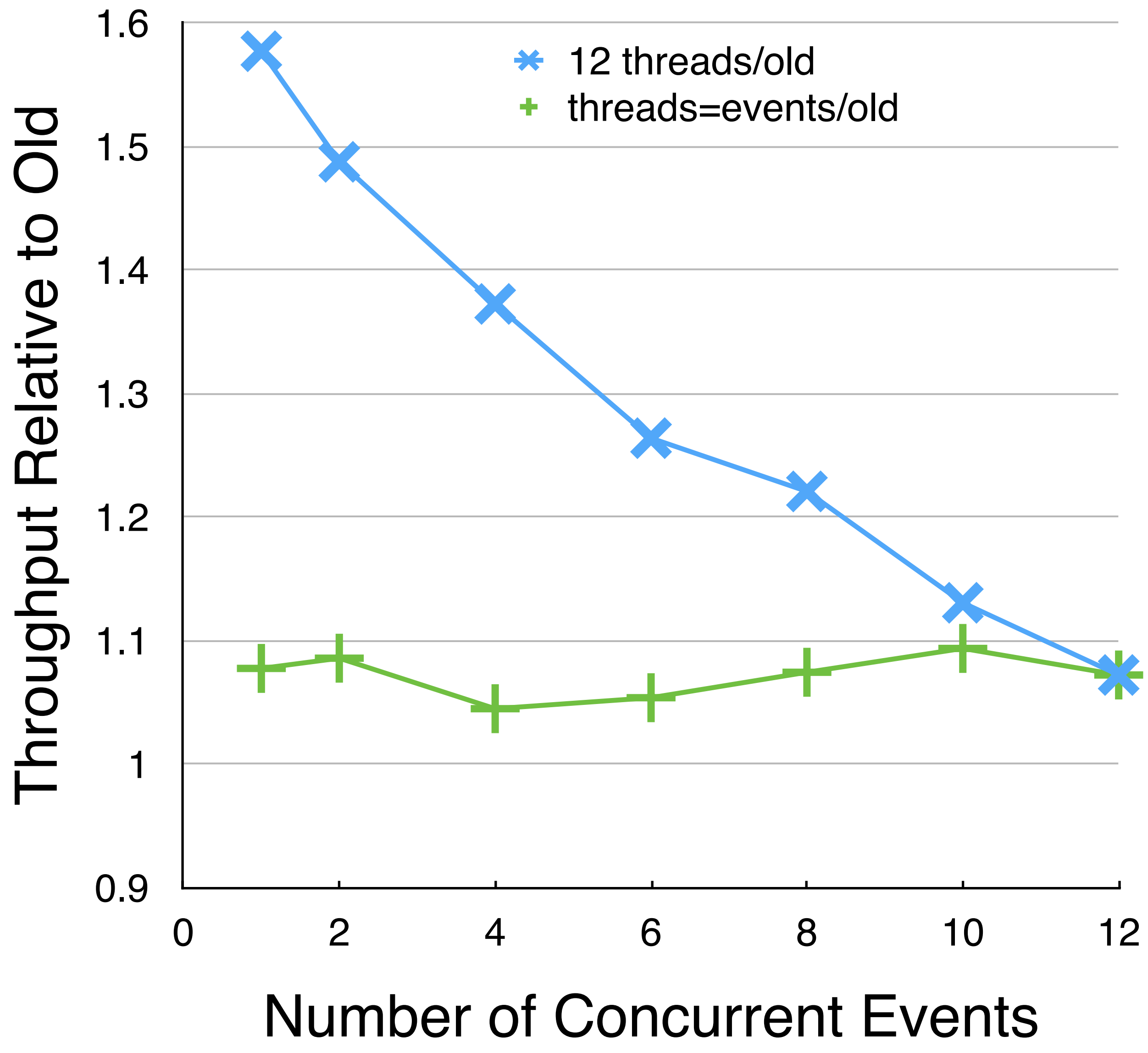
Reconstruction configuration summary

- 3 OutputModules
- 1780 other modules
