Performance of TBB Based Framework Demo

Christopher Jones *FNAL*

Outline

TBB Threading Model

Measurements

Conclusion

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 list

If a queue is empty, it will

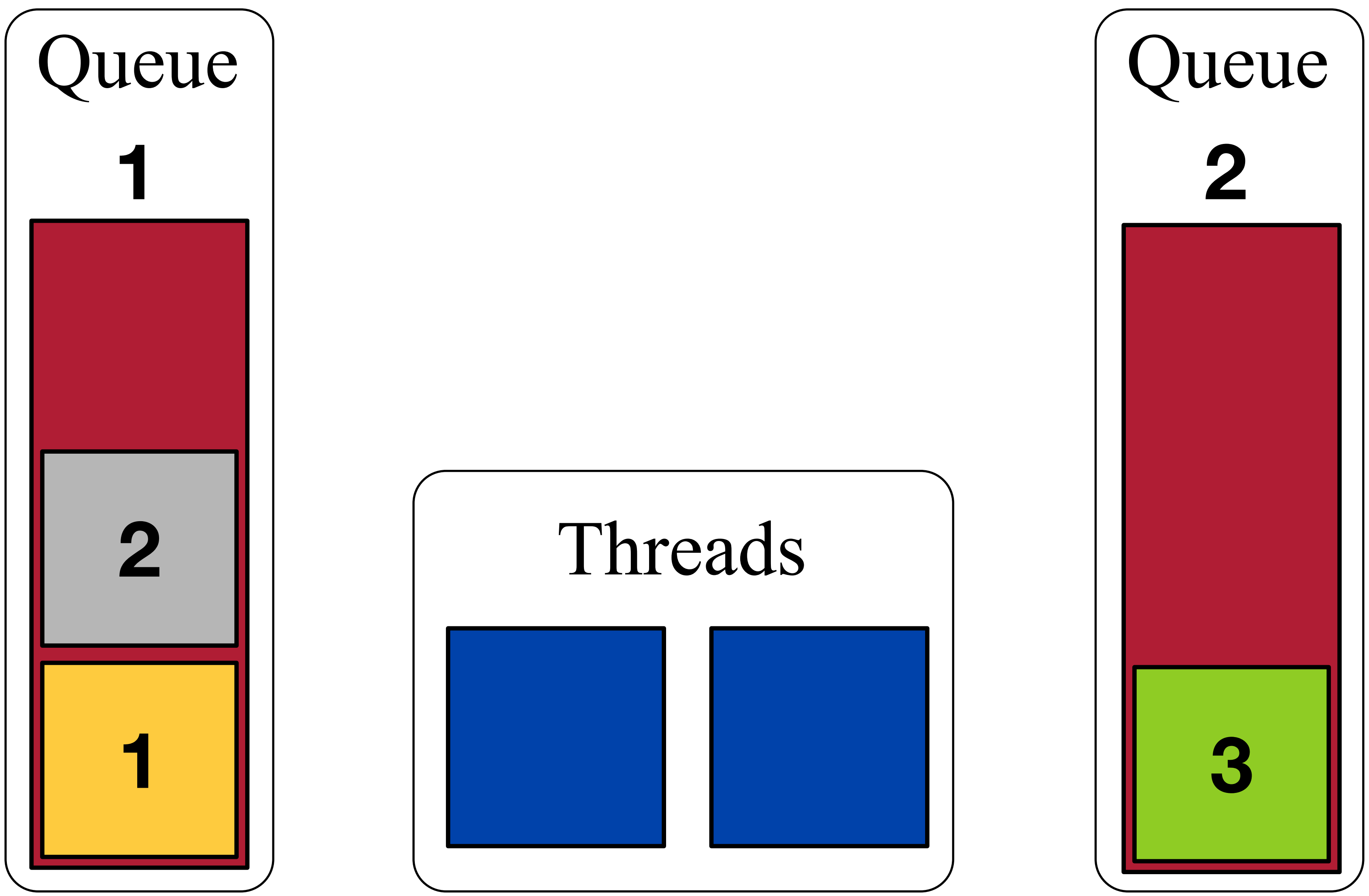
- See if a task is on the shared list and if so take the oldest one, else
- Steal oldest task from another queue

