Study of a Fine Grained Threaded Framework Design

Christopher Jones
FNAL

On behalf of the CMS Offline Organization
Outline

Motivation

Framework Design

Threading Model

Framework Implementation

Measurements

Conclusion
Why Threads?

Cost of Memory

- CPU designs double cores every 18 months
- Memory cost halves each 18 months
- Therefore, memory per core is now constant (2GB/core)
- Complexity of LHC events is only going up with high pileup
- May not be able to afford enough memory per core to use 1 event/core
  
  End of 2011 CMS could only use 6 of 8 cores in T0

Forking Is Not Enough

- Forking allows sharing of initial setup and conditions
- Each core still only handles one event
- Memory used by one event will increase as pileup increases
  
  This year shared memory between forked processes is less than private memory

Need to Use Multiple Cores/Event

- Naturally accommodated by threading
Framework Design
Framework Pieces

Events can be processed in parallel

An Event is filtered by Paths
Paths run in parallel

Paths hold a list of Filters
Filter runs only if previous Filter passes

EndPaths hold OutputModules
EndPaths run in parallel after Paths finish

Producers make data
Run first time their data is requested
Producers run in parallel

Filters, Producers & OutputModules
All are referred to as Modules
Run only after their input data is available
Parallelization

Threaded Framework

CHEP 2012
Parallelization
Parallelization
Parallelization
Parallelization
Parallelization
Parallelization

Threaded Framework
Parallelization
Parallelization
Parallelization
Parallelization

Threaded Framework
Parallelization

Threaded Framework
Parallelization
Parallelization
Parallelization
Concurrency is Limited by Dependencies Between Modules
Threading Model
libdispatch
Developed by Apple Inc
Port is available for Linux and Windows

Task Queue based system
Task is a C/C++ function plus context
  Context can be any data you want
Tasks are placed in a light weight queue
  Can easily support more than 100,000 queues in one process
Tasks are pulled from queues and then run
System guarantees that cores are not oversubscribed
Central Concepts

Global Concurrent Queue

Private Serial Queues

Task Groups
Global Concurrent Queue

One concurrent queue per process

Tasks pulled in order and run concurrently
Global Concurrent Queue

One concurrent queue per process

Tasks pulled in order and run concurrently
Global Concurrent Queue

One concurrent queue per process

Tasks pulled in order and run concurrently
Global Concurrent Queue

One concurrent queue per process

Tasks pulled in order and run concurrently
Global Concurrent Queue

One concurrent queue per process

Tasks pulled in order and run concurrently
Private Serial Queue
Can use many per process
Tasks pulled in order with only one run at a time
Private Serial Queue

Can use many per process

Tasks pulled in order with only one run at a time
Private Serial Queue
Can use many per process
Tasks pulled in order with only one run at a time
**Private Serial Queue**

Can use many per process

Tasks pulled in order with only one run at a time
Private Serial Queue

Can use many per process

Tasks pulled in order with only one run at a time
**Private Serial Queue**

Can use many per process

Tasks pulled in order with only one run at a time

Guarantees sequential behavior without having to use thread primitives
Task Group
Add new task to a queue once other tasks have finished

Queue

G

Cores

2
Task Group
Add new task to a queue once other tasks have finished

Queue

G

Cores
Task Group
Add new task to a queue once other tasks have finished

Queue

G

Cores
Task Group

Add new task to a queue once other tasks have finished
Task Group
Add new task to a queue once other tasks have finished
Task Group
Add new task to a queue once other tasks have finished
Task Group
Add new task to a queue once other tasks have finished

Provides a synchronization mechanism between tasks
Framework
Implementation
Implementation

Events
Run N Event instances simultaneously
N is configurable

Paths
Path starts a task for the first Filter on the Path
When Filter finishes it launches a task to run the next Filter on the Path

Modules
Modules have a list of data they will request from Event
Used to do parallel prefetching
Modules are shared between all Event instances
Keeps memory overhead as low as possible

