Testing Framework and Early Performance Results

Dr Christopher Jones

CCE IOS

23 September 2020

Testing Framework Purpose

- Mimic the characteristics of a HEP data processing framework
  - Similar multi-threaded behavior
  - Similar I/O behavior
  - Should reasonably behave like CMS, ATLAS and DUNE frameworks

- Easily try different I/O implementations
  - Choose what to use via command line arguments

- Experiment agnostic
  - With ability to read actual experiment ROOT files
    - ROOT will dynamically load dictionaries as needed

- Make it easier to perform performance measurements
  - I/O performance
  - threaded scaling performance

Mimicry

- Only deals with processing of *Events*
- An **Event** is just a collection of *Data Products*

- **Data Product**
  - Can be any C++ type
  - Each Data Product can be accessed independent of all other Data Products

- **Source**
  - Mimics reading of Events

- **Outputer**
  - Mimics writing of Events

- **Waiter**
  - Mimics processing of Events

Processing

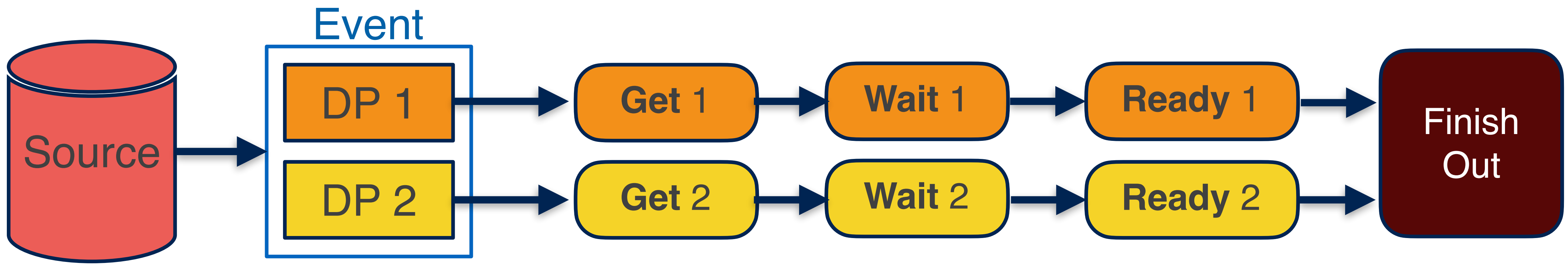
- Specify maximum number of threads to use
  - Use Intel's Threading Building Blocks to control threads
    - Used by CMS, ATLAS, DUNE and ROOT

- Asynchronous calls encapsulate work to be done into a Task
  - Task gets passed to TBB which runs it when a thread becomes available
  - When a Task finishes, it often makes another asynchronous call

- Specify the number of concurrent Events to use
  - Each Event has its own *Lane*
  - Lanes run concurrently

Lane

- Handles processing of one Event at a time



- Processing order
  - Asynchronously requests new Event from framework
    - Request goes to Source
  - Makes an asynchronous request to *get* each Data Product
    - Request goes to Source
  - When *get* completes, start asynchronous request to the Waiter assigned to the Data Product
  - When Waiter finishes, asynchronously signal to Outputer that a Data Product is *ready*
  - Once Outputer is done with all data product, asynchronously signal Outputer the event is *finish*ed

Source

- Each Lane has its own instance of a Source
  - I plan to change this so have possibility of sharing Source between all Lanes

- Source is told which Event index to retrieve
  - Tells framework when it has no more Events to retrieve which stops processing

- Source can optionally delay retrieving a Data Product until it is requested
  - This is a standard behavior of HEP frameworks

Available Source Types

- EmptySource
  - Adds no Data Products to the Events

- RootSource
  - Reads a ROOT file
  - Reads all TBranches from a TTree with name “Events”
    - Each TBranch becomes its own Data Product

- RepeatingRootSource
  - Reads the first N Entries of the ROOT file and caches the Data Products to memory
  - Cycles through the Data Product lists for each Event request

- HDF5Source
  - coming soon

- All Sources report total time taken to *read* all the Events

Waiter

- Use of Waiters can be disabled to test performance

- Waiters are configured with a scaling parameter to set sleep time
  - sleep time is scaling parameter times the *size* of the Data Product
    - Data Products size is set by Source and is the number of bytes read
  - replicates the fact that larger Events take more time to process

Outputer

- All Lanes share the same Outputer
  - Calls to Outputer are all done through an asynchronous API
    - This allows scheduling to handle serialization of calls if needed for thread-safety

- Outputer can optionally request to know when each Data Product ready
  - This allows ability to serialize each Data Product independently

- Outputer is informed when all Data Products are finished
  - This happens after all Data Product ready requests have finished
  - This allows *writing* of all Event Data Products if required to do it together

Available Outputer Types

- DummyOutputer
  - Does nothing on each call
  - Has option to turn on/off the use of Data Product *ready* calls

- SerializeOutputer
  - applies ROOT serialization to a Data Product during its *ready* call
    - these calls can be done concurrently
  - does nothing on the event finish call
  - Reports the total time spent serializing each Data Product

- RootOutputer
  - coming soon

- HDF5Outputer
  - coming soon

Initial Performance Testing

- Be sure any scaling limits are not caused by the test framework itself

- Machine Used
  - AMD Opteron(tm) Processor 6128
  - 4 CPUs with 8 Cores per CPU

- CMS ROOT file used
  - Contains standard Reconstruction output plus the RAW data
    - 272 Data Products
    - Wide distribution in size on disk/in memory for the different Data Products

- Testing procedure
  - Number of Events processed in a job is directly proportionally to number threads used
    - Exception is when jobs stop scaling with threads, then fix number events processed
  - Unless otherwise noted, number of concurrent Events == number of threads
  - Machine was always fully loaded
    - #threads per job * # concurrently running jobs == 32