- 21 Paths

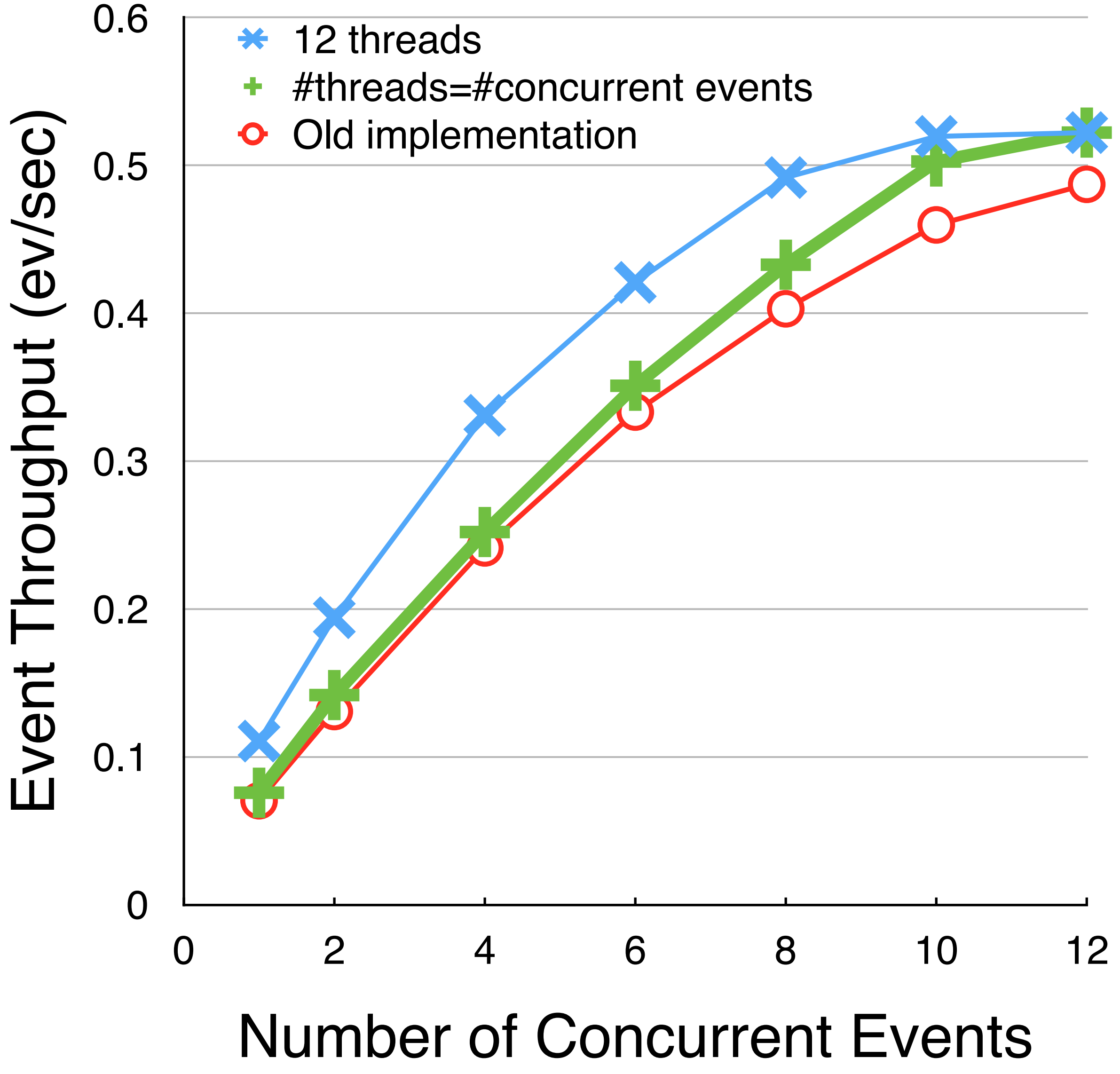
NOTE: this measurement is from 2017 and CMSSW has had further improvements