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

- Guaranteed to run on the same thread

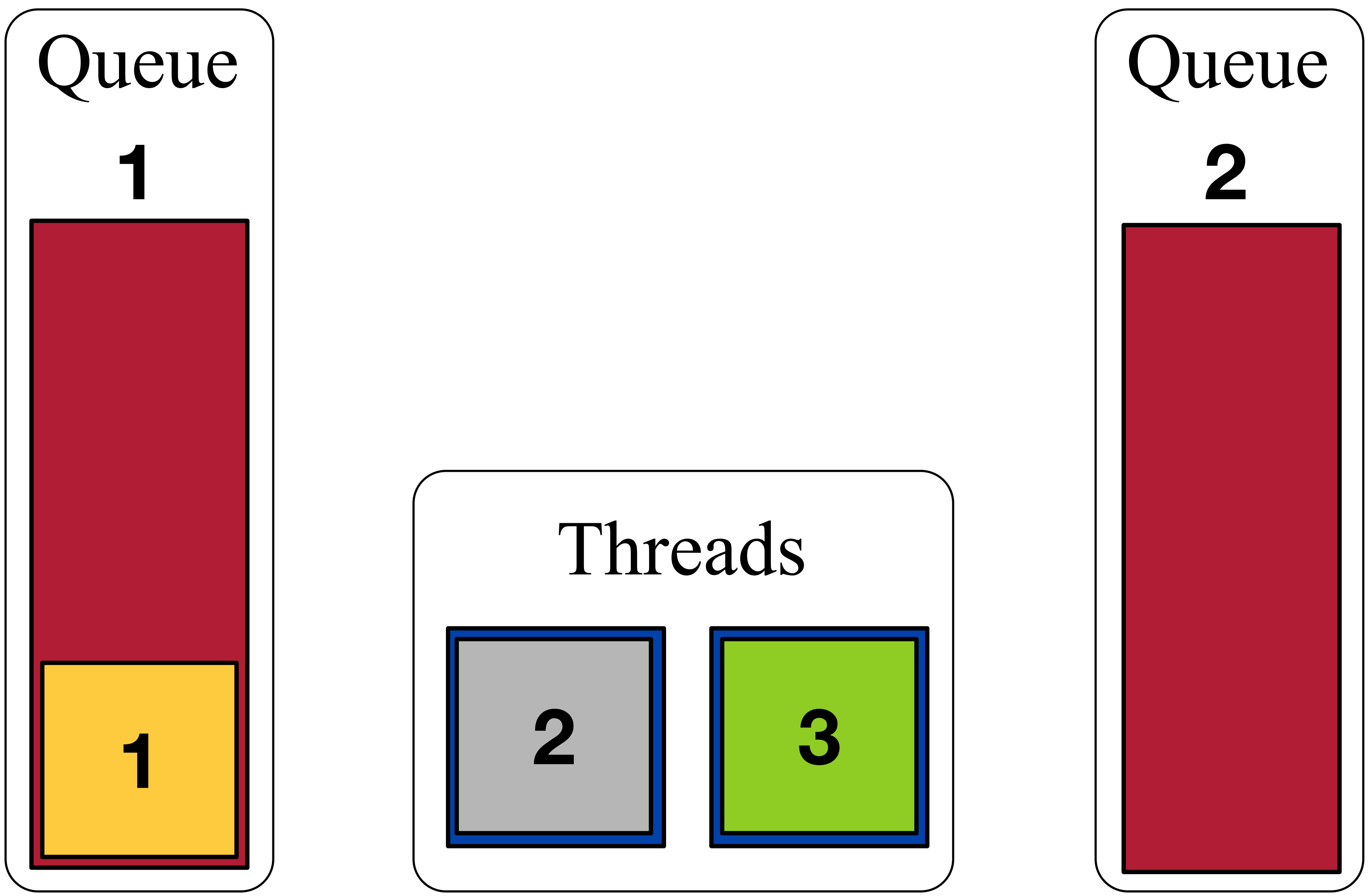
TBB Task Model

Tasks are pulled in Last In First Out order



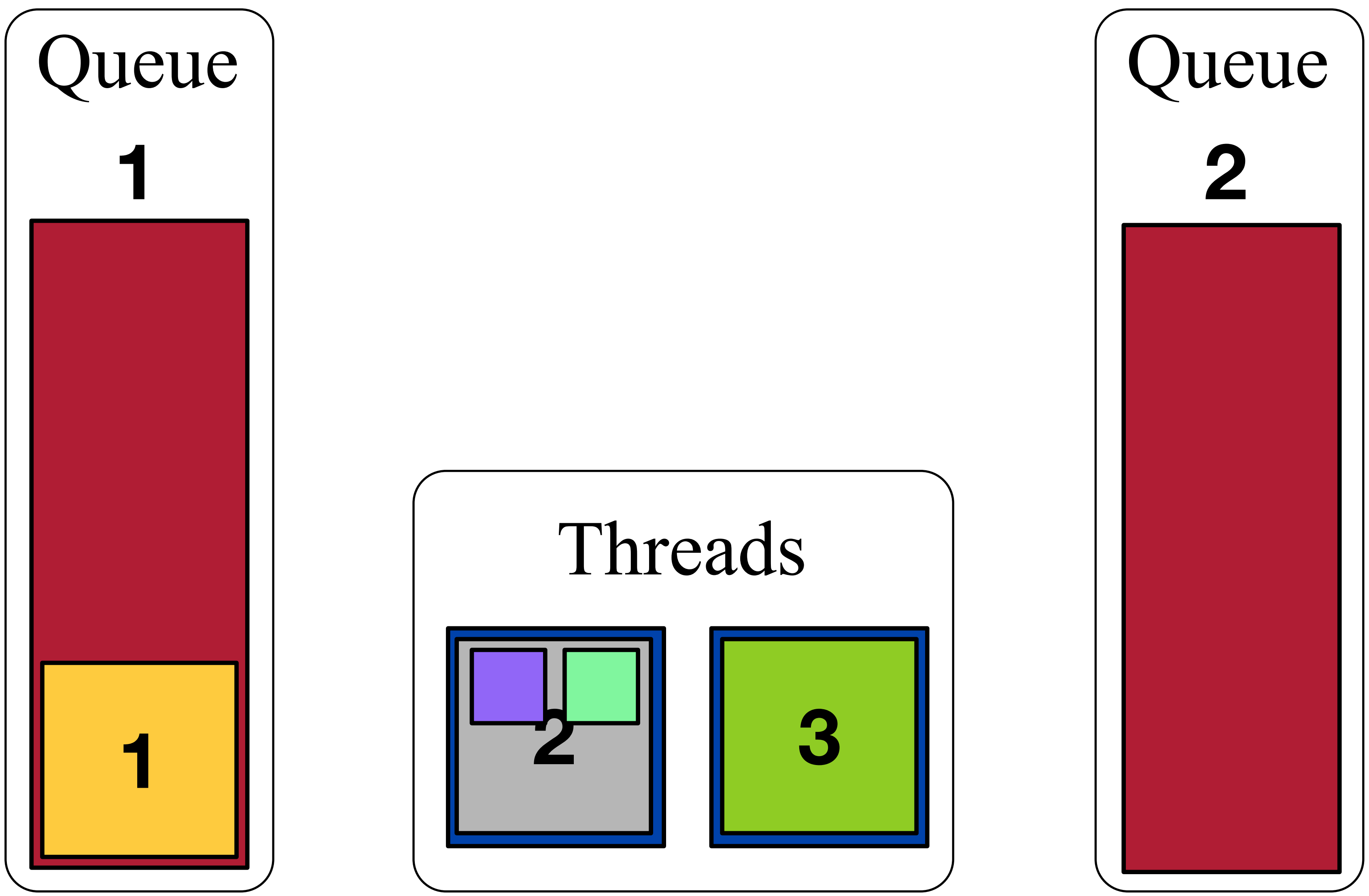
TBB Task Model

Tasks are pulled in Last In First Out order



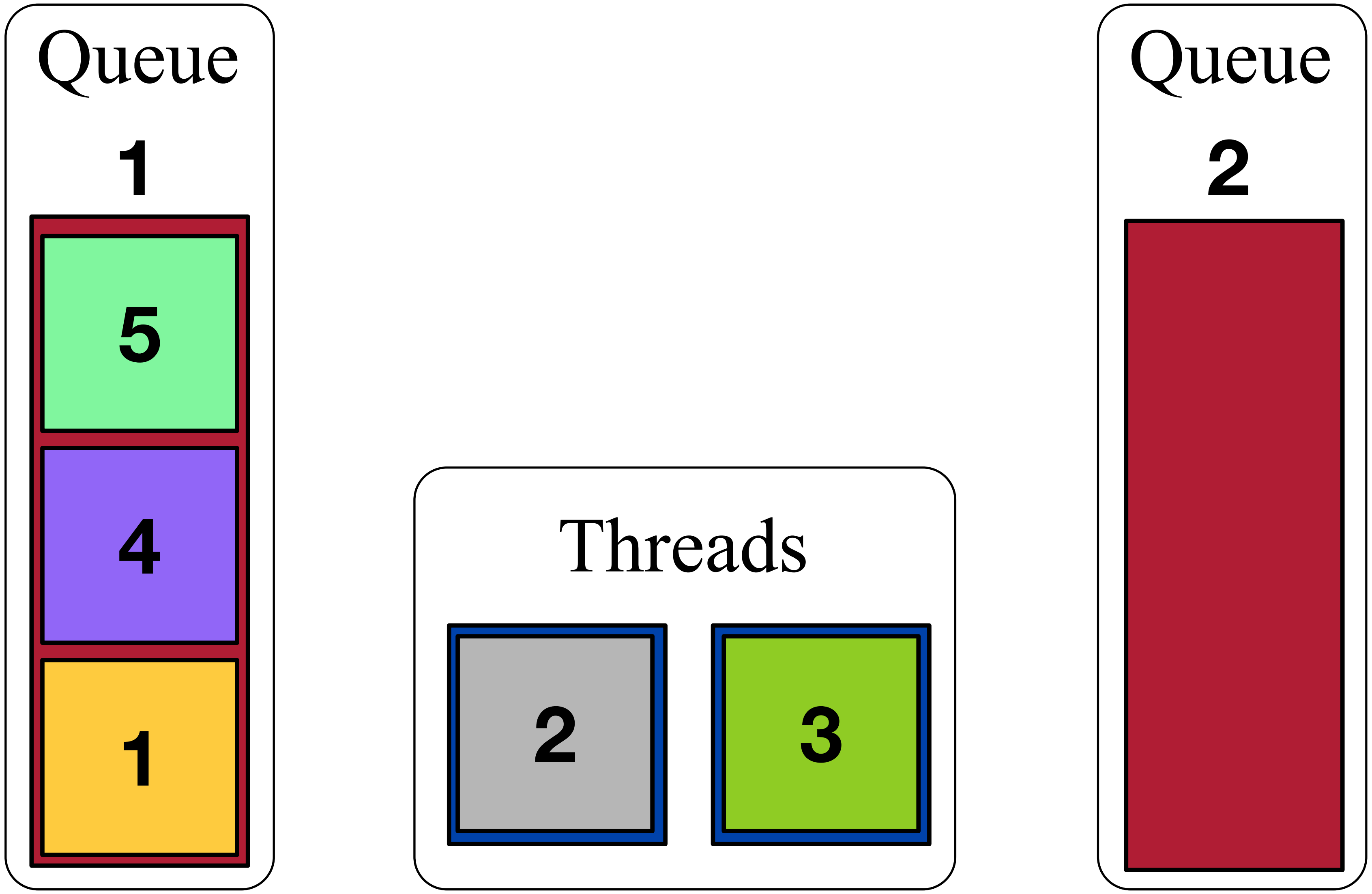
TBB Task Model

Spawned tasks go into the same thread queue as creating task



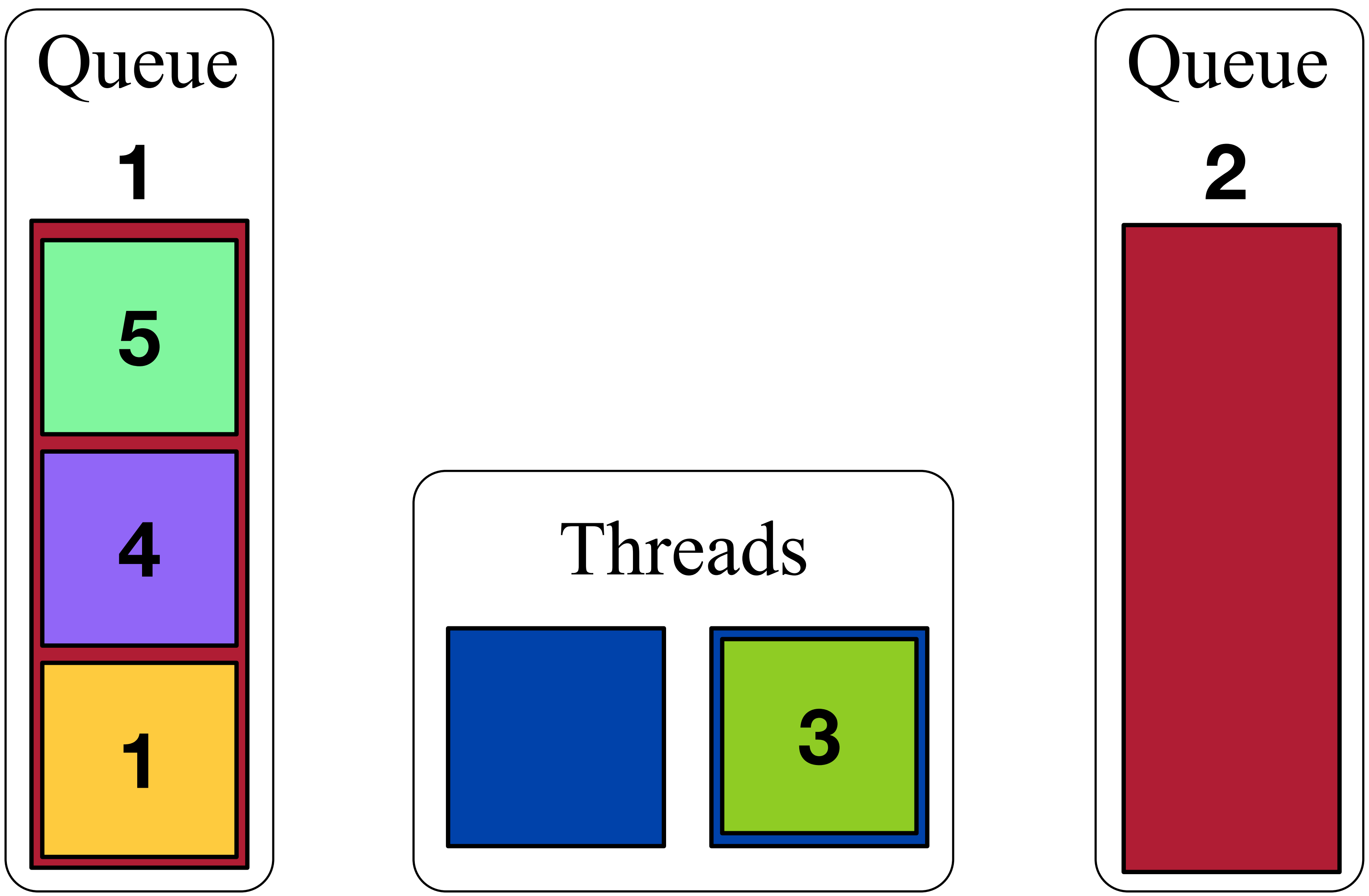
TBB Task Model

Spawned tasks go into the same thread queue as creating task



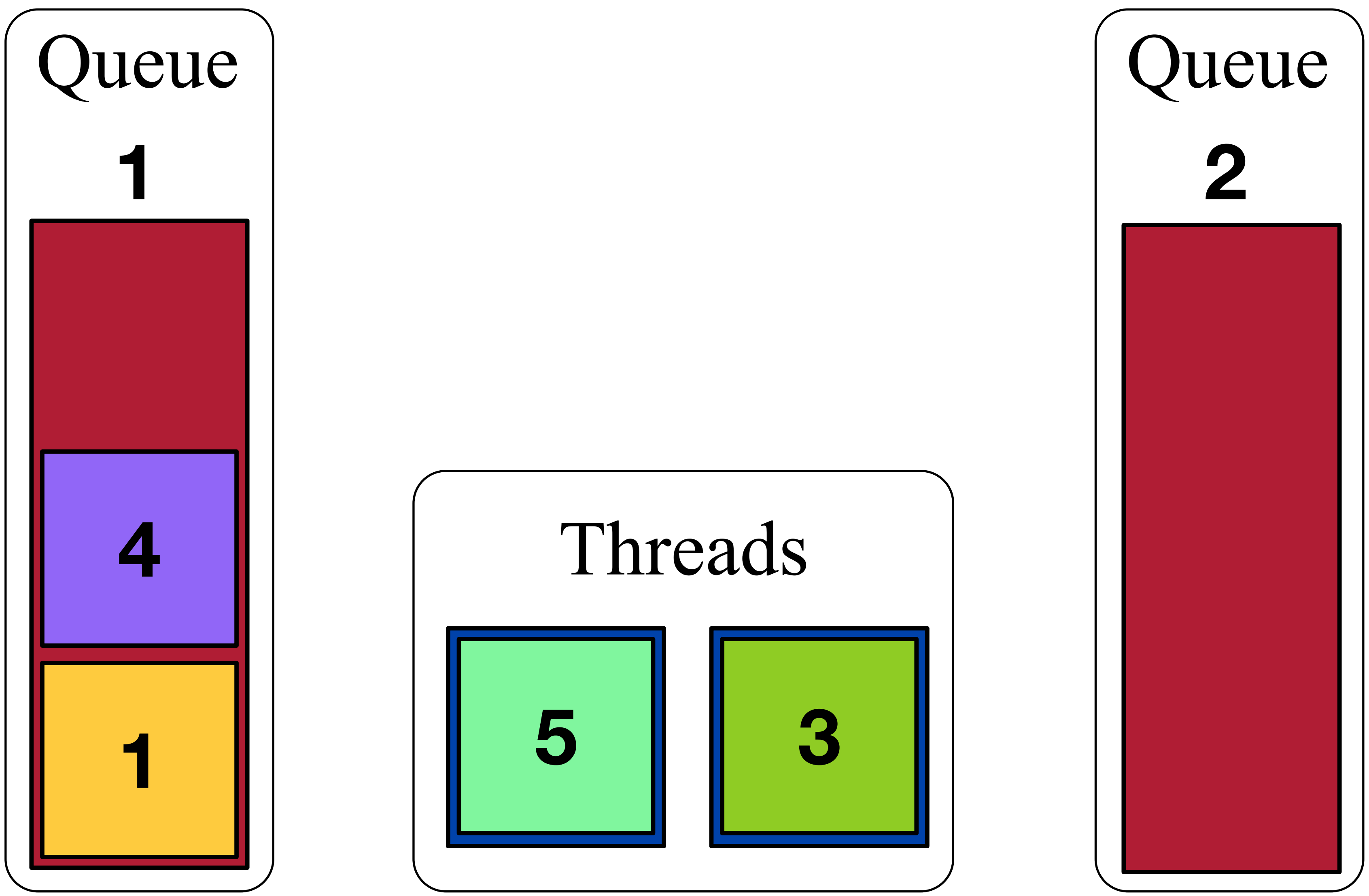