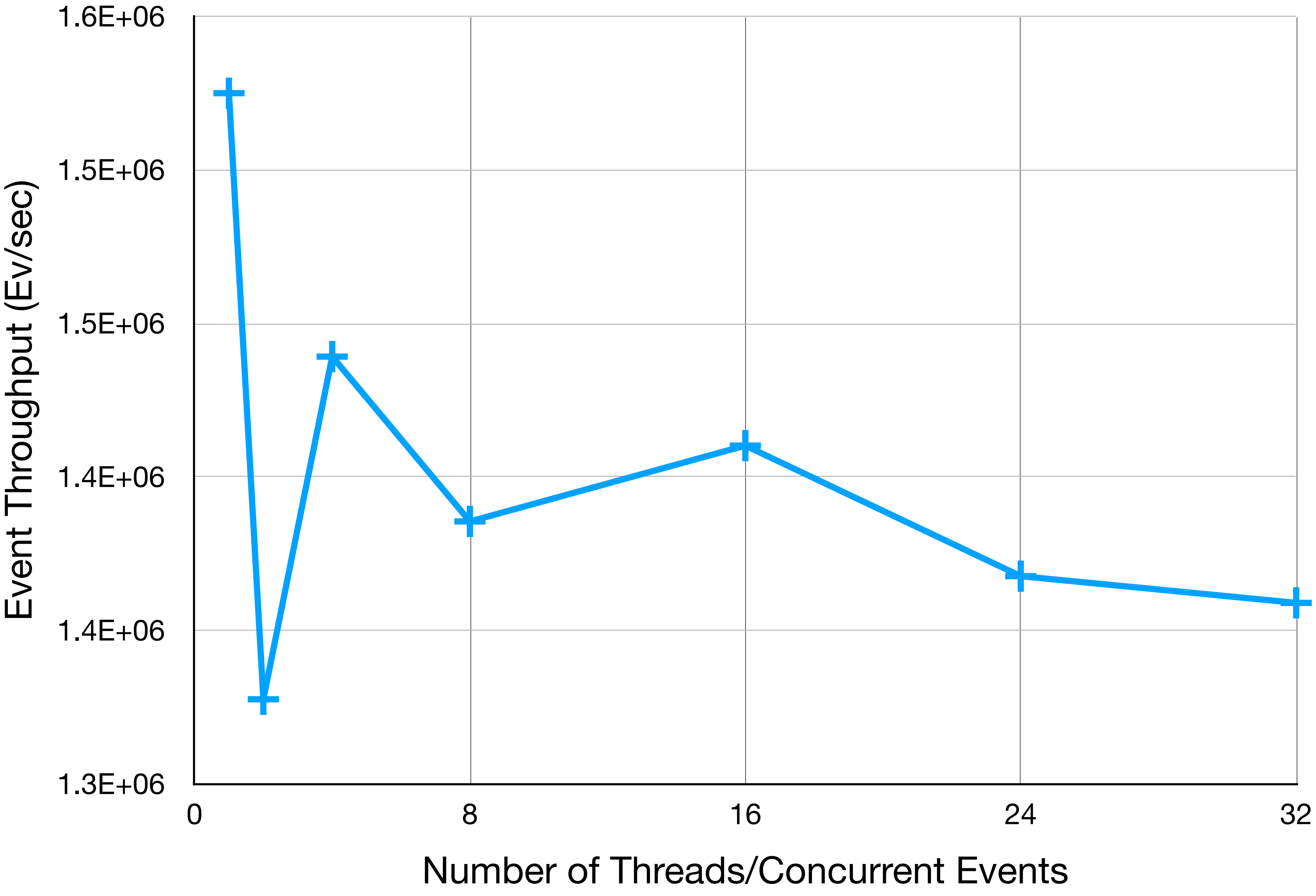
Top Speed Test

- EmptySource, no wait, no ready calls
  - Since no Data Products means no *get* calls are made
  - Only request Event and Event finished requests are made

No Scaling

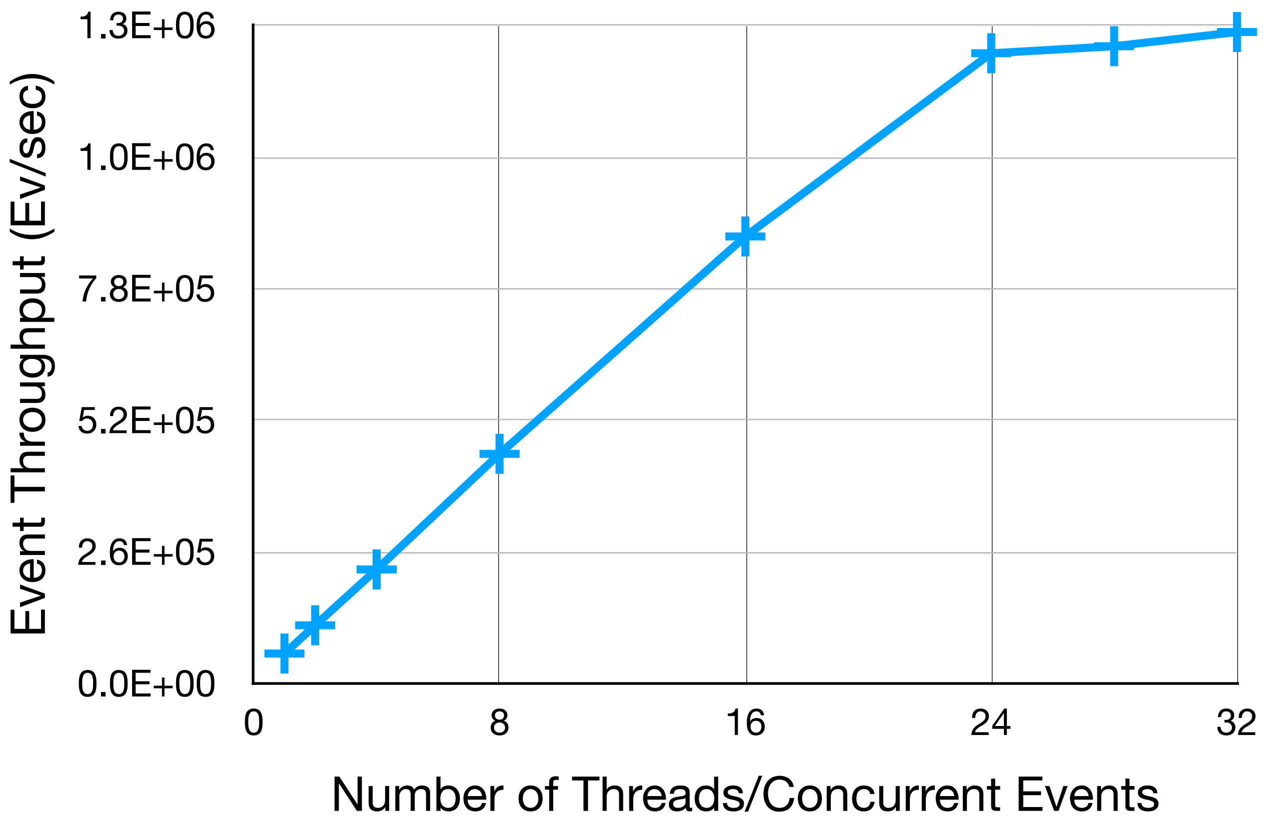
Standard CMS processing rate is 0.1 Ev/sec/thread