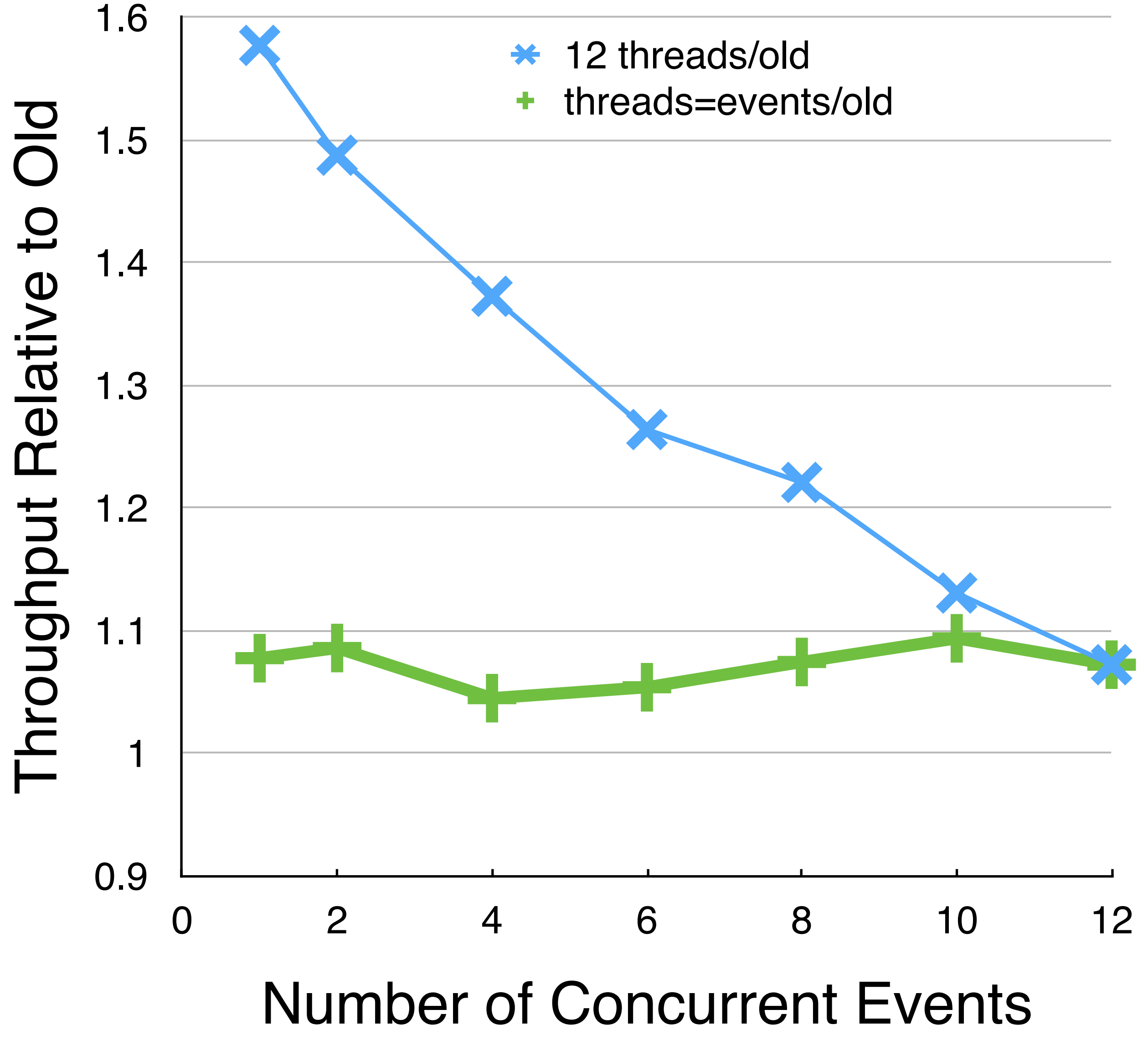
Throughput Comparisons



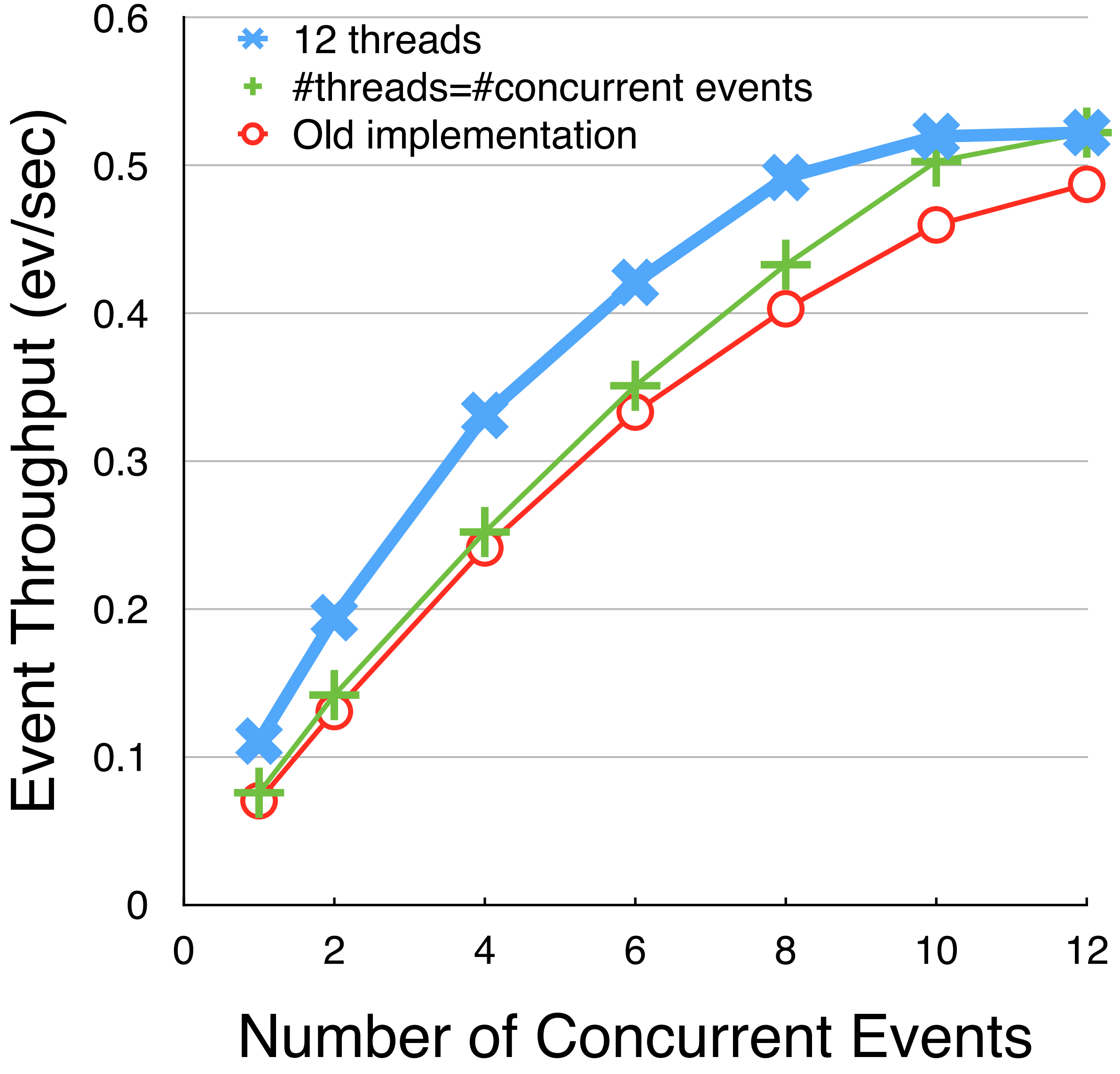


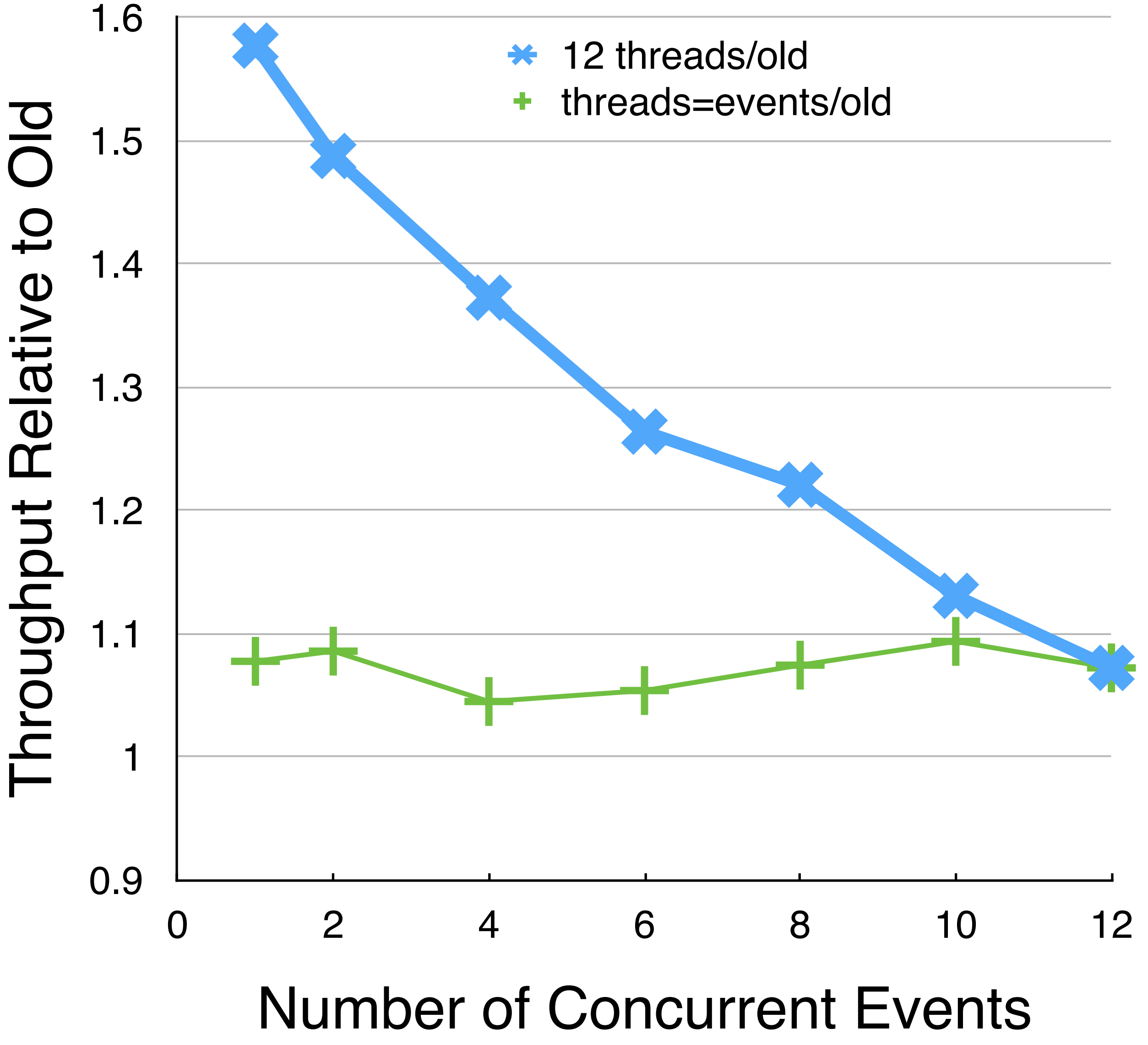
