Using libdispatch while reading with ROOT

Christopher Jones
FNAL

Outline

Goal & Strategy

At The October Concurrency meeting at CERN

I presented a demo of a highly threaded framework

http://dl.dropbox.com/u/11356841/Threaded_Framework_Discussion.pdf

Attendees were challenged to get ROOT I/O working with their thread model

Goal

Maximize CPU utilization while minimizing memory

Strategy

Run as many events in parallel as memory allows

Break up actions within one event into parallelizable chunks

- utilizes more cores if memory becomes constrained
- in case of cross-event synchronization points, allows remaining events to process faster

Share as much memory across events as possible

- E.g. Input buffers

Threading Models

Test Threading Model

Uses libdispatch

Developed by Apple Inc

Port is available for Linux and Windows

Task Queue based system

Task is a function plus context

Context can be any data you want

Task is then placed in a light weight queue

System guarantees that cores are not oversubscribed

Global Concurrent Queue

One per process (in Mac OS X one per machine)
Task placed here will be pulled in FIFO order
Multiple tasks can be run simultaneously (based on # of cores)

Private Sequential Queues

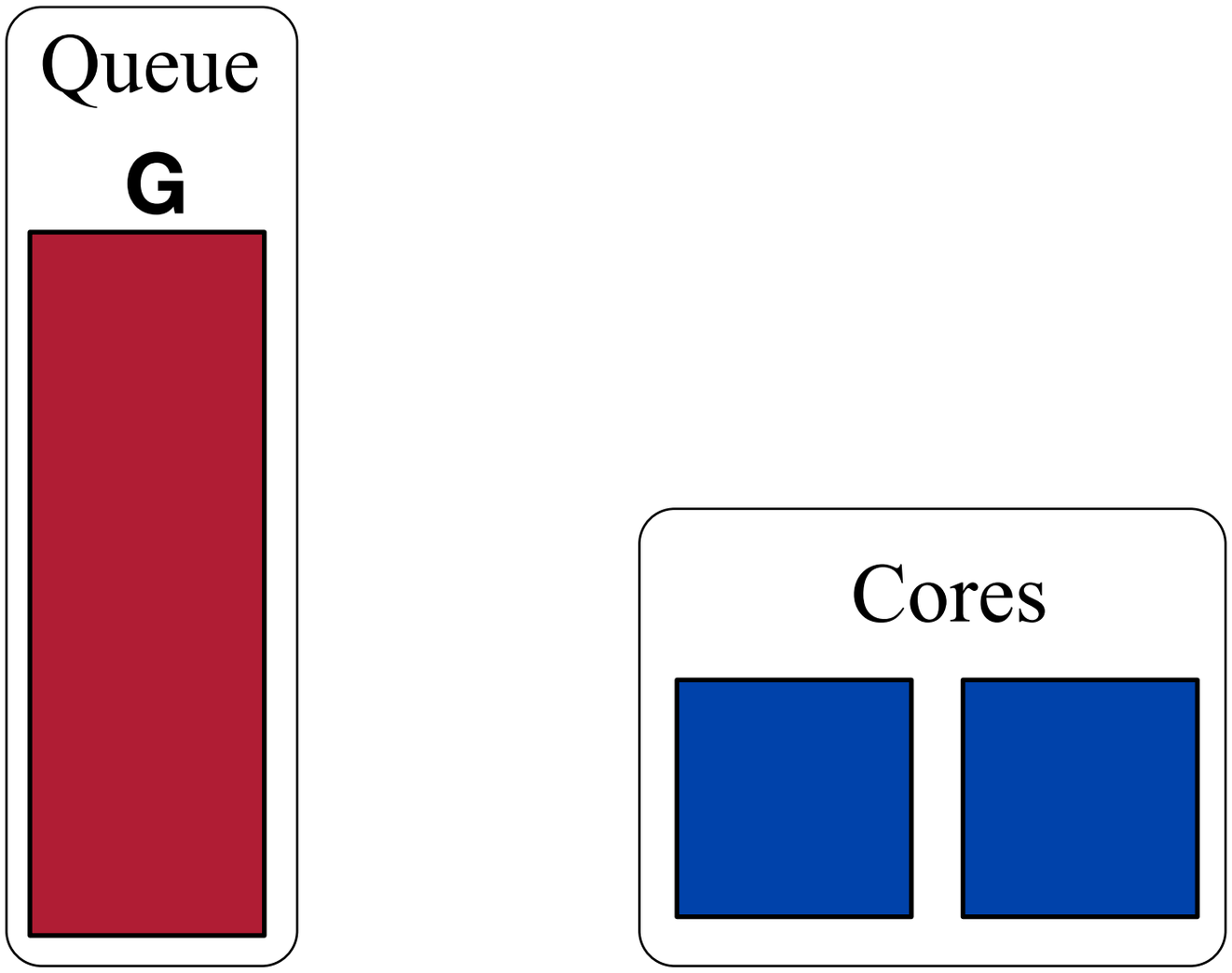
Lightweight (memory/CPU) queue of tasks
Can handle thousands of sequential queues per process
Task placed here will be pulled in FIFO order
Only one task from a given queue will be run at a time
Guarantees sequential behavior without having to use thread primitives

Task Groups

Can group multiple tasks into a group
Can register a different task to run once all tasks in a group finish