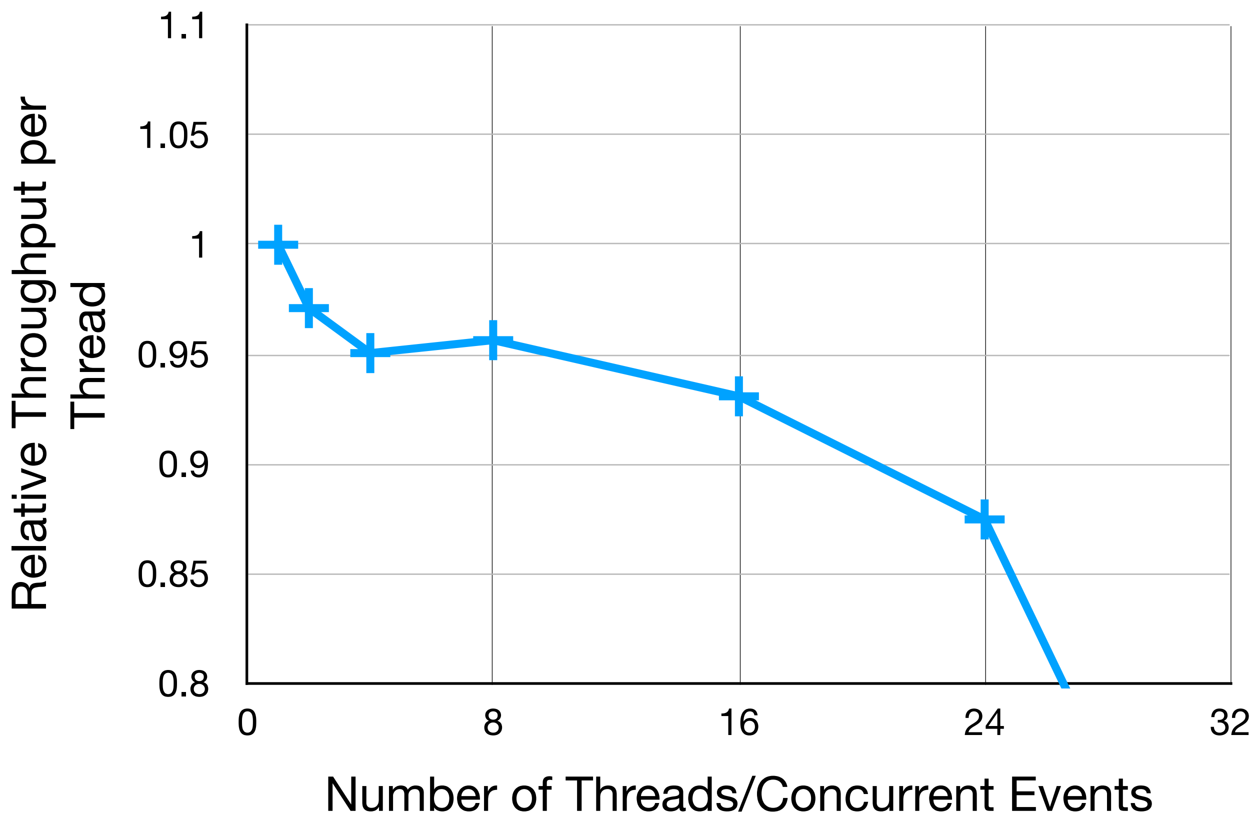
This is 7 orders of magnitude higher >> not a problem



RepeatingRootSource Top Speed Test

- RepeatingRootSource, no wait, no ready
  - Stores all Data Products for the first 10 Events into memory
  - Request Event, Data Product gets, and Event finished are called