Throughput Comparisons





Throughput Comparisons





Stall Mitigation Findings

Stalls were solely caused by just one of the OutputModules

- The one which takes longest per event

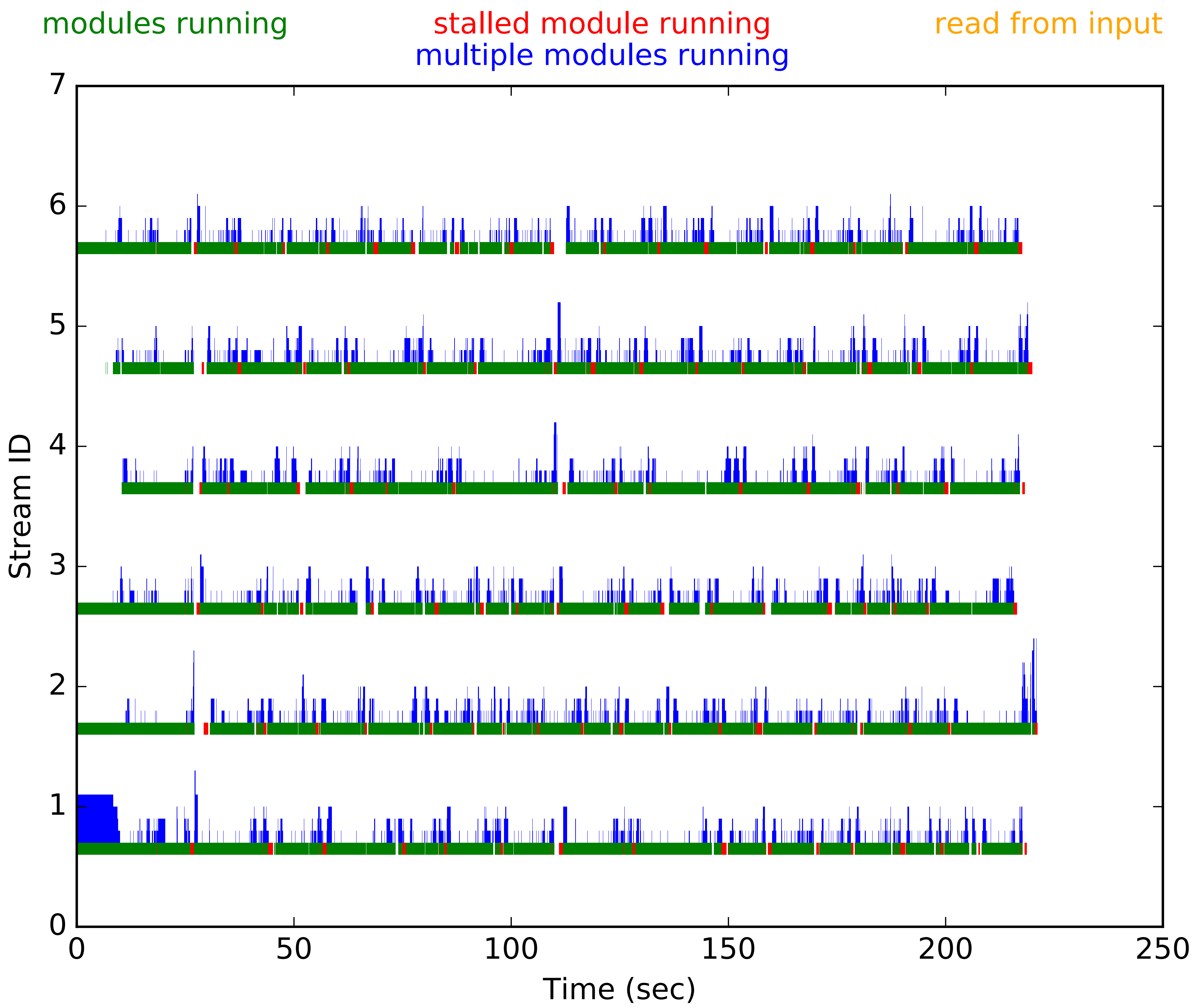
Full system's scheduling allows stall mitigation

- Framework can re-order the run order of the OutputModules
- Many tasks available per thread allows threads to stay busy