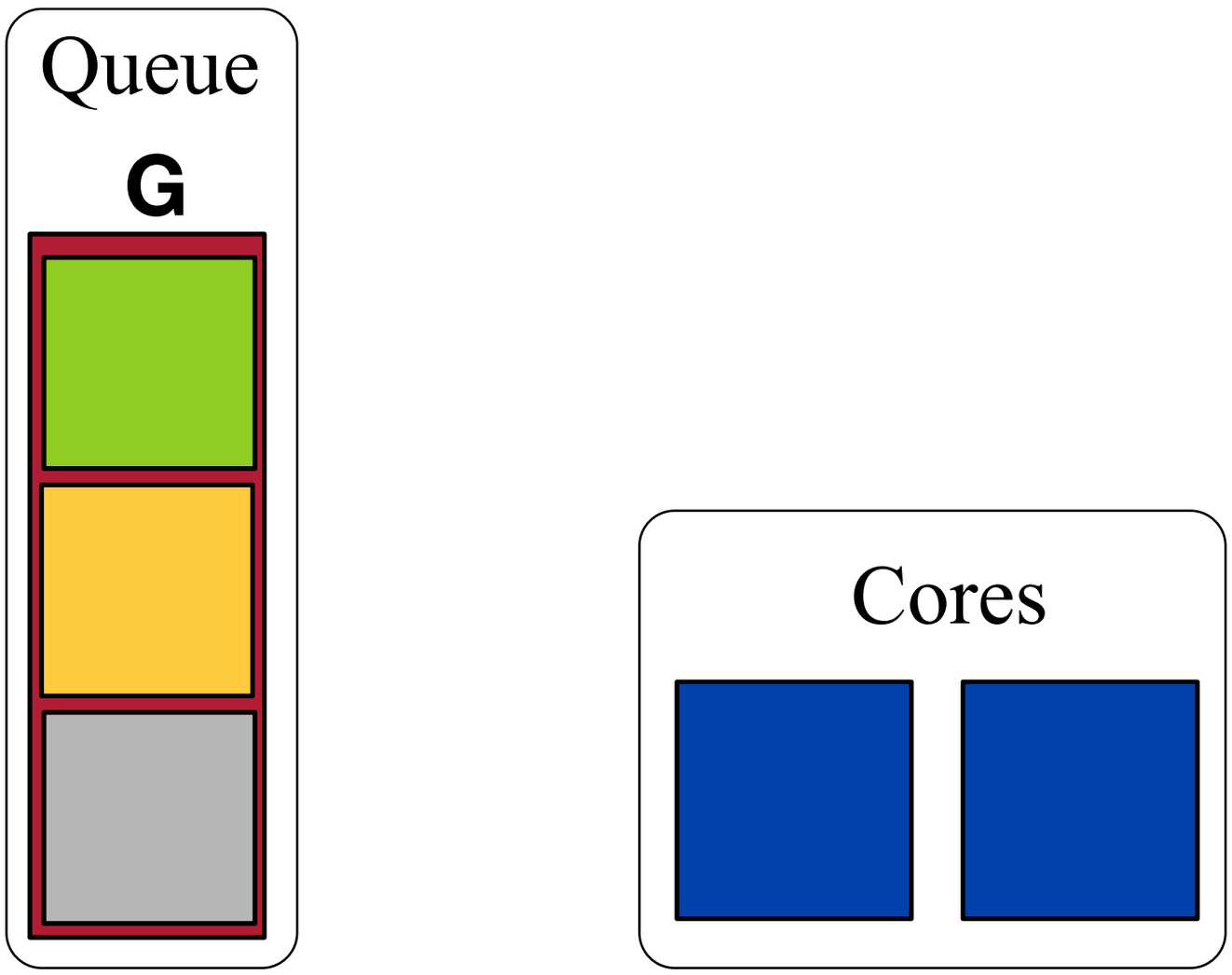
Global Concurrent Queue

Tasks pulled in order and run concurrently



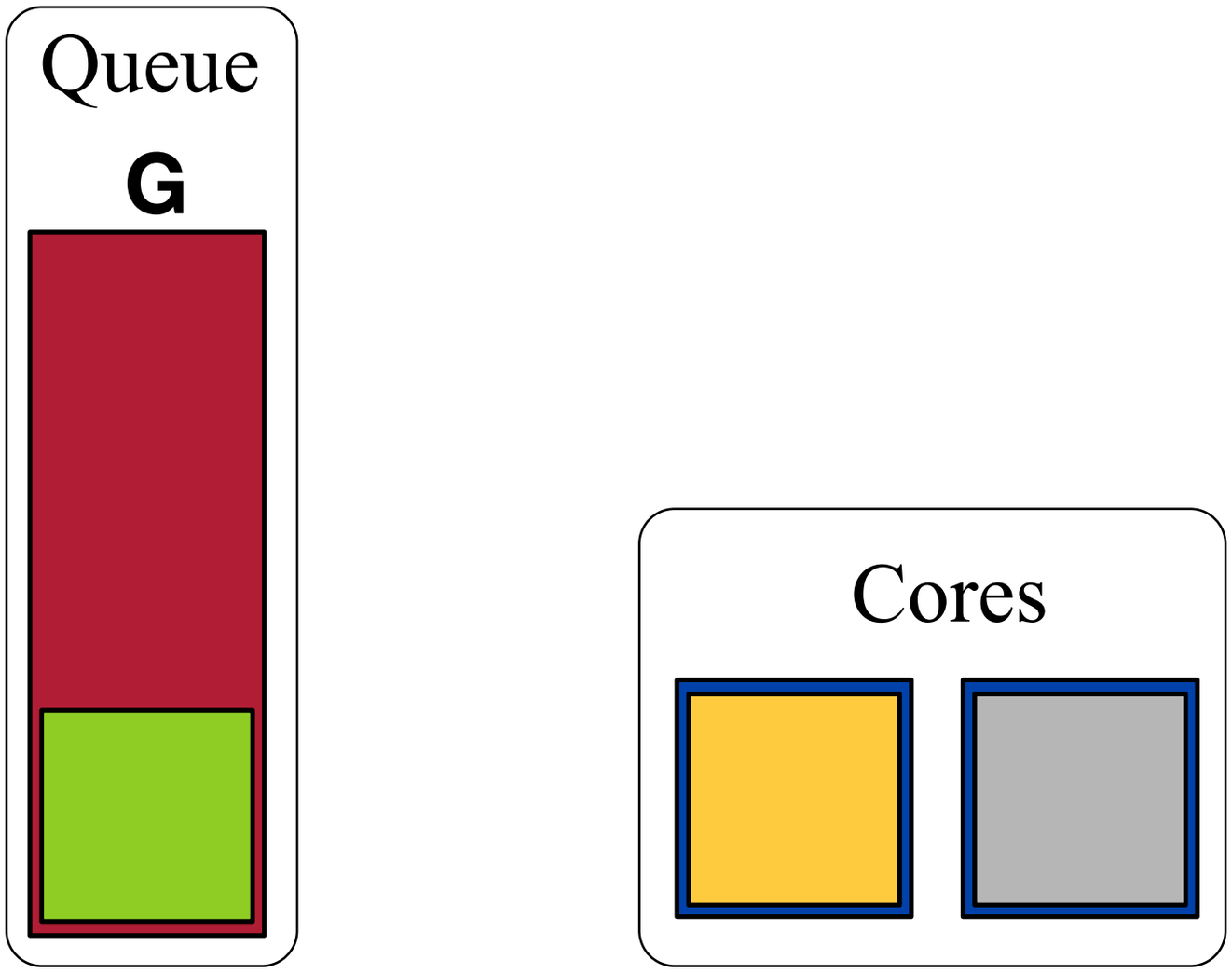
Global Concurrent Queue

Tasks pulled in order and run concurrently



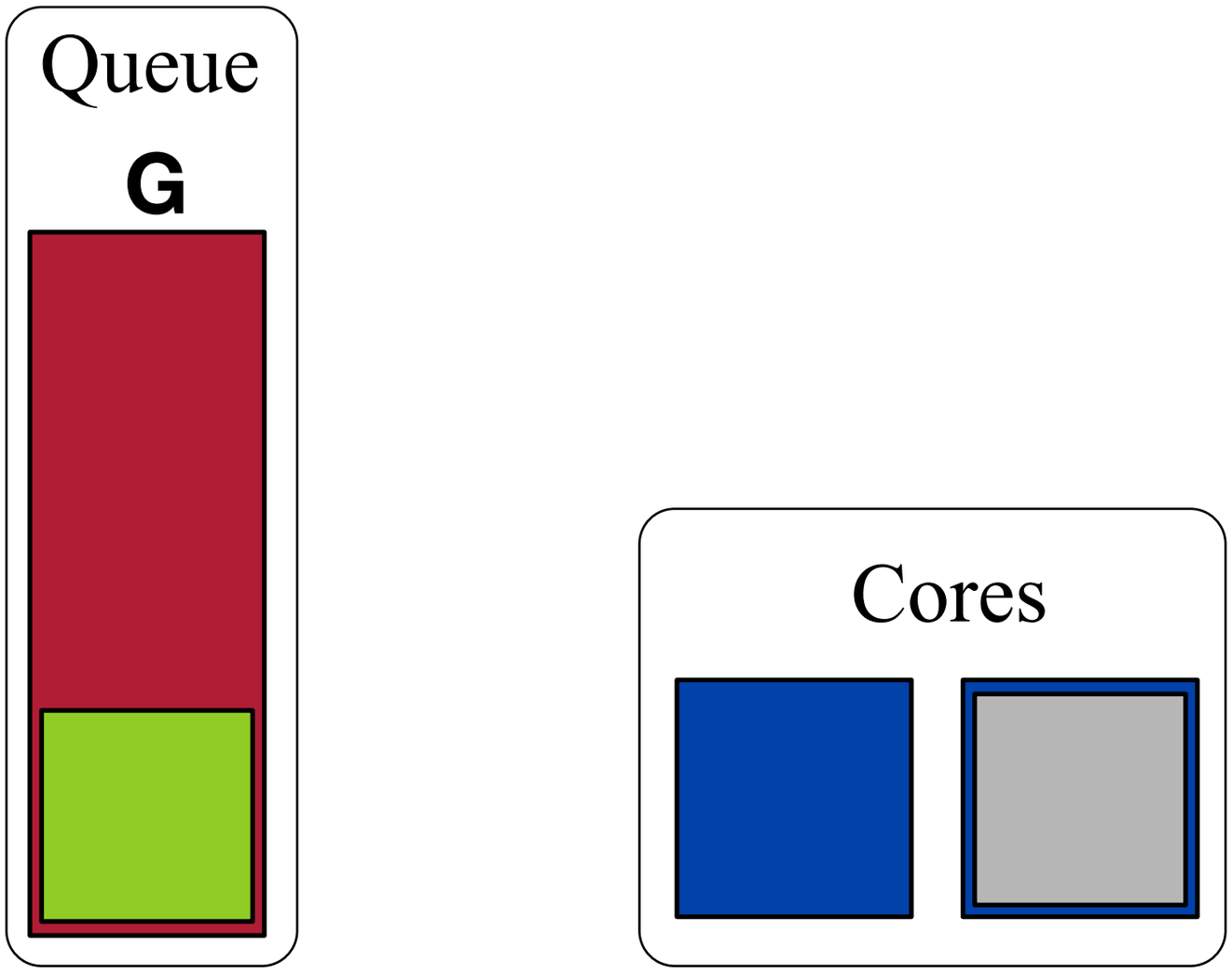
Global Concurrent Queue

Tasks pulled in order and run concurrently



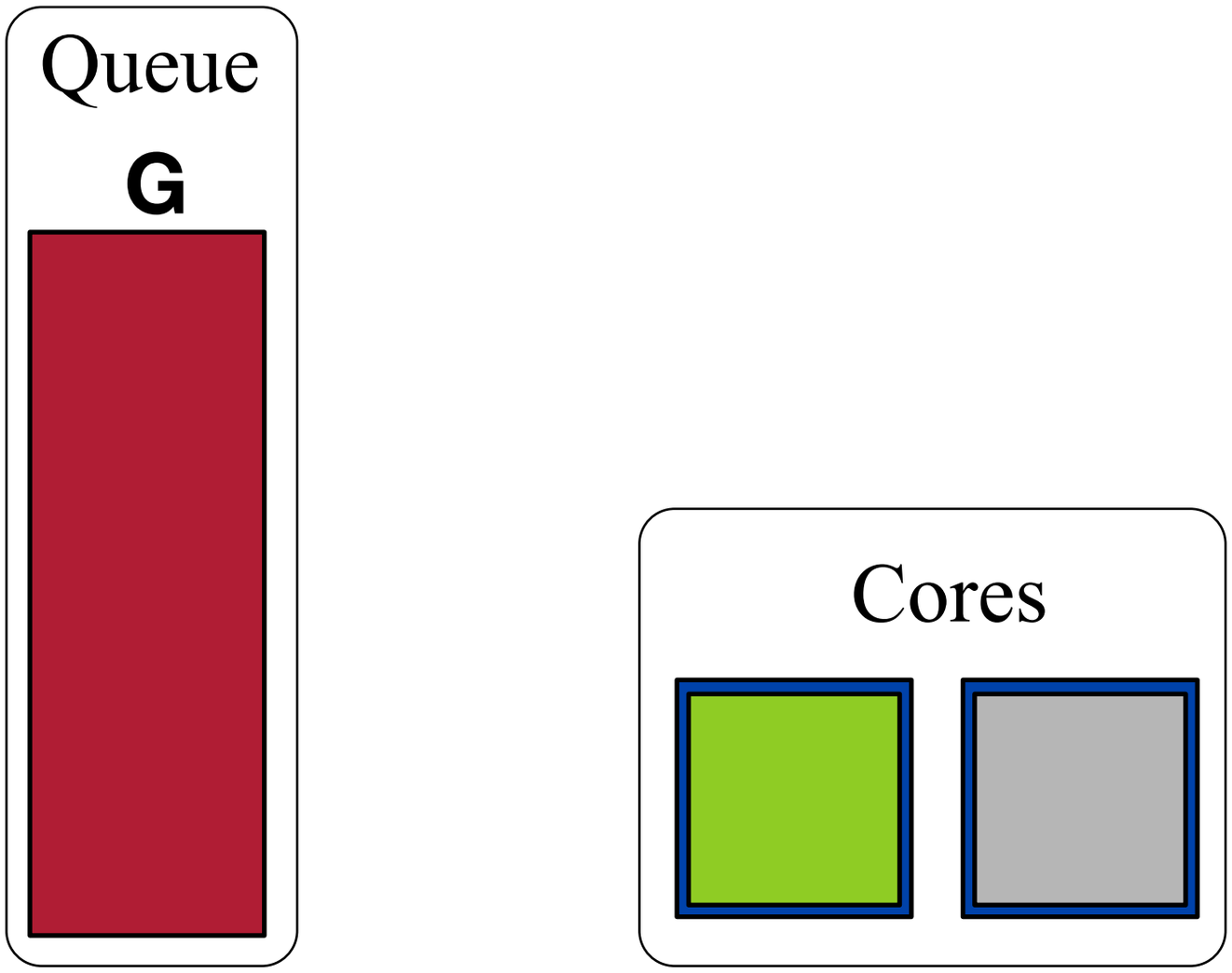
Global Concurrent Queue

Tasks pulled in order and run concurrently



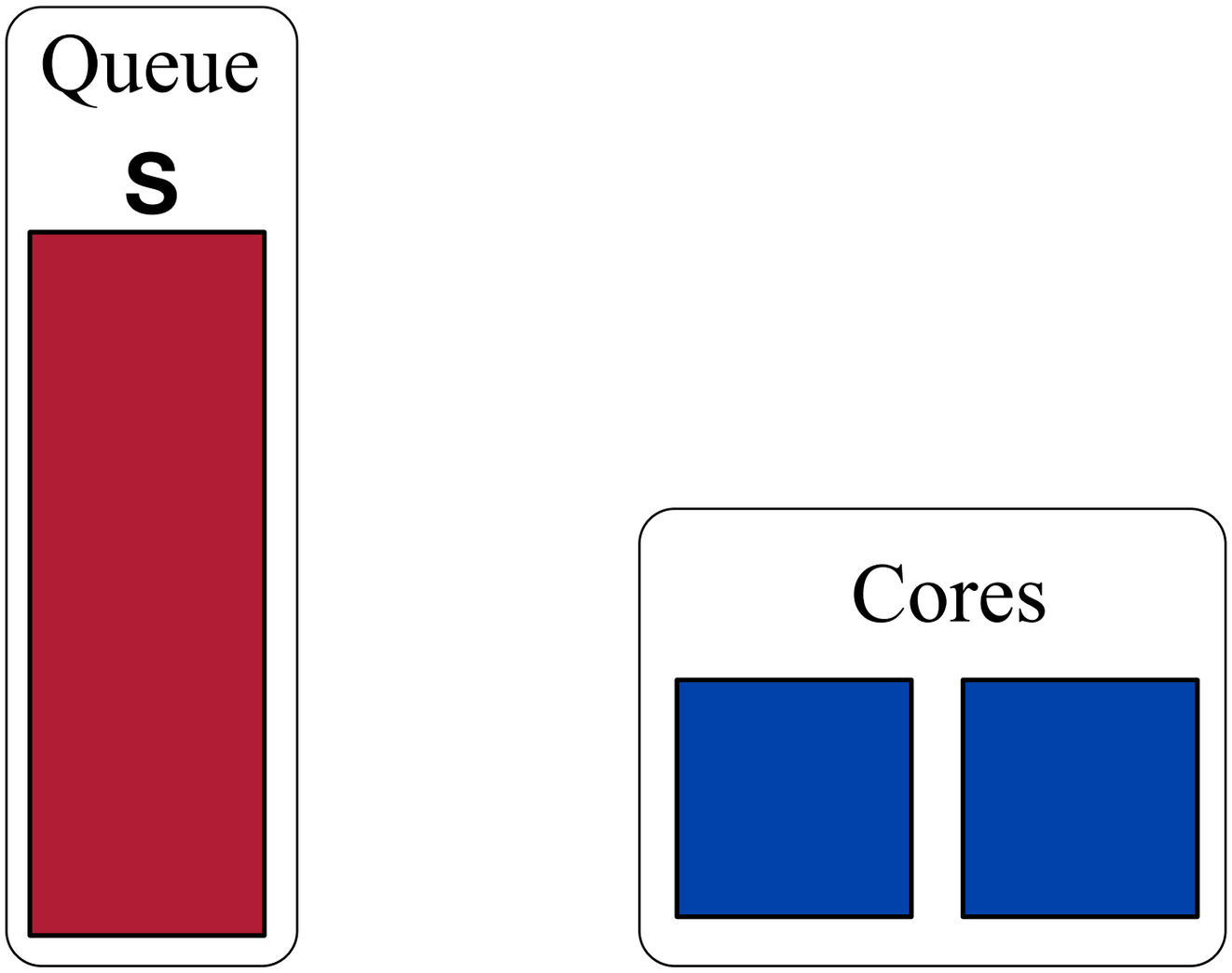
Global Concurrent Queue

Tasks pulled in order and run concurrently



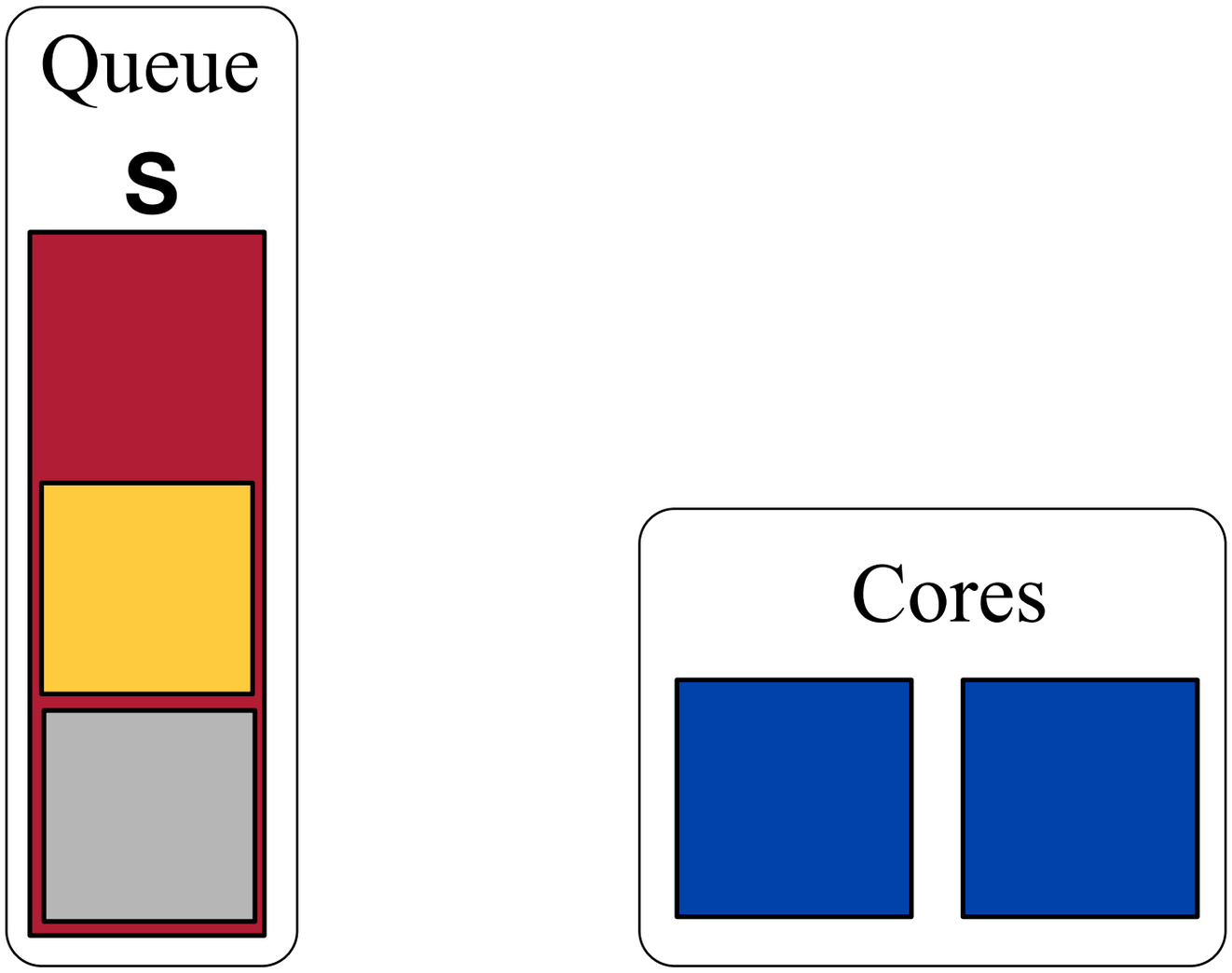
Private Serial Queue

Tasks pulled in order with only one run at a time



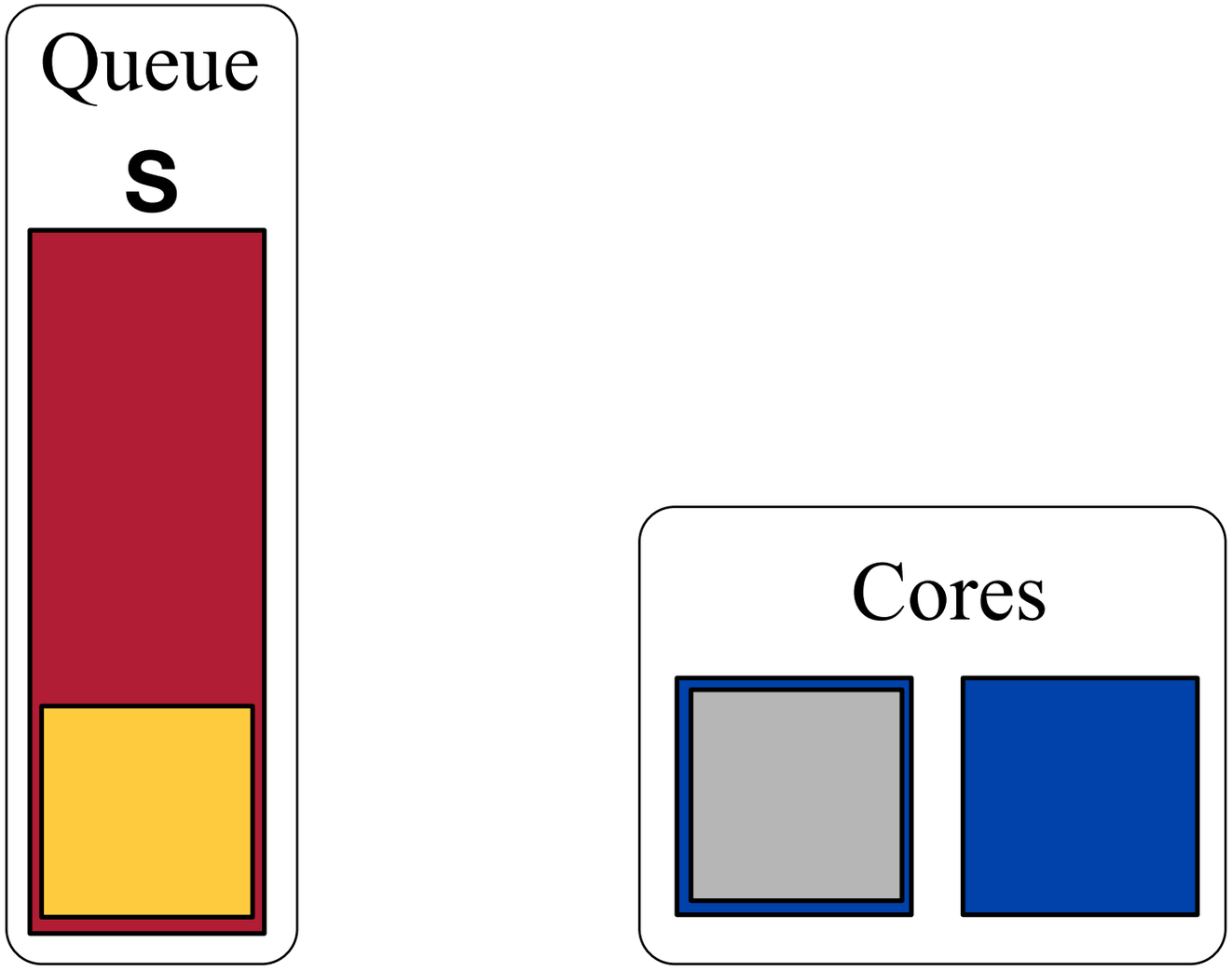
Private Serial Queue

Tasks pulled in order with only one run at a time



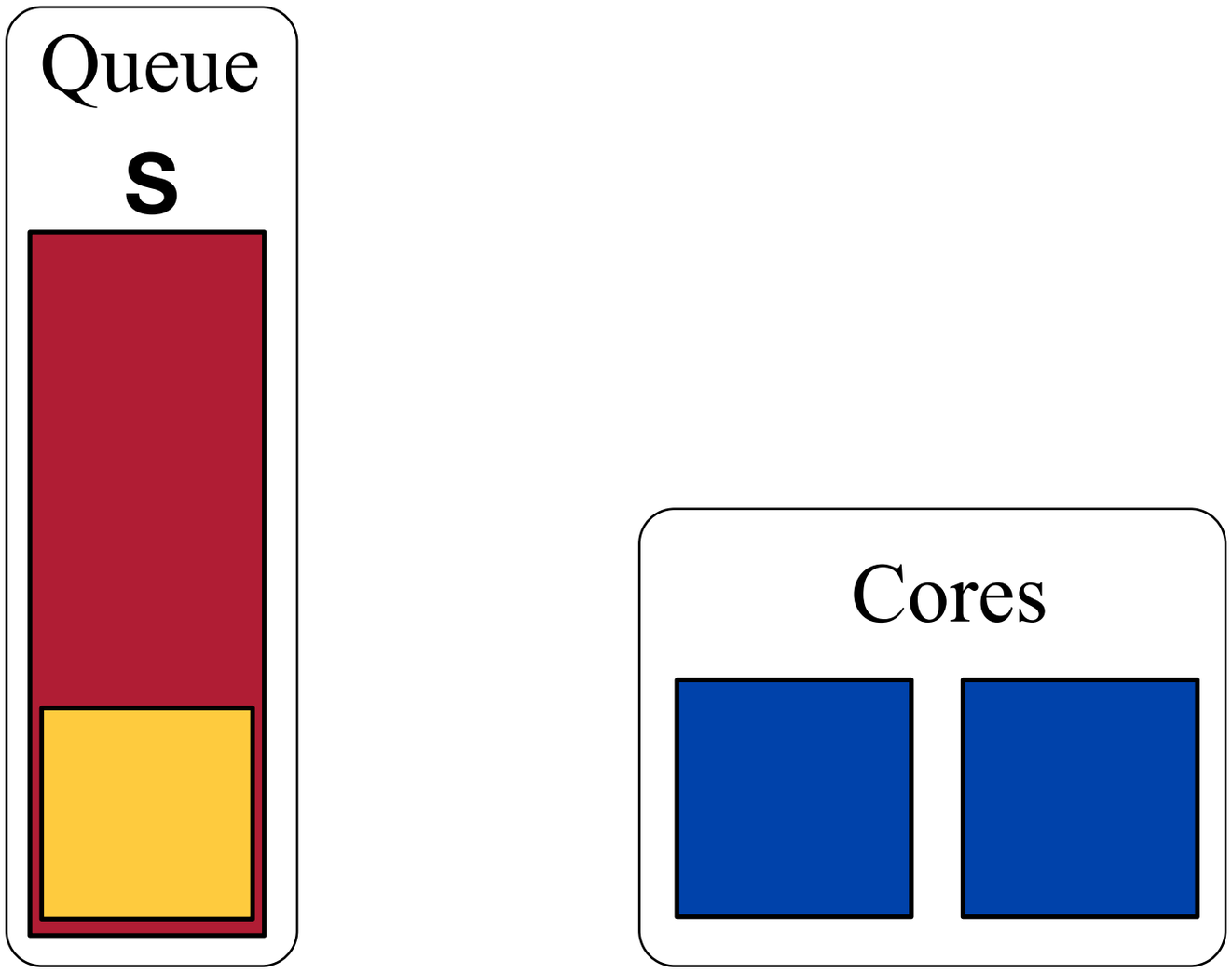
Private Serial Queue

Tasks pulled in order with only one run at a time



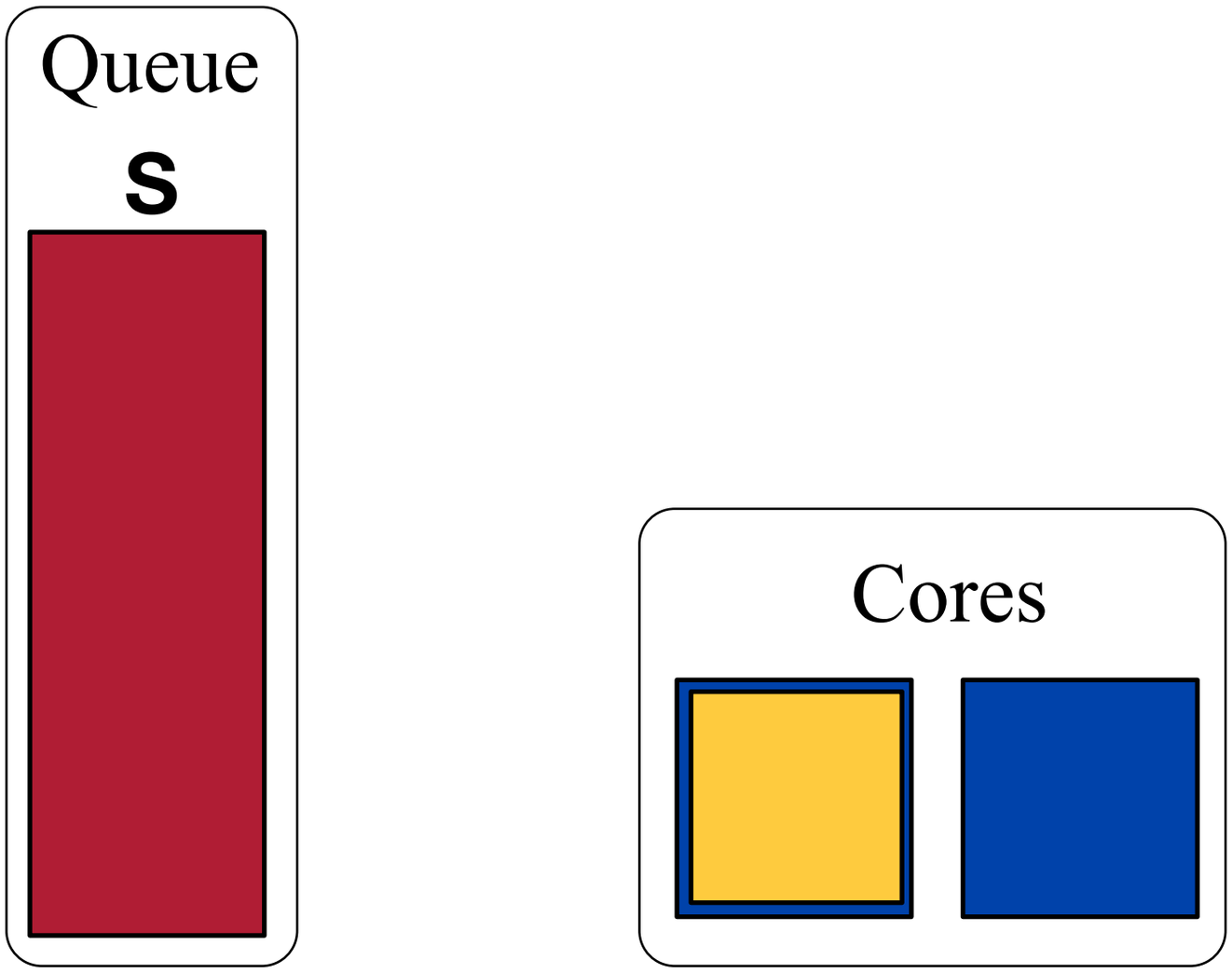
Private Serial Queue