Good Scaling till hit limit at 24 threads

RepeatingRootSource and Product ready

- Same as before except tell DummyOutputer to use Product ready calls

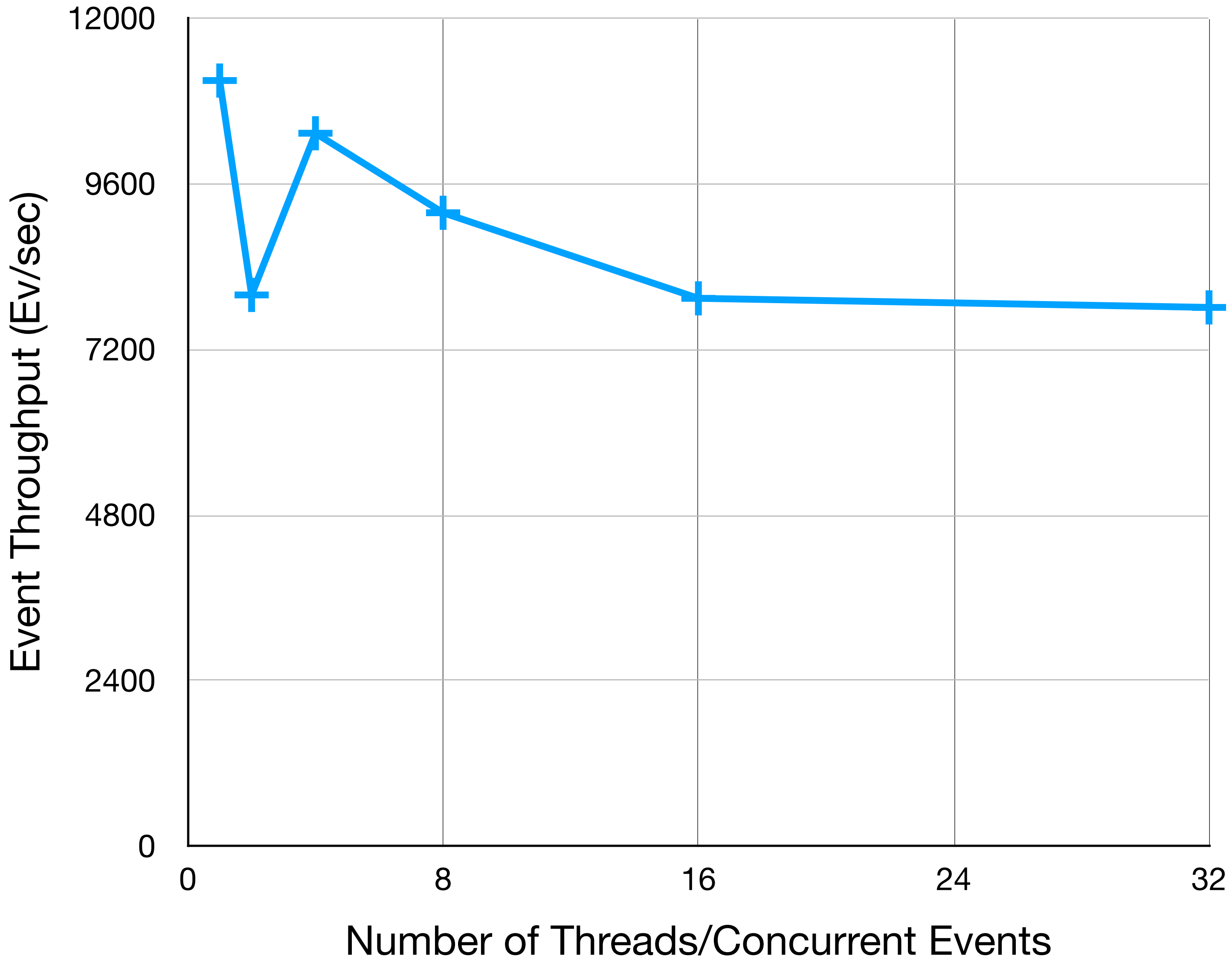
No Scaling

100x slower peak than before

Time spent in Source (not shown)
also 100x slower even though exact
same function call

Supposition: The extra work has
caused the CPU caches to flush
so extra time is handling cache misses

This is 5 orders of magnitude
higher >> probably OK



ROOT Serialization Scaling Performance Testing

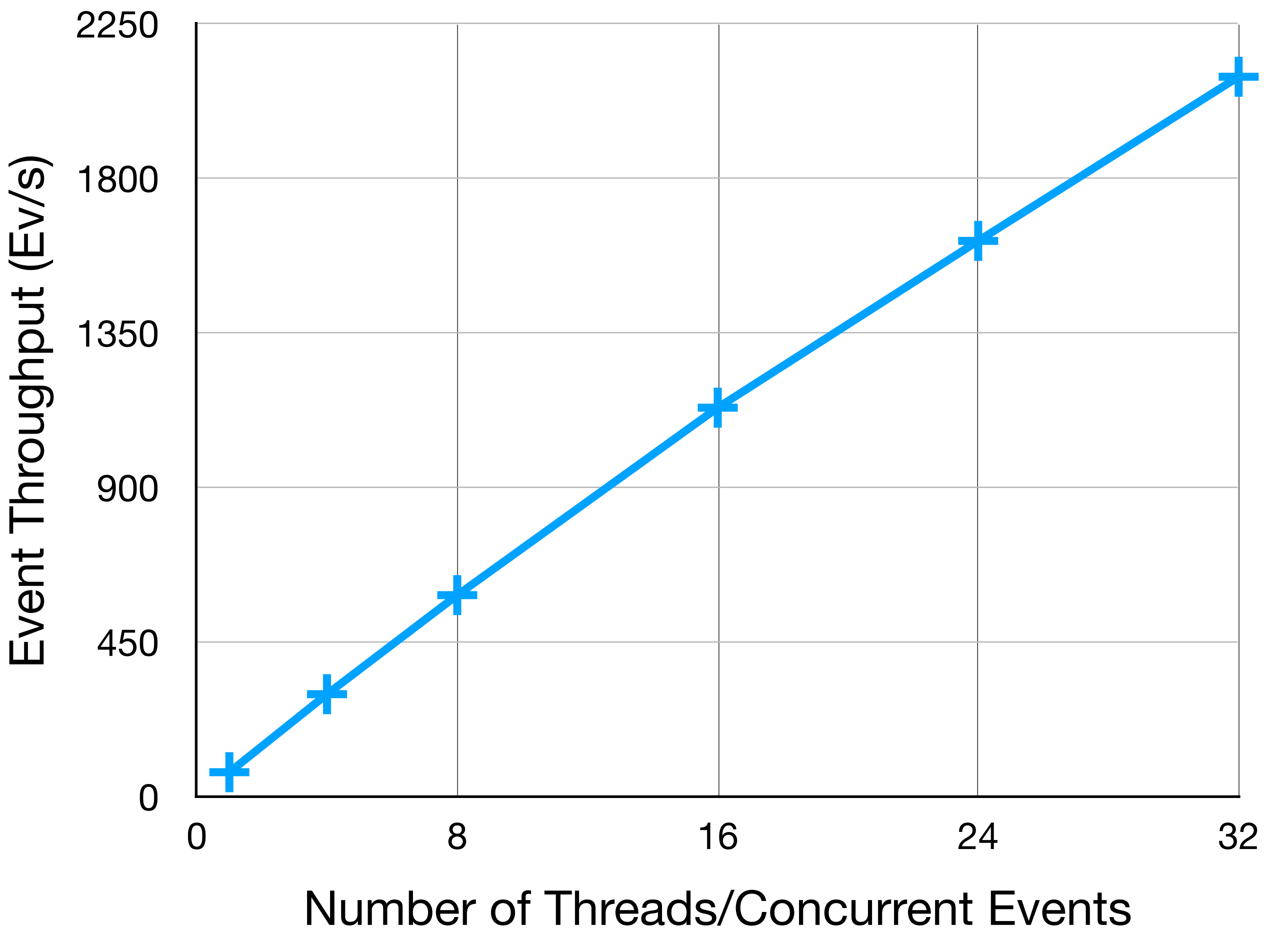
- Use same machine
- Use same testing procedure