TBB Task Model

Spawned tasks go into the same thread queue as creating task



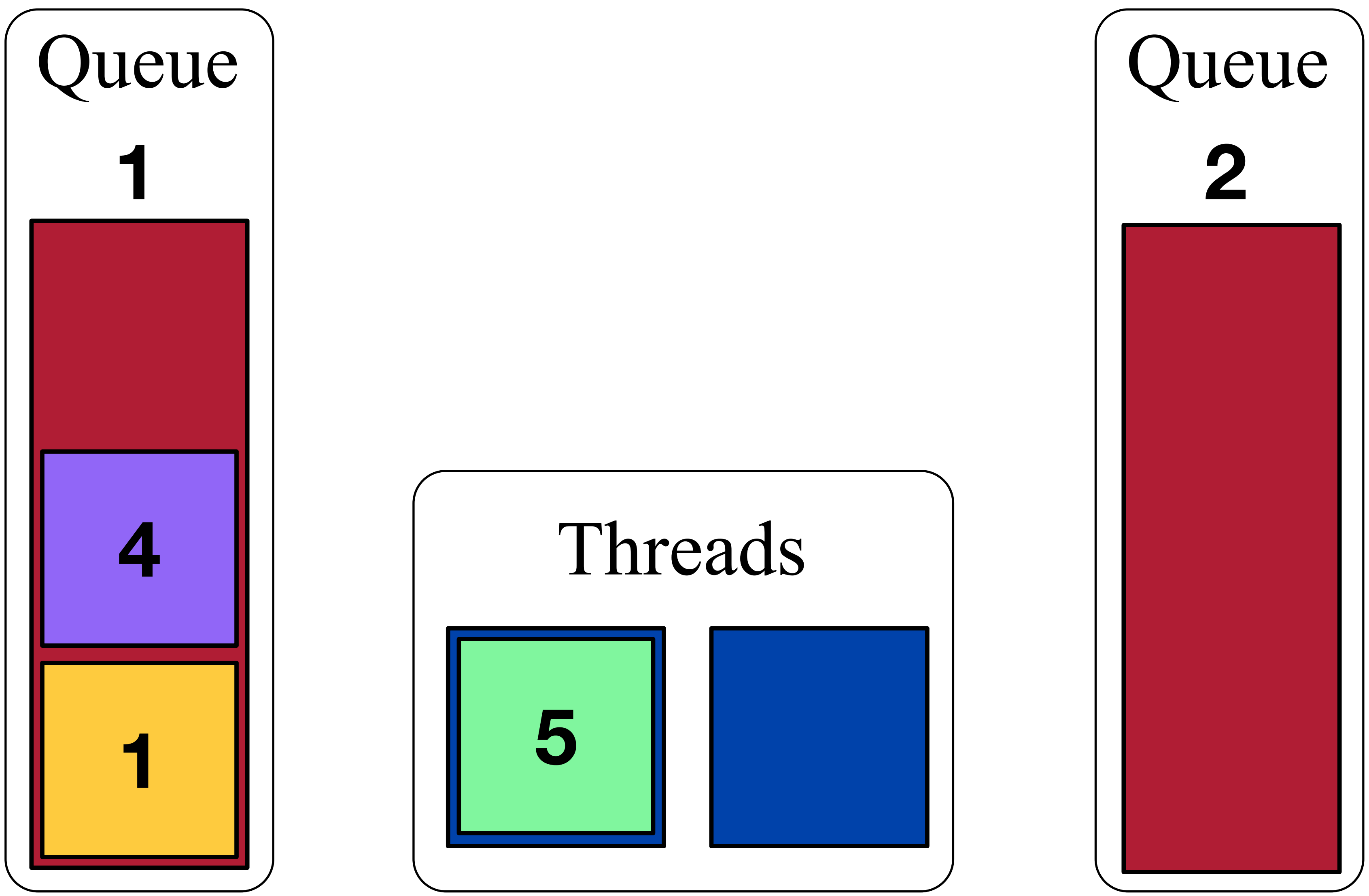
TBB Task Model

Spawned tasks go into the same thread queue as creating task



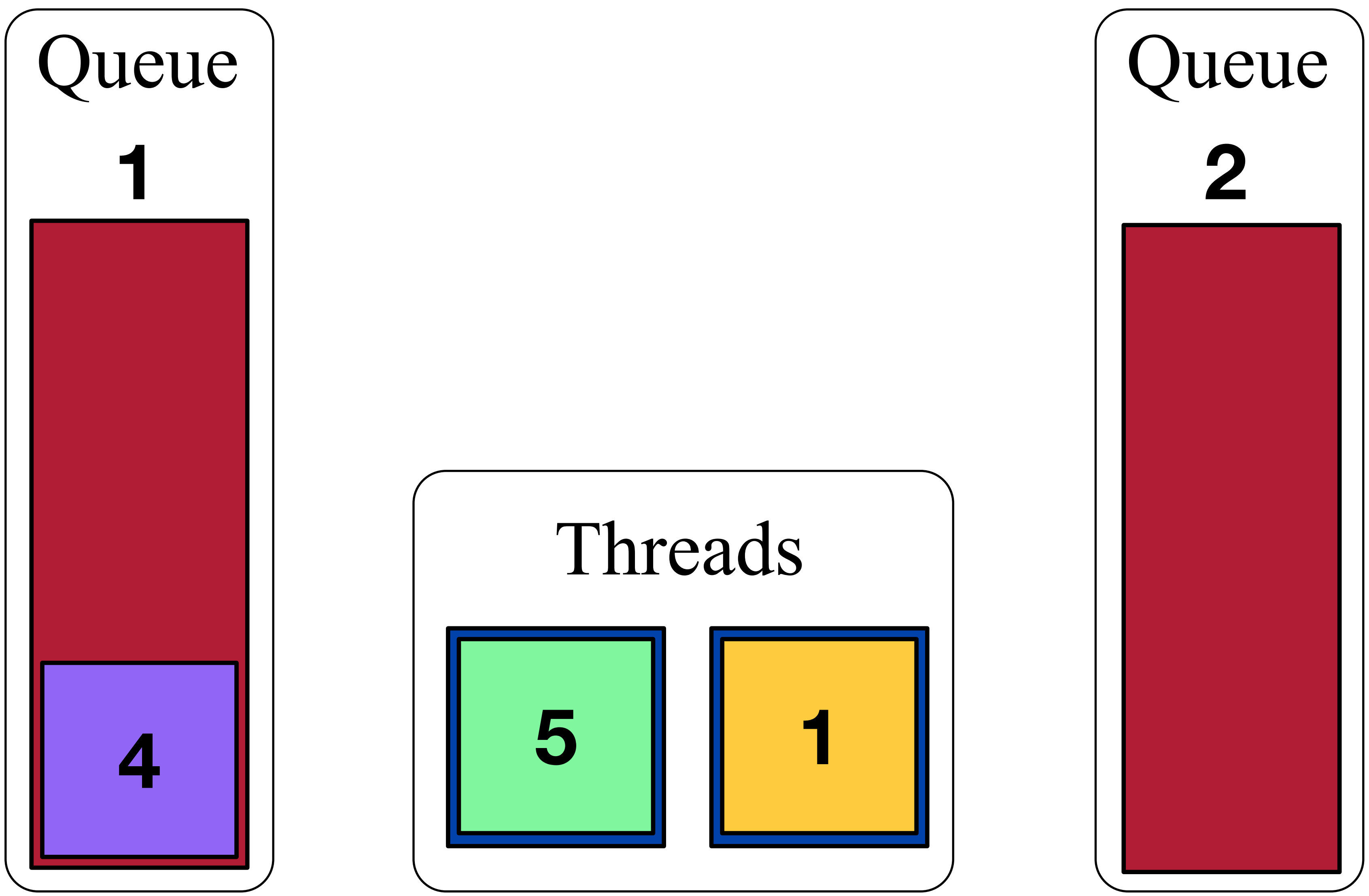
TBB Task Model

An empty thread queue steals oldest task from another queue



TBB Task Model

An empty thread queue steals oldest task from another queue



Model's Advantage

Related tasks are scheduled on same thread when possible

Improves data locality

- A task that is waiting for another task probably wants data it creates
- Right after prefetch data task completes the module's task will then run

Should create a smaller list of waiting tasks in the thread queues

- TBB does a depth first traversal of tasks while libdispatch does a breadth first
- The prefetching and then running one module will happen before going on to prefetch and run the next module

Synchronization

TBB has only primitive task synchronization mechanisms

- task::execute() can return a pointer to the next task that must be run
- A task can have children and when all children have finished the parent will run
  - A child can only have one parent
- There are no advanced mechanisms to deal with resource contention
  - Only mutex

I built tools on top of TBB primitives

- WaitingTaskList
  - Can hold a series of tasks which will be spawned when 'doneWaiting()' is called
  - Can safely add tasks simultaneously
  - Can safely add tasks and call doneWaiting simultaneously
  - Tasks added after doneWaiting is called will be spawned immediately
- SerialTaskQueue
  - Can hold a series of tasks
  - Only one task will be run at a time
  - Tasks can be added simultaneously
  - Tasks are run on a first in first out basis