Tasks pulled in order with only one run at a time



Private Serial Queue

Tasks pulled in order with only one run at a time



ROOT's Thread Model

Global Mutexes

Used to protect gROOT, CINT internals, etc

Used in memory handling in TStorage global functions

Thread Local Storage

Used TClass primarily for I/O layer

TClass holds one TVirtualCollectionProxy

Pointer to collection changed for each instance of that collection seen in a TBranch

TClass holds one TClassStreamer

Allows user to override streaming -> not used by CMS

NOTE: assumes thread accessing TClass function is the one that will use the value retrieved

TThread::initialize()

Turns on thread local storage and TStorage mutex

Test Case

Test Case

Read a ROOT file and process 4 branches each event

File

TTbar reconstruction with no pileup
    reasonably large numbers of objects
split level 1
7700 events
900 events per cluster
    8 clusters in the file

Branches read

CaloTowers
Tracks
2 types of jets

Test Machine

Macbook Pro
2.8GHz Intel Core 2 Duo
4 GB 1067 MHz memory

Job Structure

Open TFile

Setup TBranches

Guarantee that a TBranch is assigned to one and only one Event

Setup TTreeCache

Tell it about all 4 branches

Stop learning phase

Process Events

Read all 4 branches for each event

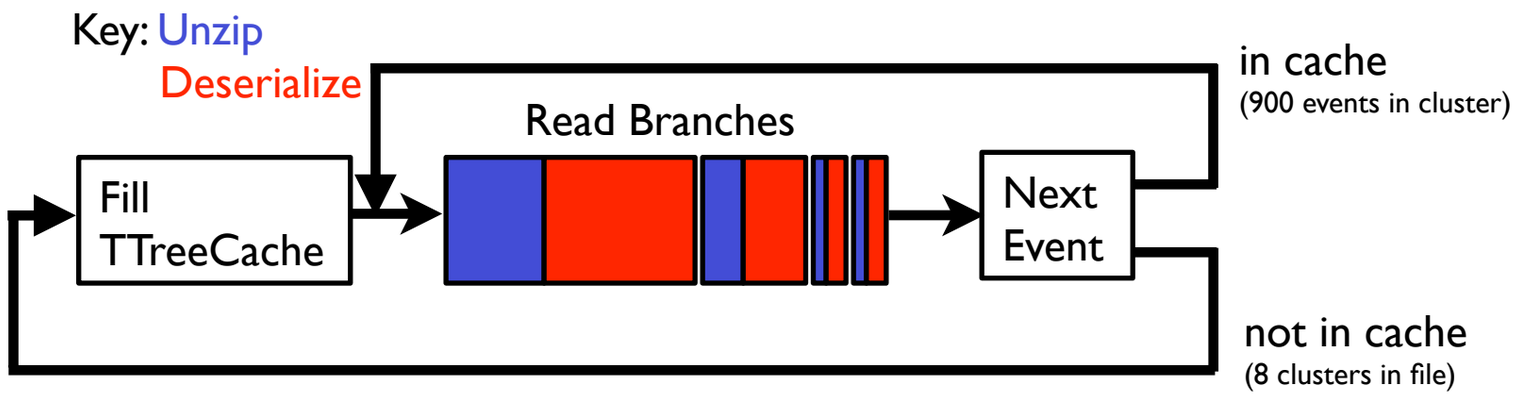
Wait until all branches have finished reading before going to next event

All time measurements are only for event processing time

Parallelizing within One Event

Single Threaded

Event Logic



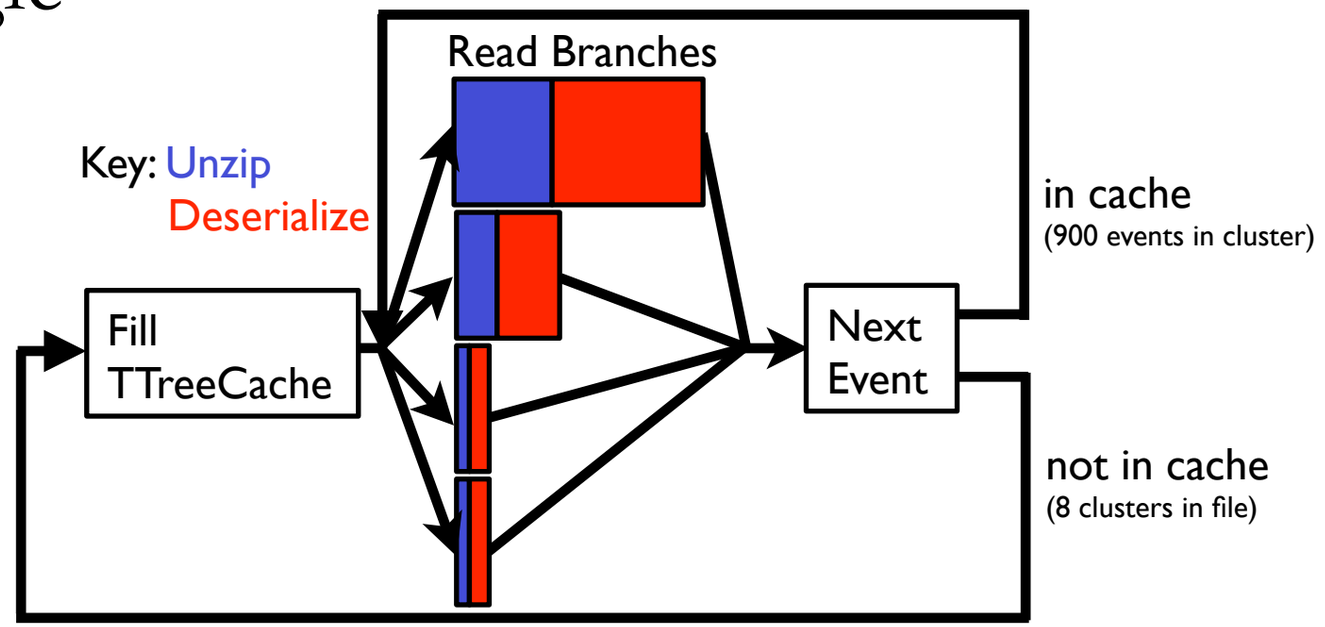
Total Event Times

Fill TTreeCache	0.24s
CaloTowers branch	6.44s
Tracks branch	2.08s
Jets1 branch	0.52s
Jets2 branch	0.31s
Total	**9.60s**

Best possible parallel branch reading: 6.68s

Multi Threaded

Event Logic



Parallel handling of branches

Use group to wait for all branches of an event to finish before continuing

Modifications to ROOT

TVirtualCollectionProxy

ROOT makes only one TVirtualCollectionProxy per class type

- holds methods used to interact with a collection
- also holds temporary pointer to a collection instance in order to manipulate the collection

No longer holding temporary pointer to a collection instance

Use of the proxy replaced with a new class with instance on the stack

- New class holds a pointer to TVirtualCollectionProxy and the collection instance

Buffer management

Compressed buffer for reading was shared by all TBranches in a TTree

Moved so each top level TBranch has own compressed buffer

- assumes that reading a TBranch is a sequential operation

TBasket::ReadBasketBuffers

If data is in TTreeCache's buffer, call directly to cache without changing TFile's state

Profiling:Instruments

Instruments is Apple's performance tool

Presents information in time order

Allows filtering of information by time range

System Trace

Records all system calls

I/O, interrupt handling, locks, etc.