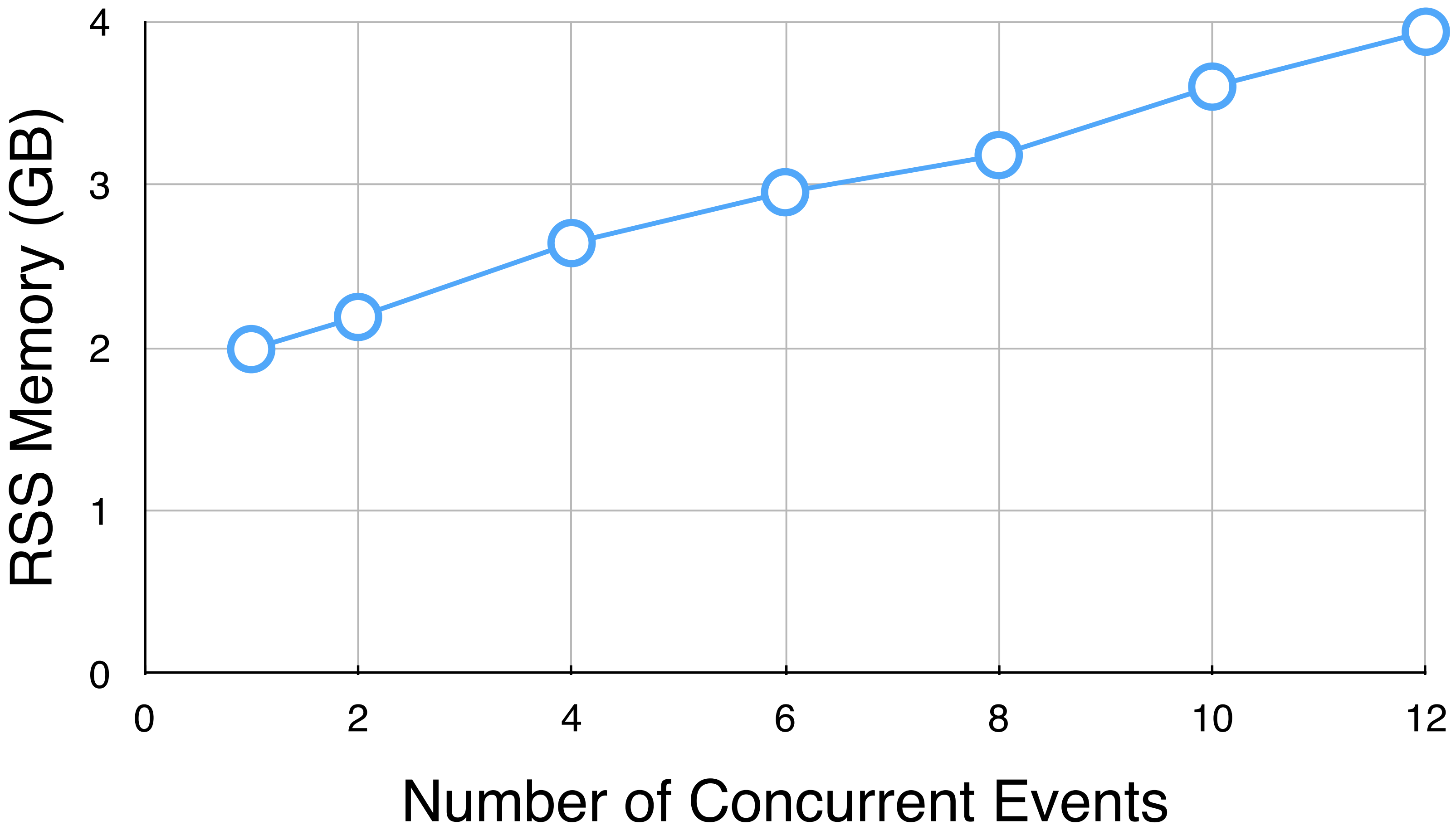
Additional threads increase throughput

- CMS has limited concurrency within an Event
  - about 1.6 threads per event is the maximum concurrency
- limited because of module dependencies and module run time

Reconstruction with 8 threads and 6 Concurrent Events



Memory Utilization



High initial memory

- ~2 GB

Memory grows slowly w.r.t number of concurrent events

- ~150 MB/event

Conclusions

CMS is happy with the threaded framework

- Have used in production since beginning of LHC Run 2 (2015)
- Used for all production workflows
  - online high level trigger farm, event generation, Geant simulation, and reconstruction
- Typically use 8 threads per job
  - Number of concurrent events == number of threads
  - This limit is primarily set by the grid site's batch slot configuration

The scheduling mechanism

- Has very little overhead even when only using very fast running modules
  - Using longer running modules the overhead is negligible
- Scales very well with number of threads
  - Only limited by the number of serial tasks queues in the job
  - The serial task queue around the Source is the ultimate limit at the moment