# Handling External Libraries with Conflicting Thread Pools

Dr Christopher Jones

LArSoft Multi-threading and Acceleration Workshop

02 March 2023

# When to Worry About Threads Not Controlled by Framework

- Thread pools used for waiting on external communication are good
  - e.g. xrootd client’s pool of threads which are waiting on IO calls
  - e.g. thread asynchronously waiting for communication with a database

- Multiple thread pools used for heavy computation are problematic
  - Examples of such cases are
    - the eigen vectorization library
    - the tensorflow library
    - E.g. using the C++ standard parallel algorithms
    - E.g. Using OpenMP

# What is the Problem?

- Using more computational threads than cores on a machine is a waste
  - Will not get work done faster
    - given the context switching required it might actually make jobs a bit slower
  - Requires more memory than necessary
    - each thread requires its own stack which takes some memory
    - the work being done may require additional temporary storage
      - e.g. algorithm may be temporarily filling an std::set

- Jobs running on a grid site are allotted a fix number of cores to use
  - Exceeding a job slots allowed core utilization
    - can get jobs killed or
    - get your experiment band from a site

# How to Deal with the Problem

- Glib answer

# How to Deal with the Problem

- Glib answer

**Don’t Do It**

# Simplest: Whole Node

- Run jobs only where you are given the whole node
  - E.g. this is the standard behavior for super computer sites

- Still pay the penalty for oversubscribing the machine
  - application maybe doing lots of context switches which may slow it down
  - requiring more memory than actually needed

- Can emulate this on grid sites using virtual machines or containers
  - you grid jobs start up a VM or a container which specifies how many cores to use
    - the actual job is run within that environment but only ever uses the number of cores even if the number of threads in the application is higher
  - there is some performance lose when using a VM or container

# Best: Share the Pool

- Work with library developers to allow use of framework’s thread pool

- This is what CMS has attempted to do
  - CMS, ATLAS and art use the Intel Thread Building Block for thread pool
  - CMS worked with the following groups to be sure we could use their software
    - ROOT
    - Geant4
      - Made sure their original threading model would not break CMS’ model
    - Tensorflow
      - Gave feedback to Google on allowing external thread pools to run the tasks

# Compromise: Pools with only One Thread

- Explicitly tell 3rd party libraries to only use 1 thread
  - CMS does this for Eigen and is how CMS originally dealt with Tensorflow
  - CMS does this with the CPU implementation of SONIC
    - CMS application asks SONIC to do work and the CMS thread blocks till work is done
    - SONIC CPU implementation only uses lots of CPU time when doing work, not while waiting

- This works if framework can keep all of the threads busy
  - Collider based experiments typically have perfectly parallelizable event processing
    - each event is statistically independent of each other
    - CMS can run as many concurrent events as threads and still be within reasonable memory limits
  - CMS can also use multiple threads to concurrently process data from a single event

# Hard: Statically Allocate Threads to Each Pool

- Specify at job setup how many threads each pool can use
  - e.g. TBB set to 2 threads, Tensorflow set to 4 and Eigen to 2

- Requires lots of testing ahead of time for each job configuration to determine good mix of threads
  - Also different computation sites might allow different number of cores per job which means would need special configurations for those sites

- Would be easy to accidentally oversubscribe

- Likely to be less efficient than other cases