Records all virtual memory activities

Zero page fills, Copy on Write, etc

Records thread state transitions

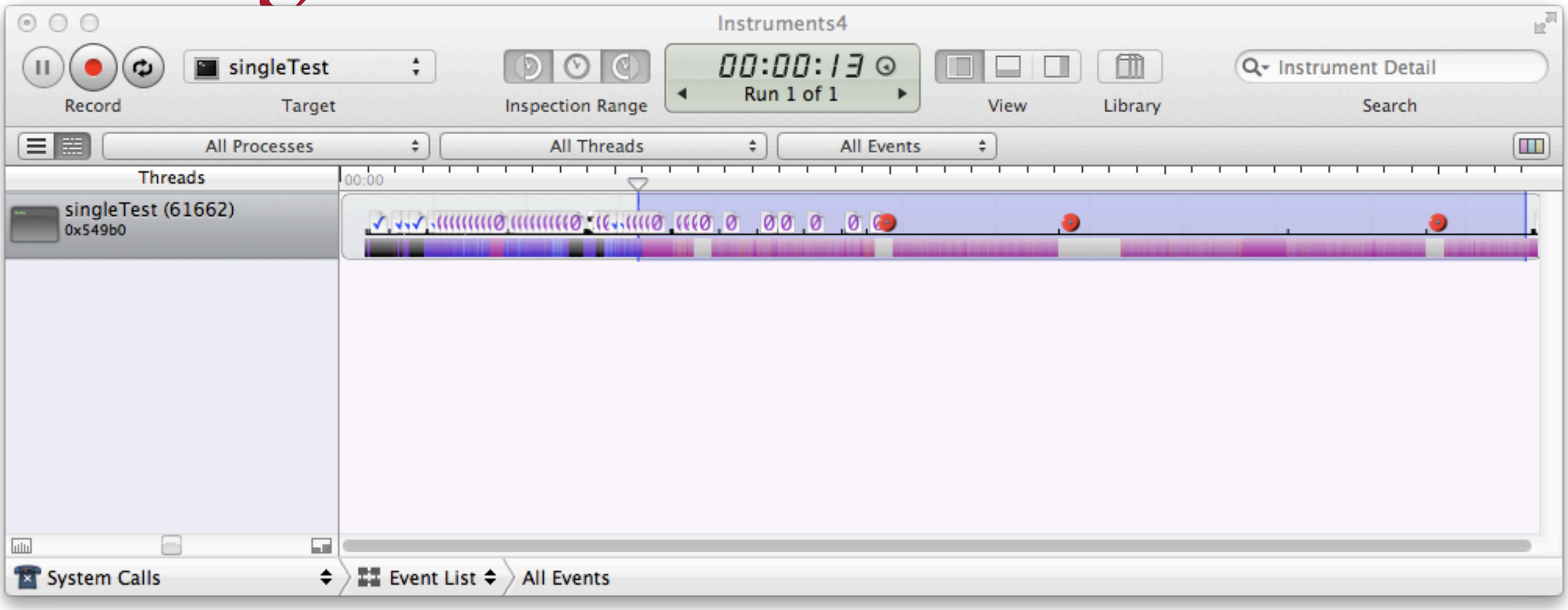
context switches, blocking, running, supervisor

Time Profiler

Samples executable and kernel at regular time intervals

Snapshots stack traces for all threads

Single Threaded



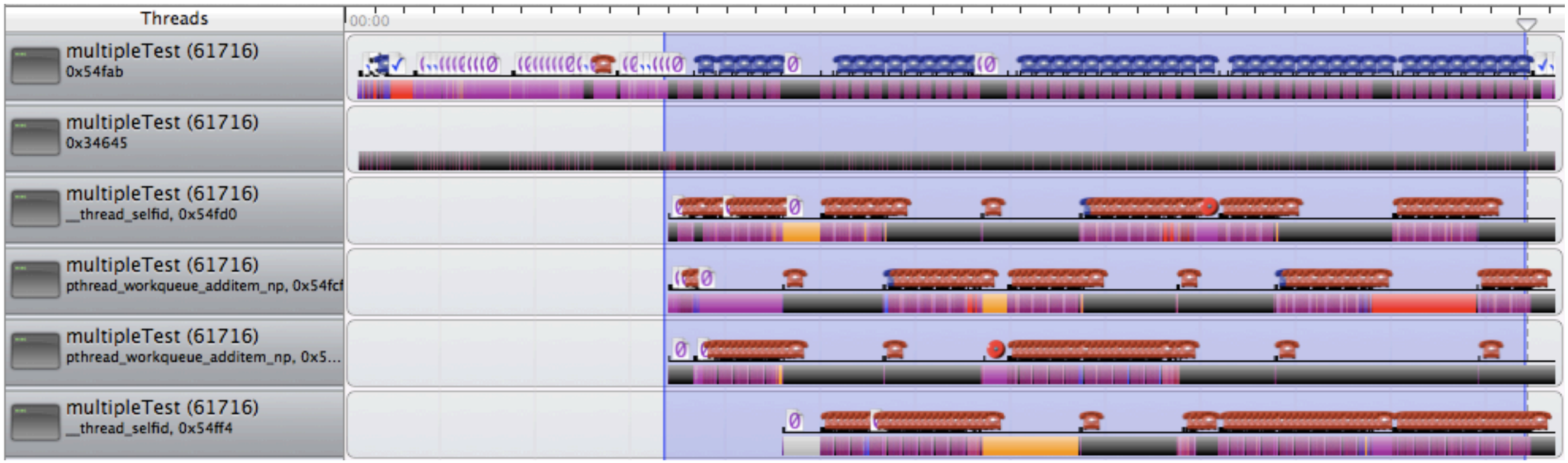
Main View Area

X axis shows events in time order
Y axis shows threads
Thread state is given by color of bar
System activity on thread given by icon

Thread State Key

- blocking
- user
- supervisor
- preempted
- interrupt

Multi-Threaded 1



Event loop on main thread

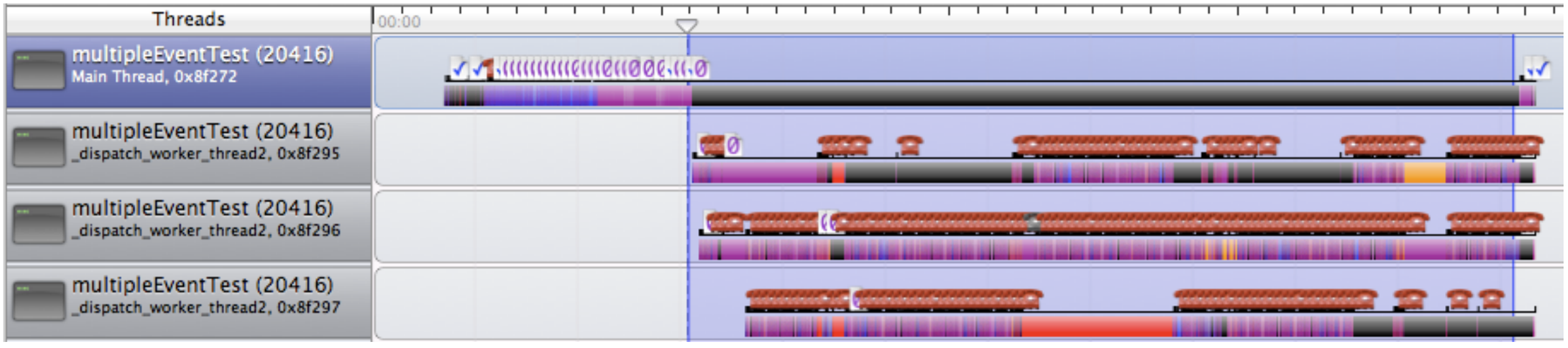
Can see 'blue telephones' which show the wait on a dispatch group