Measurement Strategy

Approximate reconstruction behavior

489 Producers

2 OutputModules

278 Producers have their data requested directly from OutputModule

Module Dependencies

What data each module uses

Such information is recorded by CMS framework already

Module Timing

Get per event module timing for 2011 high pileup data

~30 interactions per crossing

Feed dependencies and timing to demo framework

Compare timing to a simple single threaded demo framework

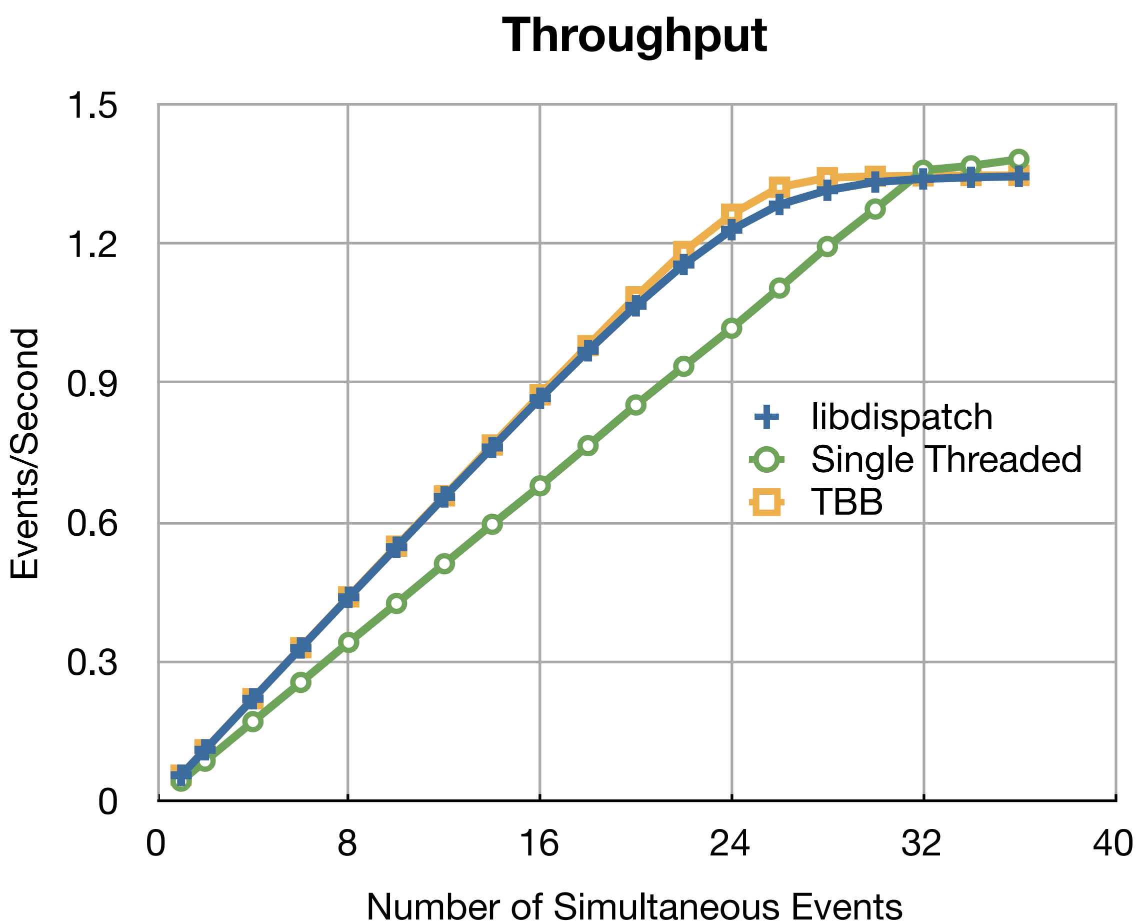
Test System

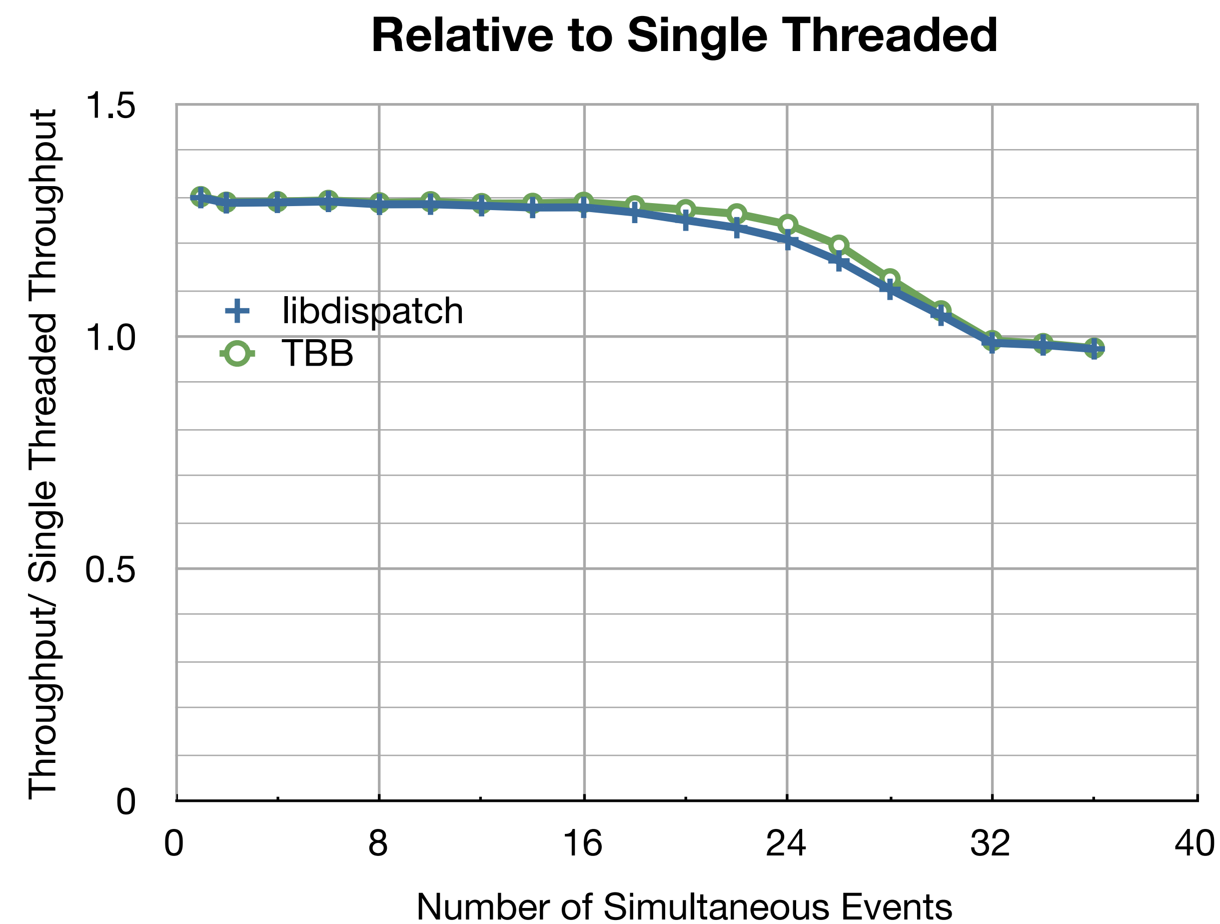
AMD Opteron(tm) Processor 6128

32 Cores

64GB memory

Scaling: 32 Cores





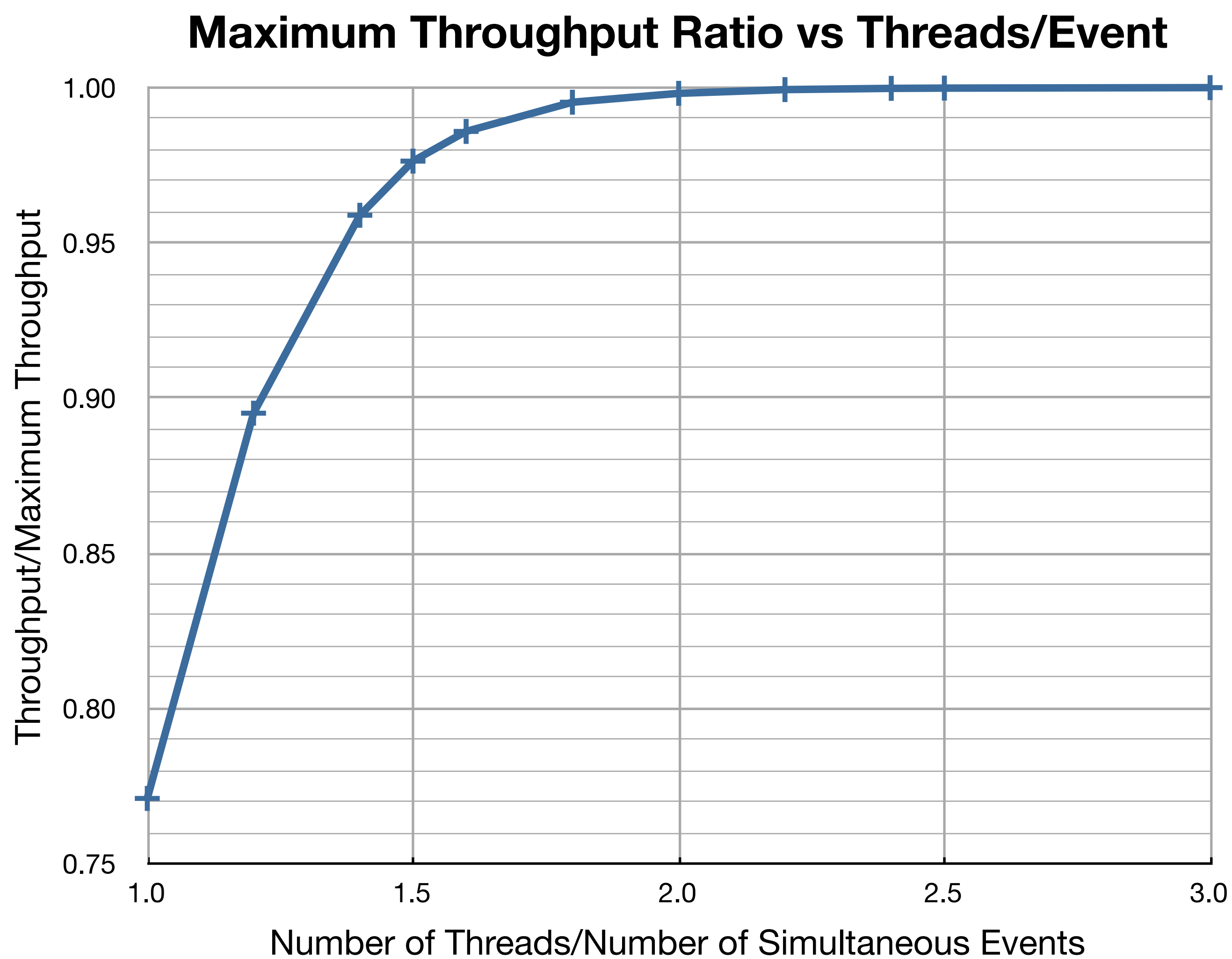
Producers fully use a core by doing a numeric integration

