Introduction to Multi-Threading

Christopher Jones
LArSoft 2017 Workshop
20 June 2017
Overview

Who am I?

What is multi-threading?

Why use multi-threading

Multi-threading difficulties

Thread safety

Things to avoid
Who am I?

I’m in charge of the CMS event processing framework
art was derived from an earlier version of the CMS framework

I lead CMSes effort to switch to a multi-threaded framework
I designed the how the CMS framework uses threads
I lead the team that changed the framework
I implemented a large portion of the conversion
I provided instructions to the CMS developers on how to adapt their code

This talk is based on that 6 years of experience
What is Multi-threading?

A **thread** is the smallest unit of sequential processing

- Each thread has its own call stack
  - Have own function local variables
  - Have own function callback stack

All threads in a process share the same memory address space

- If a thread changes a value in memory it affects other threads using the memory

Contrast: multiple processes do not share the same memory address space

- If a process changes a memory address it can NOT affect other processes
Why Use Multi-threading

Speed

Shared Resources
Speed

Using multiple threads can make processing one event faster
There is always some point where threads are waiting for the last thread to complete

Multi-threading does not decrease the time to process ALL events
Running multiple single-threaded jobs each processing events is usually faster
Many single-threaded jobs are usually the most CPU efficient

Why bother with multi-threading?
Computing Hardware Trends

CPU frequencies no longer increase

Manufacturers are increasing number of CPU cores

Cost of memory decrease at a slower rate than increase core count
Memory per Core is either flat or decreasing

Intel Xeon we can afford 2 GB/core

Intel Xeon Phi have 256 ‘cores’ but only afford 96GB total
If single threaded job takes 1.5 GB could only run 64 jobs on the machine
Would take 4x as many computers to use single-threading compared to multi-threading
Shared Resources

Multi-threaded programs can share memory across many events
  E.g. Geometry, Conditions, Configuration
  CMS: 1.5GB can be shared and each event needs 0.2 to 0.5 GB

Multi-threading help Batch and Workflow management systems scale
  Do not have to have a batch slot per core
  Systems only grow as the number of machines, not number of cores

Multi-threading puts less pressure on computing sites
  Database connections are shared across cores
  File opens can be shared across cores
Multi-threading Difficulties

Race condition

Deadlock
Race Condition

A shared memory address where
One thread is writing to the memory
Another thread is reading or writing to the memory

There is no such thing as a ‘benign’ race condition
Race Condition Example

```cpp
std::vector<double> values;

Thread 1
values.push_back(1.);
for (auto v : values) cout << v;

Thread 2
values.push_back(2.);
for (auto v : values) cout << v;
```

Results could be

- 1.0 1.0 2.0
- 2.0 2.0 1.0
- 1.0 2.0
- 2.0 1.0
- segmentation violation
CPU Memory Model

![CPU Memory Diagram]

Double-click to edit

Main Memory
Race Condition Example

```cpp
std::vector<bool> bools={false,false};

Thread 1
bools[0] = true;
cout <<" 0 "<<bools[0];

Thread 2
bools[1] = true;
cout <<" 1 "<<bools[1];
```

Results could be
‘0 true’ and ‘1 true’
‘0 false’ and ‘1 true’
‘0 true’ and ‘1 false’
Deadlock

A deadlock is when two or more threads are waiting for other threads in the group to finish before continuing.

Thread 1

acquireGeometry();
acquireCalibration();
...
releaseCalibration();
releaseGeometry();

Thread 2

acquireCalibration();
acquireGeometry();
...
releaseGeometry();
releaseCalibration();
Thread Safety

The easiest way to be thread safe is to never have a thread write to memory that another thread will read.

For all other cases, C++ requires the use of synchronization mechanism

E.g. mutex, semaphore, atomic

These will not be covered in this talk

art will automatically handle synchronization across modules

An object put into the Event by a module can safely be read by a module in another thread
Levels of Thread Safety for Objects

- Thread-hostile
- Thread friendly
- const-thread safe
- Thread safe
Thread Hostile

It is not safe for more than one thread at a time to call methods even for different class instances

E.g. with `static`

```cpp
class Foo {
public:
    int convert(int iIn) const {
        static int s_oldIn{iIn};
        static int s_cache{ calculate(iIn) };
        if( iIn != s_oldIn ) {
            s_cache = calculate(iIn);
            s_oldIn = iIn;
        }
        return s_cache;
    }
    ...
};
```
Thread Friendly

Different class instances can be used by different threads safely
Sharing the same instance across multiple threads is not safe

E.g. with mutable cache

```cpp
class Foo {
    mutable int cache_; 
    mutable int oldIn_; 

public:
    int convert(int iIn) const {
        if( iIn != oldIn_) {
            cache_ = calculate(iIn);
            oldIn_ = iIn;
        }
        return cache_; 
    }
    ...
};
```
const Thread-Safe

Multiple threads can call const methods on the same class instance
Classes in the C++ standard library are const thread-safe

Classes put into the art::Event must be const thread-safe

```cpp
class Foo {
    std::vector<int> values_;

public:
    Foo() : values_{calculateAllAllowedValues()} {} 

    int convert(int iIn) const {
        return values_[iIn];
    }

    ... 
};
```
Thread Safe

Multiple threads can call non-const methods on the same class instance

Intel’s Thread Building Block library has thread-safe containers
tbb::concurrent_vector, tbb::concurrent_hash_map, etc.
TBB is distributed with art

Thread 1
```
tbb::concurrent_vector<double> values;
startAndWaitForOtherThreadsToFinish(values);
for( auto v:values) { cout << v<<“ “; }
```

Thread 2
```
values.push_back(1.);
```

Thread 3
```
values.push_back(2.);
```

Results could be
```
1.0 2.0
2.0 1.0
```
Things to Avoid

- non-const global memory
- mutable data members in Event data products
- art based Services with mutable state
- Starting your own threads
Avoid: Non-const Global Memory

Global Memory: memory accessible from global C++ scope

Types of global memory
- File scope variables
- Function static variables
- Class static variables

No way to know if another thread is changing the values
Avoid: Mutable Data Members in Event Data Products

Event data products are shared across threads
Multiple threads can be calling const functions on the same class instance

class Displacement {
    Cartesian3D vec_;  
    mutable Polar3D polar_;  
    mutable bool polarIsSet_;  
    public:
        Polar3D const& polar() const {
            if(not polarIsSet_) { polarIsSet_ = true;  
                polar_ = calculatePolar();  }  
            return polar_;  
        }  
    ...  
};
Avoid: Services with Mutable State

Services are shared across threads

NOTE: storing event data in a Service is not considered best practice in *art*

class TrackFittingService {
    std::vector<Hit> hits_;  
public:
    void setHits(std::vector<Hit> const& iHits) {
        hits_ = iHits; }
    Track fitToTrack() const;
    ...
};
Avoid: Starting a Thread

Grid sites specify how many threads a process can use

*art* will start with the maximum allowed number of threads

Additional high CPU threads can lead to a site killing the process

*art* will provide facilities to allow you to do work concurrently within a module
Calling TBB parallel algorithms within an art module is supported
To start TBB add to configuration: `services.num_threads: <n>`
Conclusion

Multi-threading is coming to *art*

This will allow HEP processes to run on resource constrained systems

Need to prepare code now
  Remove use of ‘global’ variables
  Do not use mutable member data for Event data products
  Remove mutable state from Services
Advanced Resources

Useful book about C++ concurrency
“C++ Concurrency in Action” by Anthony Williams

Useful talk about C++ threading memory model