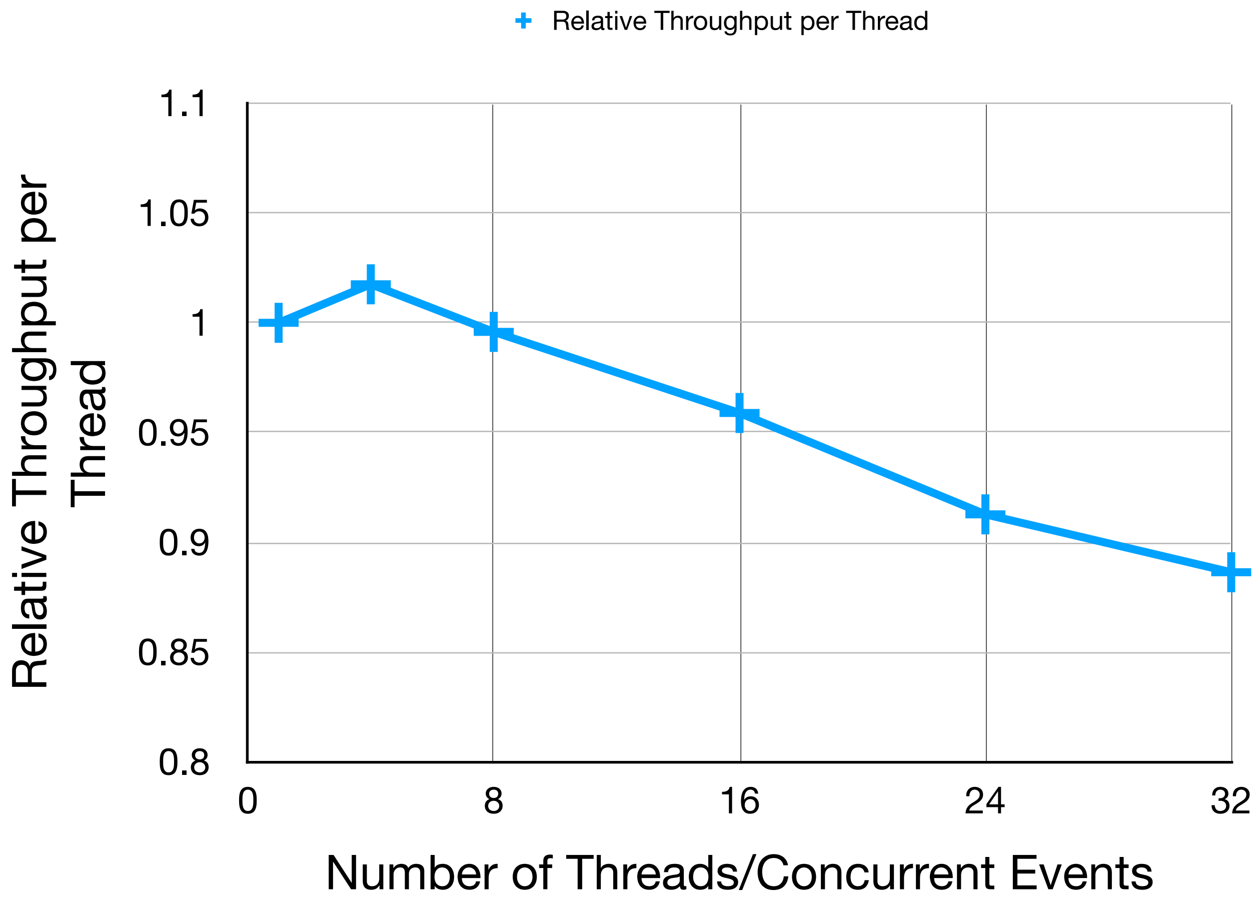
- Use RepeatingRootSource
  - Read first 10 events of same file used in previous tests

- Use SerializeOutputer
  - On Product Ready call it uses ROOT to serialize the Data Product
    - measure the time it takes to do serialization