Time: 8.21s

1.17x faster than single threaded

81% of theoretical max

Multi-thread 2



Main loop waits for all events to be processed

Going to next event is handled by calling a function placed in a serial queue

On finish of event a new task is added to the global queue

Time:8.13s

1.18x faster than single threaded
82% of theoretical max

Use of queue rather than waiting in main thread was faster

Multiple Events

Two TFiles

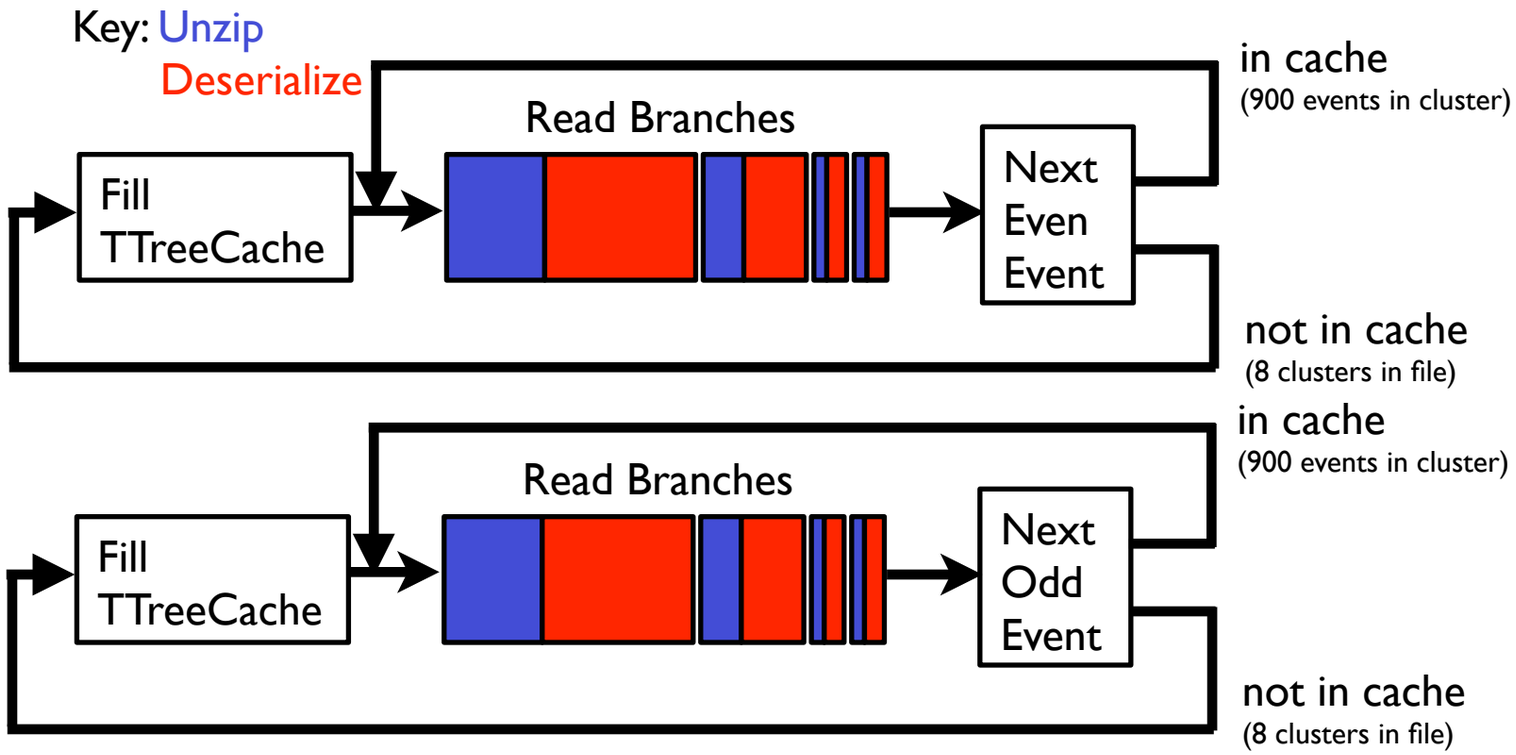
Open 2 TFiles on the same file

Avoids the need for synchronization

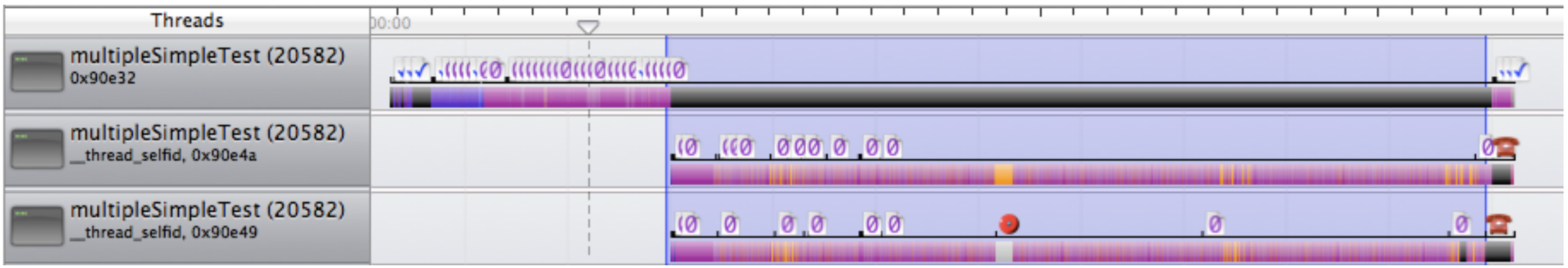
Run two tasks on the global queue

Each task has its own TFile

One task does even events and the other task the odd events



Two TFiles



Notice how few system calls (telephones)