calibrated how many seconds per integration step

TBB slightly outperforms libdispatch

TBB was told to use 32 threads for all the above measurements

Threads per Event



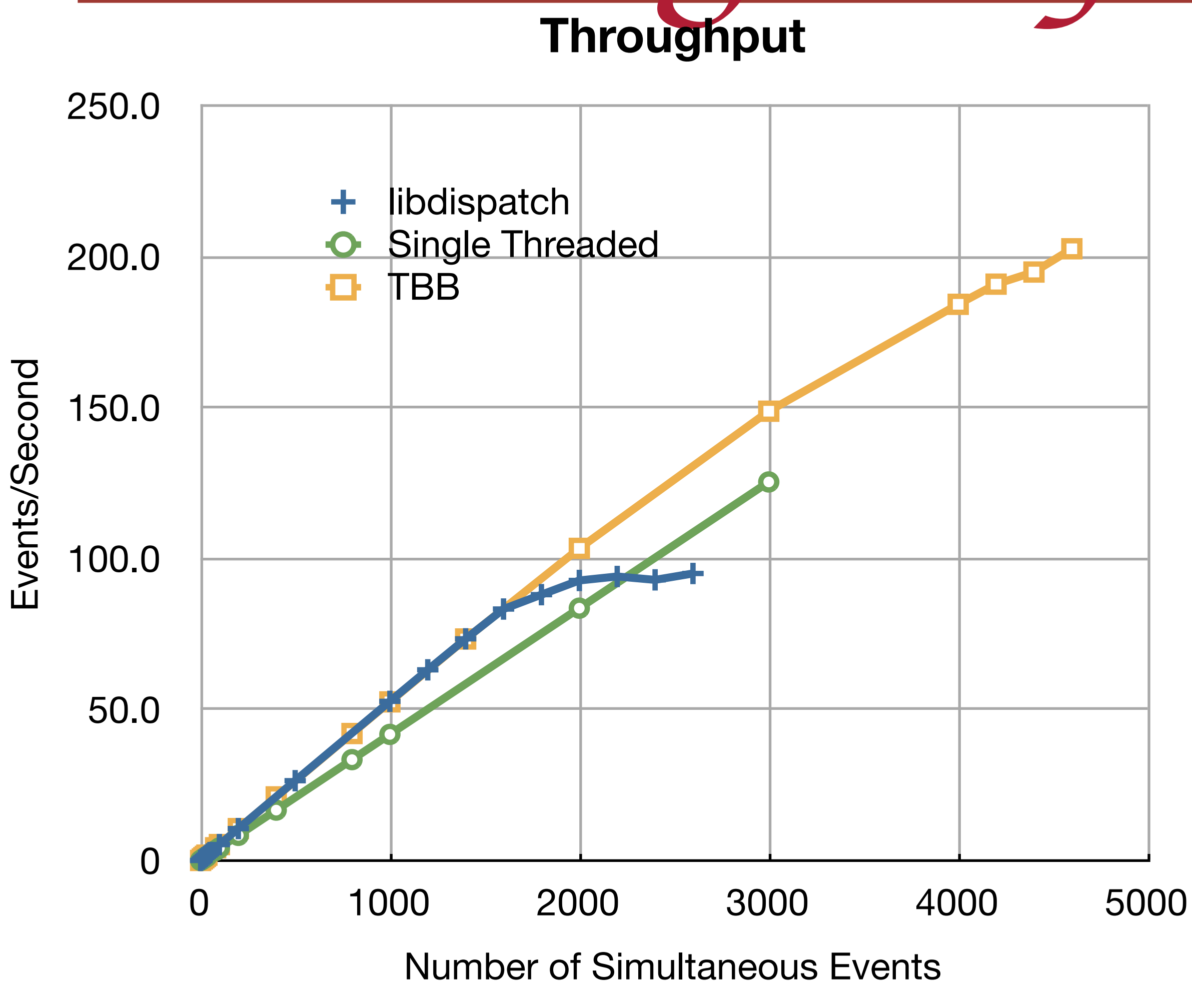
Measurement

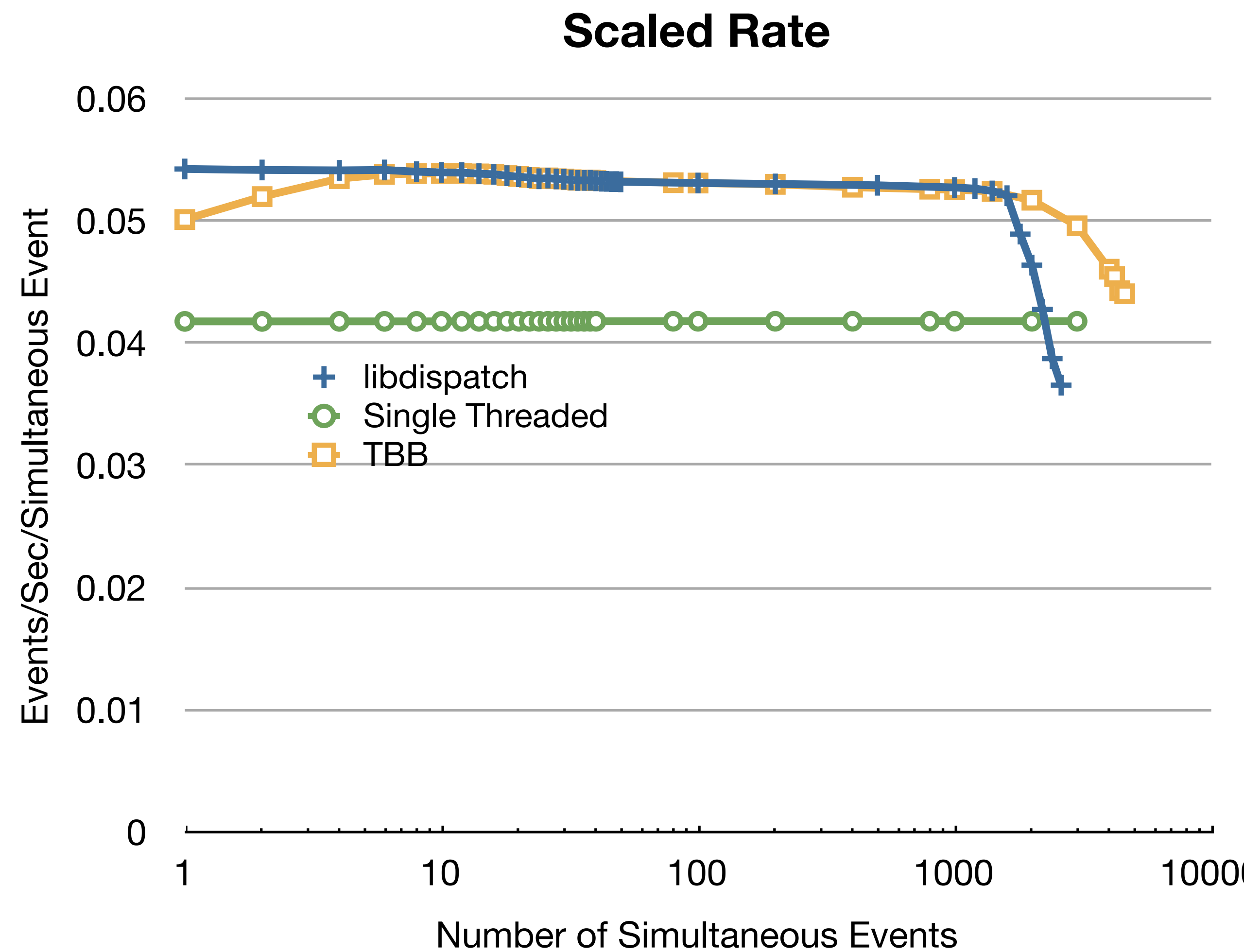
Keep number of simultaneous events constant (10)
Vary number of threads (10-30)