Throughput

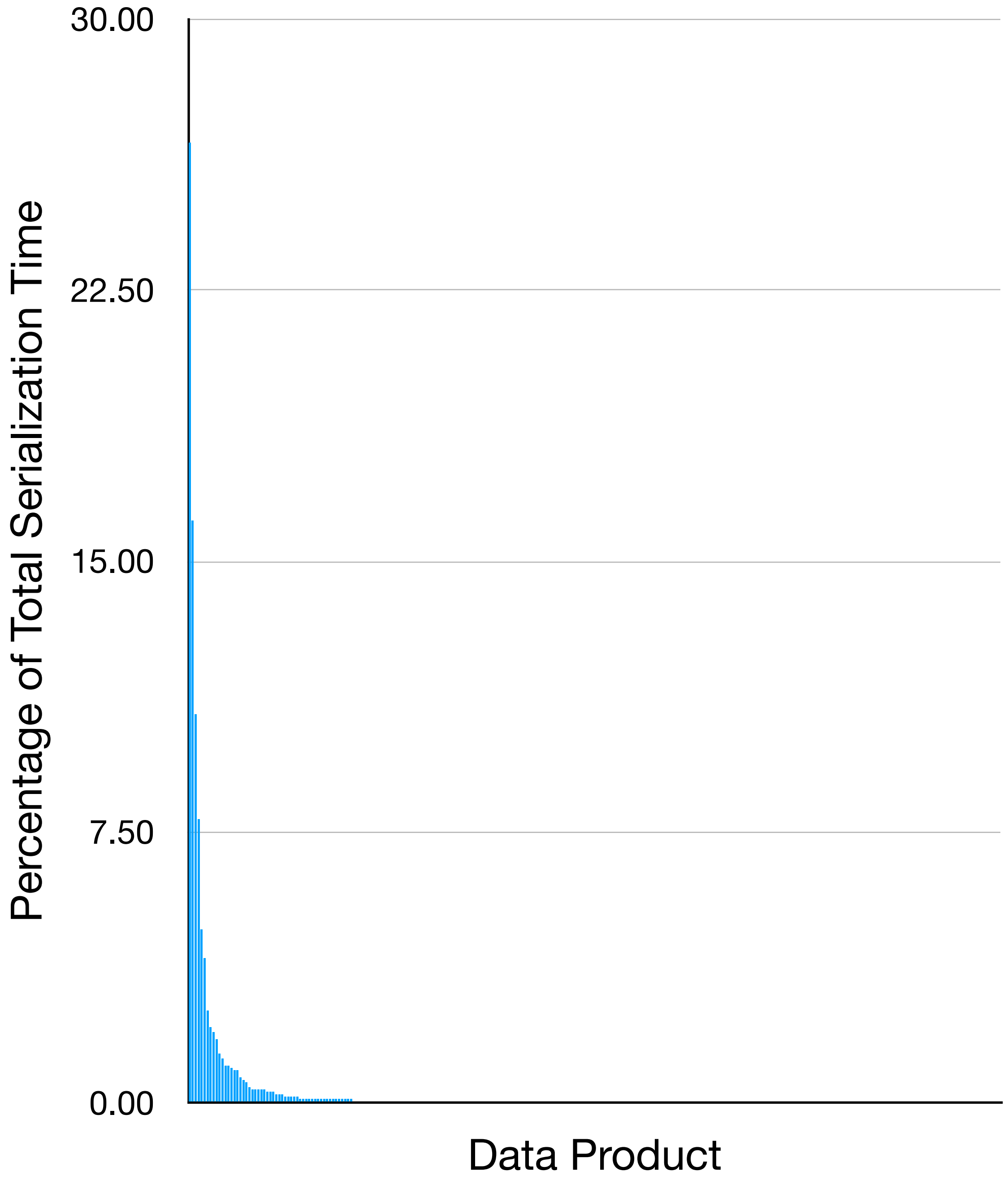
- See good scaling
  - 88% efficient at 32 threads
  - Possible sign of a minor concurrency issue?