Time:6.9s

1.39x faster than single threaded

NOTE: by avoiding TThread::Initialize() I save 7%

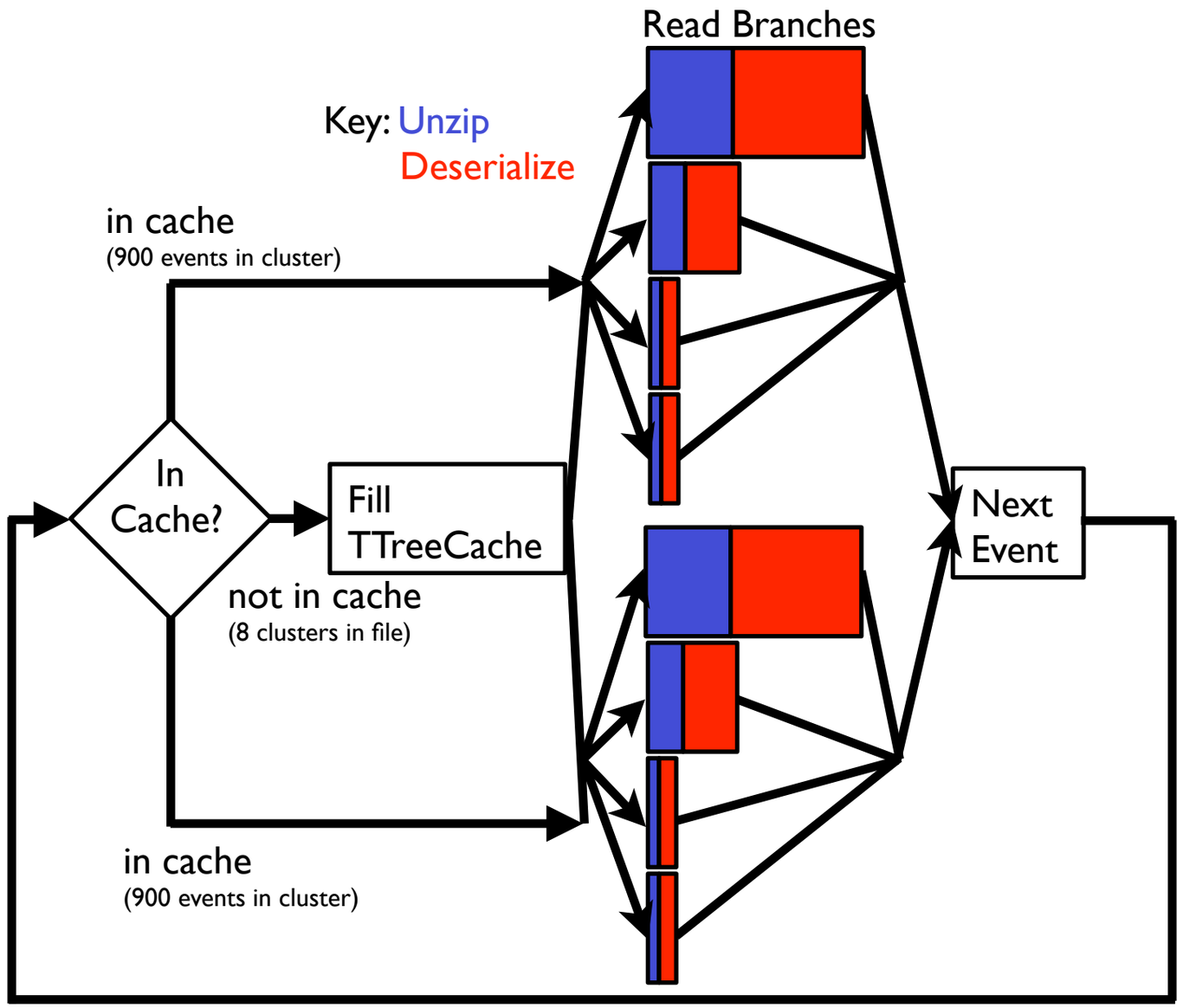
Shared TTreeCache

Simple case where branches wait for the cache to fill

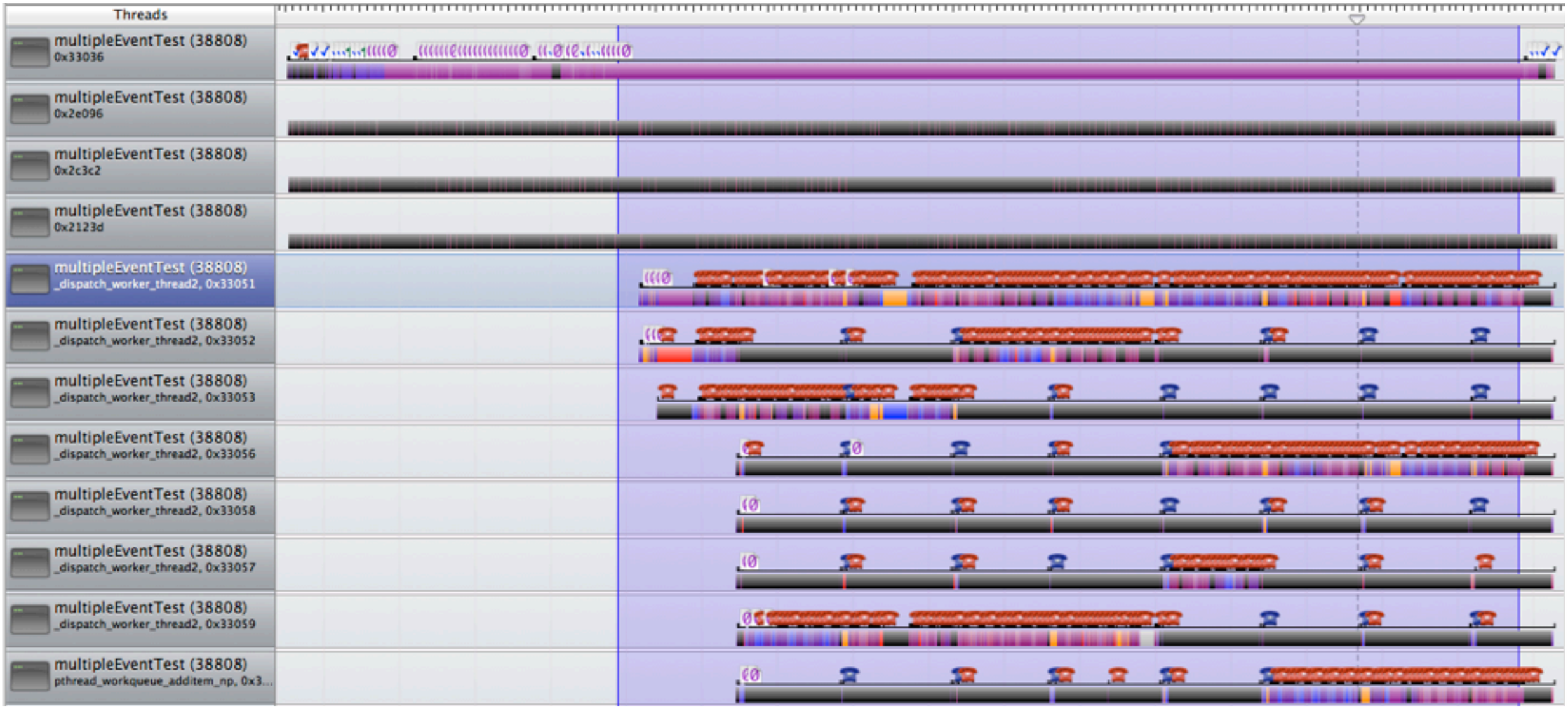
Each event has own TFile, TTree and TBranches

TFiles have been changed so they can share one TTreeCache

Access to check on availability in TTreeCache is done through a queue



Shared TTreeCache



TBasket read blocks on TTreeCache inspection

Notice the 7 synch lines from the TTreeCache fills

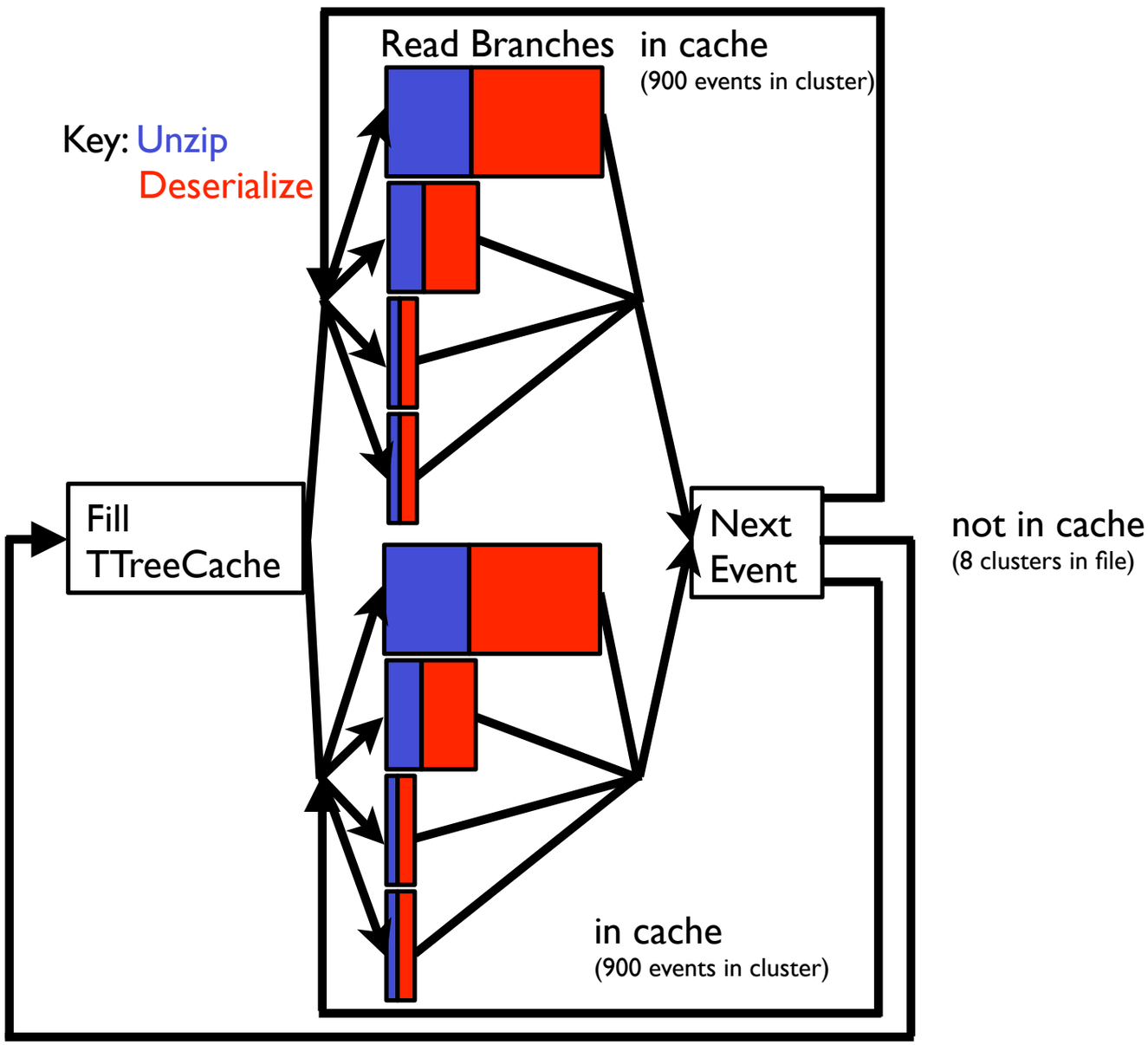
All the extra threads are because when one thread waits, libdispatch starts another thread since it doesn't know the next task will also have to wait

Shared Cache No Waiting

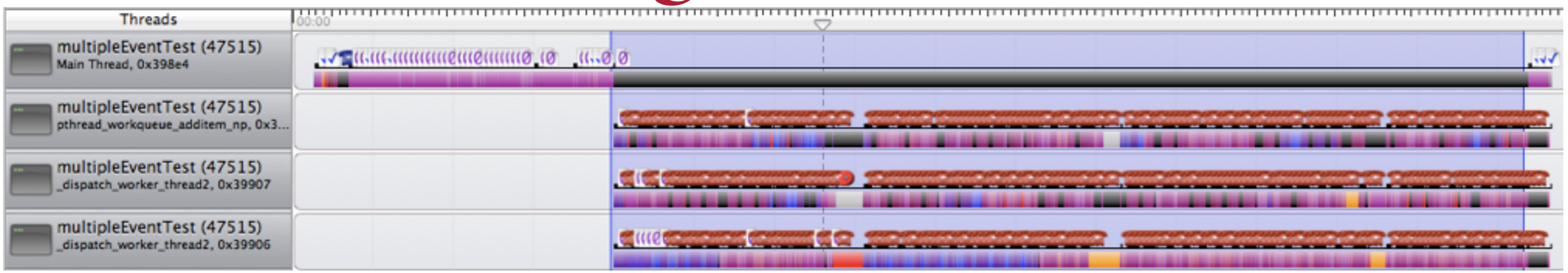
Check when looking for next event to see if in cache

If in cache don't do any wait

If not in cache, hold until all tasks reading cache are done, fill cache and then start tasks for next events



No Waiting



Much fewer threads

Time:6.7s

1.43x faster than single threaded

Why So Slow?

Look at Time Profiles

	Single Threaded	Two TFiles	Shared Cache
Unzip Baskets	2843.0ms 29.8%	5755.0ms 43.5%	5839.0ms 44.6%
Deserialize	6308.0ms 66.2%	6603.0ms 49.9%	6587.0ms 50.3%

Duplicating work on each thread

Events are each unzipping the same baskets

Change 2 TFile case

Have one thread read 1st half of file and other thread read 2nd half of file
Time:5.06s which is 1.90x faster than single threaded

Conclusion

Nice if ROOT was more thread friendly

- Removal of global caching in the I/O layer
- Also helpful for forking
  - I'd be willing to help with that

Good performance tools are a must

- Time profiling
- Synchronizations between threads

ROOT file structure makes event parallelization challenging

- Data not grouped by event
- TTreeCache and TBaskets correlates data across events

Important to separate serial and parallel parts

- Blocking other threads while only one updates a cache 'lazily' causes libdispatch to spawn more threads

Backup Slides

Exploration

I spent a couple of weeks building a demo threaded system

Features

Process multiple events simultaneously
- Can set maximum # based on available memory

Within one event can run paths in parallel
- A path is a series of filters that decide to keep or reject an event

Producers are run the first time their data is requested
- Multiple Producers within one Event can run at the same time

Supports thread unsafe modules (or parts of modules)
- Would allow transition

Supports threading internally to a module using same thread pool

Supports using the same module instance for all events
- Minimizes memory use

Makes sure all events in a Lumi Block are processed before going to next block
- When hit lumi end, remaining events get more cores to process so go faster

Makes sure all events in an IOV are processed before going to next block
- Minimizes memory used by EventSetup
- When hit IOV change, remaining events get more cores to process so go faster