ModuleWrappers
One per Module per Event
Has serial queue used to guarantee module is run only once per event
Has a task group used to notify when data prefetches have completed

ProducerWrappers
One per Producer per Event
Remembers if Producer has already run for that Event
Has a task group used to notify others when Producer has made its data
Simple Example

One Path
Contains Filter F

Two Producers
F needs data from 1 and 2
Simple Example

One global concurrent queue
Labelled G

Each module has own serial queue
Label and color match module

Queues

G  S  F  1  2

Cores

Job

1  2

S  F

Job

Threaded Framework

CHEP 2012
Simple Example

Machine has two cores

Queues:

- G
- S
- F
- 1
- 2

Cores:

- Job
  - 1
  - 2

Threaded Framework
Simple Example
Job starts by requesting new Event

Queues

G | S | F | 1 | 2

Cores

Job

1

2

S

F

Cores
Simple Example
Job starts by requesting new Event

Queues

Cores

Job

S

F

1

2

Threads Framework

CHEP 2012
**Simple Example**

Job starts by requesting new Event

---

**Queues**

G  S  F  1  2

**Cores**

Job

1  2

S → F

---

Threaded Framework

CHEP 2012
Simple Example
Task creates new task for source to read next event
Simple Example

Task creates new task for source to read next event

The task is placed on the source queue
Simple Example

Task creates new task for source to read next event
The task is placed on the source queue

Queues

G  S  F  1  2

Job

Cores

Threaded Framework 49  CHEP 2012
Simple Example

Task creates new task for source to read next event

The task is moved to the global queue
The source queue is blocked
The previous task finishes

Queues

| G | S | F | 1 | 2 |

Cores

Job

1
2
Simple Example

Task creates new task for source to read next event

The task is moved to the global queue
The source queue is blocked
The previous task finishes

Queues

G S F 1 2

Cores

Job

S F 1 2

Threaded Framework

CHEP 2012
Simple Example

The source reads the file

Queues

G S F 1 2

Cores

Job

1 2

S F

Threads Framework

CHEP 2012
Simple Example

The source reads the file
Once done, starts task to run Paths

Queues

G  S  F  1  2

Cores

Job

1  2

S  F

Threaded Framework  CHEP 2012
**Simple Example**

The source reads the file
Once done, starts task to run Paths

_queues.png

_queues.png

_cores.png

_queues.png

_queues.png

_queues.png
**Simple Example**

The source reads the file

Once done, starts task to run Paths

---

**Queues**

- **G**
- **S**
- **F**
- **1**
- **2**

**Cores**

**Job**

- **1**
- **2**

---

Threaded Framework
Simple Example

The source reads the file

Once done, starts task to run Paths

Queues

G  S  F  1  2

Cores

Job

1  2

S  F
Simple Example
Path task determines F must be run

Queues

G S F 1 2

Cores

Job

S F

1 2
Simple Example

Path task determines F must be run

Runs task to prefetch F’s data

Queues

G  S  F  1  2

Cores

Job

1  2

S  F

Threaded Framework

CHEP 2012
Simple Example

Path task determines F must be run

Runs task to prefetch F's data
Simple Example

Path task determines F must be run

Runs task to prefetch F's data
Simple Example

Path task determines F must be run

Runs task to prefetch F’s data
Simple Example
Prefetch queues data requests

Queues

G S F 1 2

Cores

Job

1 2

S F

Threaded Framework

CHEP 2012
Simple Example
Prefetch queues data requests

Queues
G S F 1 2

Cores

Job

Prefetch queues data requests
Simple Example

Prefetch queues data requests
Creates group to wait for both data requests

Queues

G  S  F 1  2

Cores

Job

1  2

S  F

Prefetch queues data requests
Creates group to wait for both data requests
Simple Example

Prefetch queues data requests
Creates group to wait for both data requests
Simple Example
Prefetch queues data requests