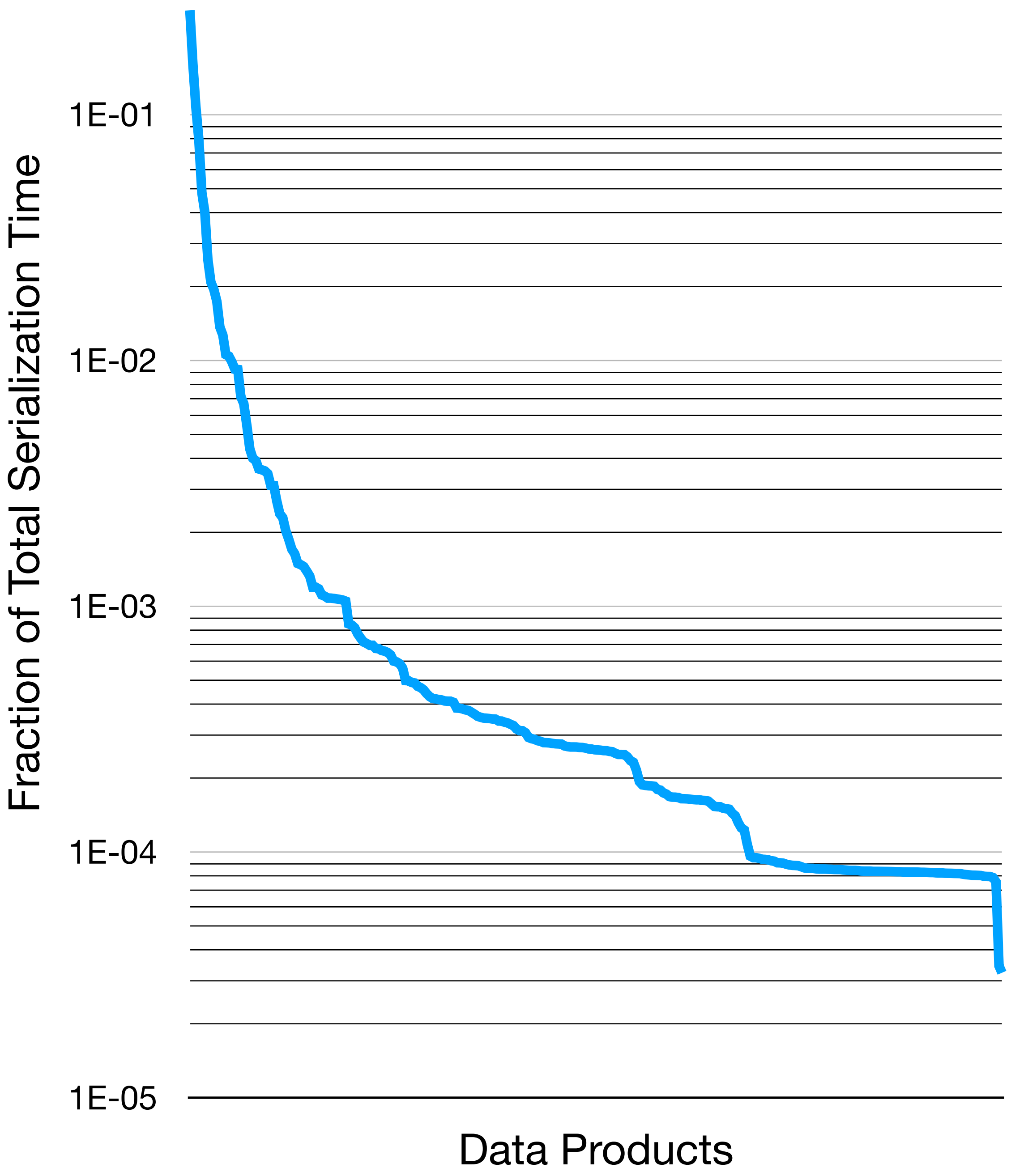




Time Doing Serialization

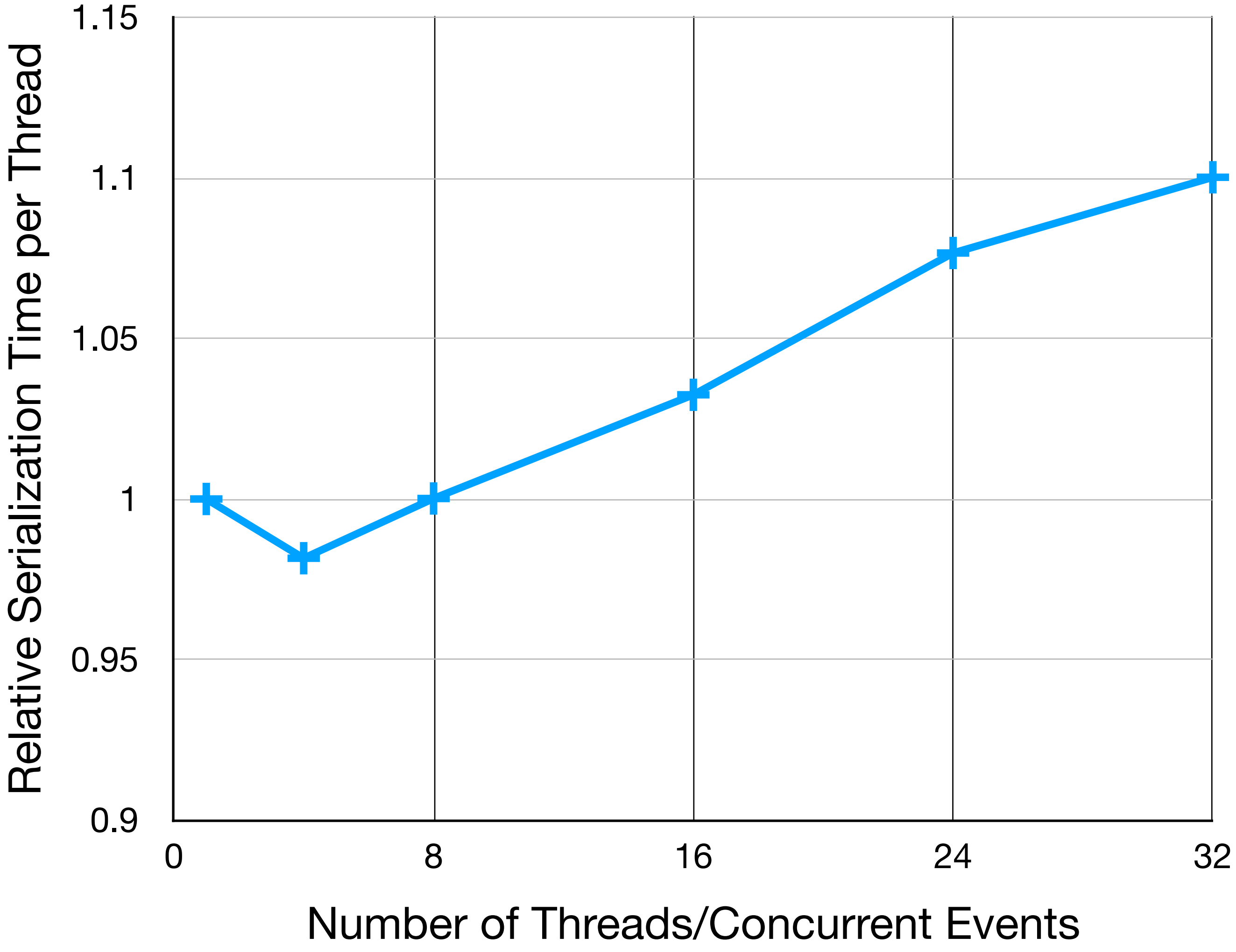
- 50% of total serialization time is spent by just 4 Data Products