1.5 Threads/Event gives 97.6% of max

2 Threads/Event gives 99.8% of max

Scaling: Infinite Cores





All Producers are calling usleep

TBB stops scaling around 2000 simultaneous events (se)

- Is using 1.3 threads/simultaneous event
- Lowering threads/simultaneous events improves scaling limit slightly

libdispatch hits scaling limit around 1600 se

Single threaded framework hits memory limit at 3000 se

Max Speed Test

Simple Configuration

1 Filter that requests 3 products but doesn't wait
3 Producers who do not wait
Only 1 simultaneous event

Demo Framework	Max Throughput	Relative Slowness
Single Threaded	1,460,000	1
TBB	375,000	3.9
libdispatch	3,600	406

NOTE: libdispatch was built without optimization

Synchronous Event::get

All previous tests were with prefetching

Before running a module would asynchronously prefetch the data it needed

Need to support synchronous Event::get calls

Request from inside a module to get data that isn't yet in the Event
Needed to support legacy modules
Useful in the future for data that is only needed for some Events
Synchronous get is similar to how threading inside a module would work

libdispatch implementation failed when used for RECO config

Each synchronous Event::get caused working thread to block
libdispatch noticed thread block and spawned new threads
HOWEVER, there appears to be a limit to # of threads libdispatch will spawn
When reach limit, the job is then blocked waiting for work that never runs

TBB implementation works well

tbb::task::wait_for_all doesn't block threads, instead it processes waiting tasks
tbb uses 'last in/first out' for tasks
So Event::get task runs right after it is requested rather than waiting in task list

Conclusion

TBB is easy to build and is designed for C++

libdispatch is difficult to build and requires clang

Apple recently changed libdispatch to require more Objective-C entanglement

TBB has lower CPU overhead than libdispatch

Should allow it to be used with smaller units of work

TBB has primitive synchronization mechanisms

However, more useful ones can be constructed from the primitive ones

libdispatch synch mechanism has a limit which when reached freezes the job

TBB requires that the # of threads be set at the beginning

Workflow management is requiring that jobs only use specified # of cores

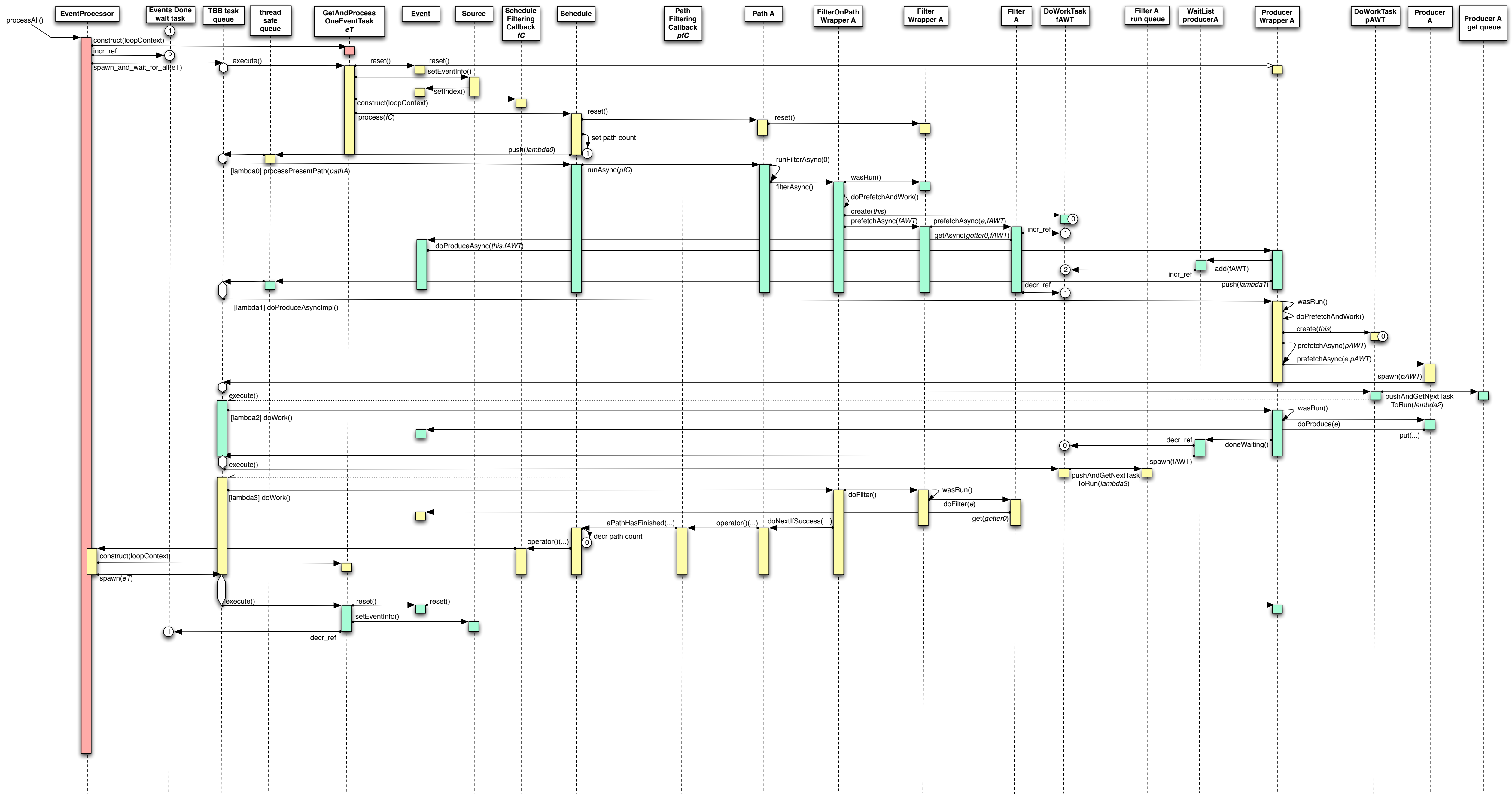
But I/O activities that must wait will need their own threads

Must avoid undersubscribing threads since they will busy wait

libdispatch gives no control over how many threads are to be used

TBB is a better match for CMS than libdispatch

Backup Slides



Timing Diagram

One Filter
One Producer
One Simultaneous Event