Queues
G S F 2 1 2

Cores

Job
1 2

Prefetch queues data requests
Simple Example
Prefetch queues data requests

Queues

G
S
F
1
2

Job
1
2

S
F

Cores
Simple Example

Prefetch queues data requests

Producer queues are halted
  Makes sure only run once per event

Queues

Cores

Job

G  S  F  Queues

1  2

S  F

Job

Cores
Simple Example
Producers do their work

Queues

G | S | F | 1 | 2

Cores

Job

1 2

S F

Job

Cores
**Simple Example**

Producers do their work

Queues

- G
- S
- F
- 2
- 1
- 2

Job

1

2

Cores

S

F

Threaded Framework

CHEP 2012
**Simple Example**

Producers do their work

---

**Queues**

- G
- S
- F
- 1
- 1
- 2

**Cores**

**Job**

1 → S → 1

2 → F → 2

---

Threaded Framework  62  CHEP 2012
Simple Example
Producers do their work

Queues

G S F1 1 2

Job

S F

Cores

1 2
Simple Example

Group runs task to run F

Queues

G  S  F 0  1  2

Job

1  2

S  F

Cores

63
Simple Example
Group runs task to run F

Queues
G S F 1 2

Job
1 2
S F

Cores
Simple Example

Group runs task to run F

F’s queue is halted
  If F were on many paths it would be called once

Queues

| G | S | F | 1 | 2 |

Cores

Job

1

2

S

F

Threads Framework

CHEP 2012
Simple Example

Group runs task to run F

F’s queue is halted

If F were on many paths it would be called once
Simple Example

Filter F is run

Queues

G  S  F  1  2

Job

1  2

S  F

Cores

Threaded Framework  66  CHEP 2012
Simple Example

Filter F is run
Once done, runs task to advance Path

Queues

Cores

Job

Filter F is run
Once done, runs task to advance Path
Simple Example

Filter F is run
Once done, runs task to advance Path

Queues

G  S  F  1  2

Jobs

Cores
Simple Example

Filter F is run
Once done, runs task to advance Path

Queues

G S F 1 2

Cores

Job

1 2

S F
Simple Example
Path task sees Path has finished
Submits new task to get next event

Queues

G S F 1 2

Cores

Job

S F 1 2
Module Thread Safety

Fully Re-entrant
Same Module instance can be run simultaneously for different events
ModuleWrapper’s serial queue is unique for each event instance

One event at a time
Module can only handle one event at a time
E.g. it uses member data to store event info temporarily
All ModuleWrappers for the same Module share the same serial queue

Thread-unsafe
Module can’t run at the same time as other thread-unsafe Modules
E.g. all the modules call the same third party non-thread safe library
ModuleWrapper for thread-unsafe module share the same serial queue
Measurements
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
Testing System

Physical Machine
Intel(R) Xeon(R) CPU E5620
16 physical cores @ 2.40GHz
4 Cores/CPU with 4 CPUs
47 GB RAM

Virtual Machine
16 virtual cores
15 GB RAM
Scientific Linux 6
All Producers are doing numeric integration calibrated how many seconds per integral length

Thread-unsafe scales to 95+% of single threaded
Both are running N processes rather than N events in one process

Fully re-entrant peaks at 30% faster
All Producers are calling usleep
One event per module slower after 2 simultaneous events (se)
One event I/O turns over at 25se and stops growing at 44se
Fully re-entrant stays 30% faster till runs out of system threads
Single threaded runs out of memory at 800se
Concurrency Limit

Number of Running Modules vs Time for High Pileup RECO

Short periods of high module level parallelism

Long periods with only 1 or 2 modules

First period is tracking
Second period is photon conversion finding

Parallelizing within those module would be beneficial
Conclusion

Task queue based systems can be used for HEP frameworks

Technology scales well
Can transition code to be thread safe one module at a time
Don’t expose thread primitives to physicists
   Can use task queues internal to their own modules which are simpler than locks

Concurrency limited by dependencies between modules

Parallelizing tasks within long running modules would be beneficial

Presently testing additional threading technologies
OpenMP
Intel’s Threading Building Blocks

CMS will choose a threading technology this year
Start transitioning CMS’ framework to use threads in 2013