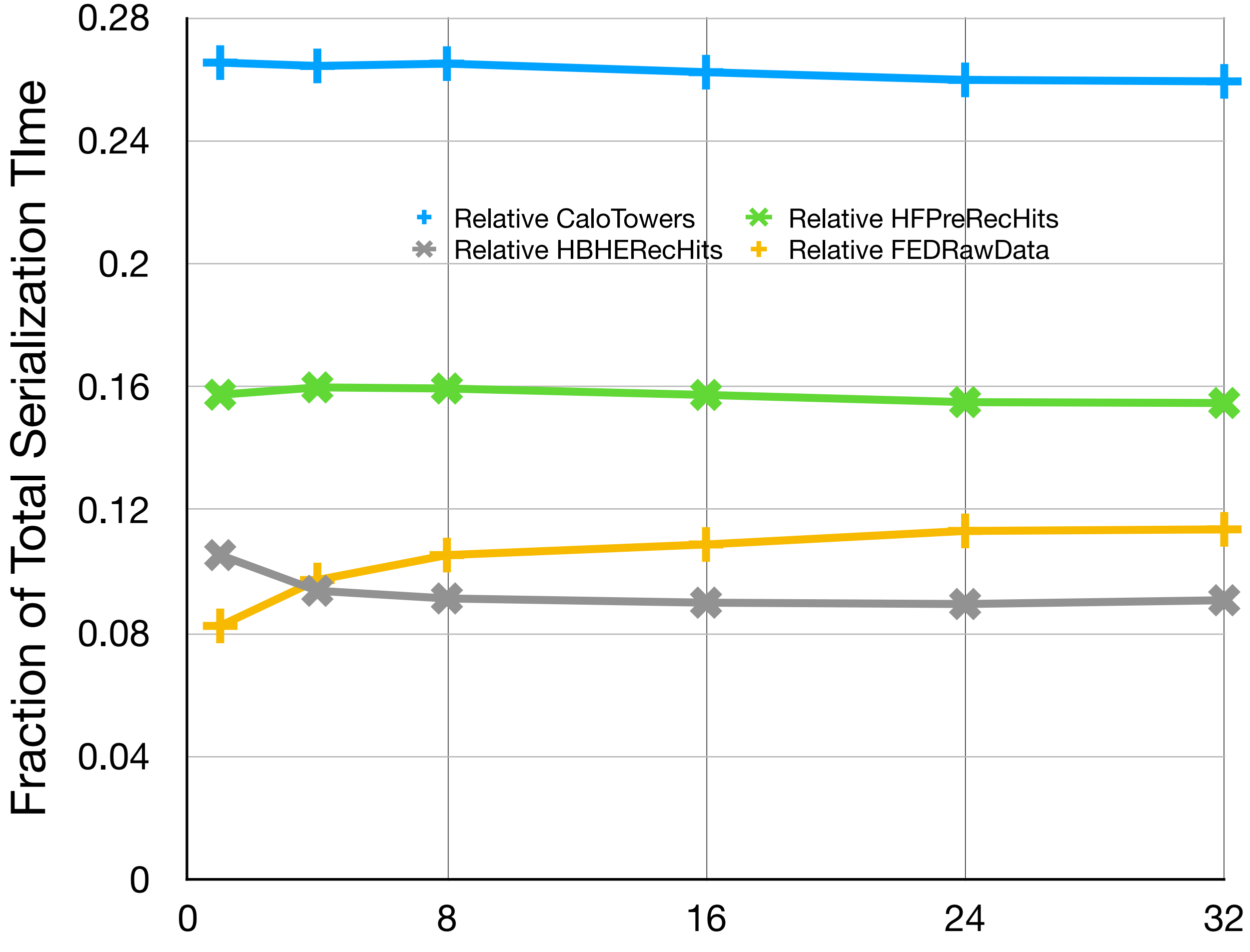




Time Doing Serialization

- Time to do same amount of serialization increases with number of threads
  - 10% longer at 32 threads
- Relative fraction of serialization changes with number of threads
  - Two of the Data Products appear to have different thread scaling

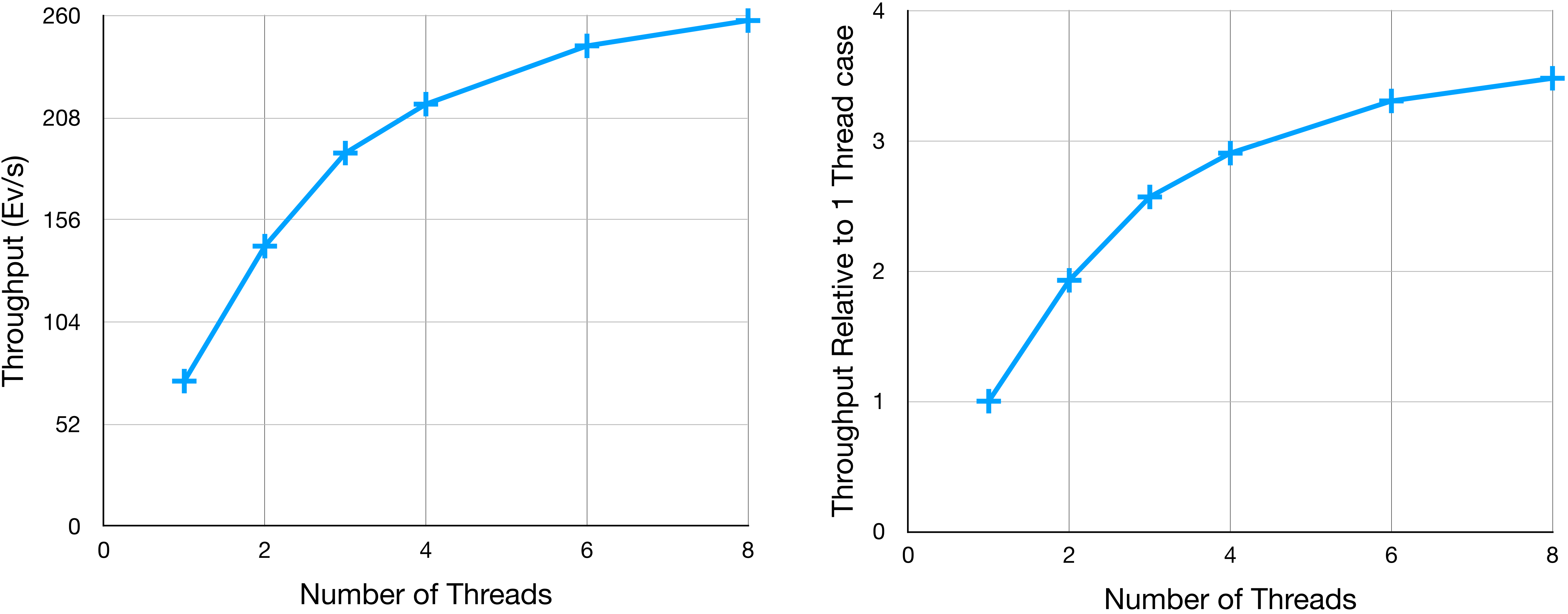




Root Serialization with 1 Event and Multiple-Threads

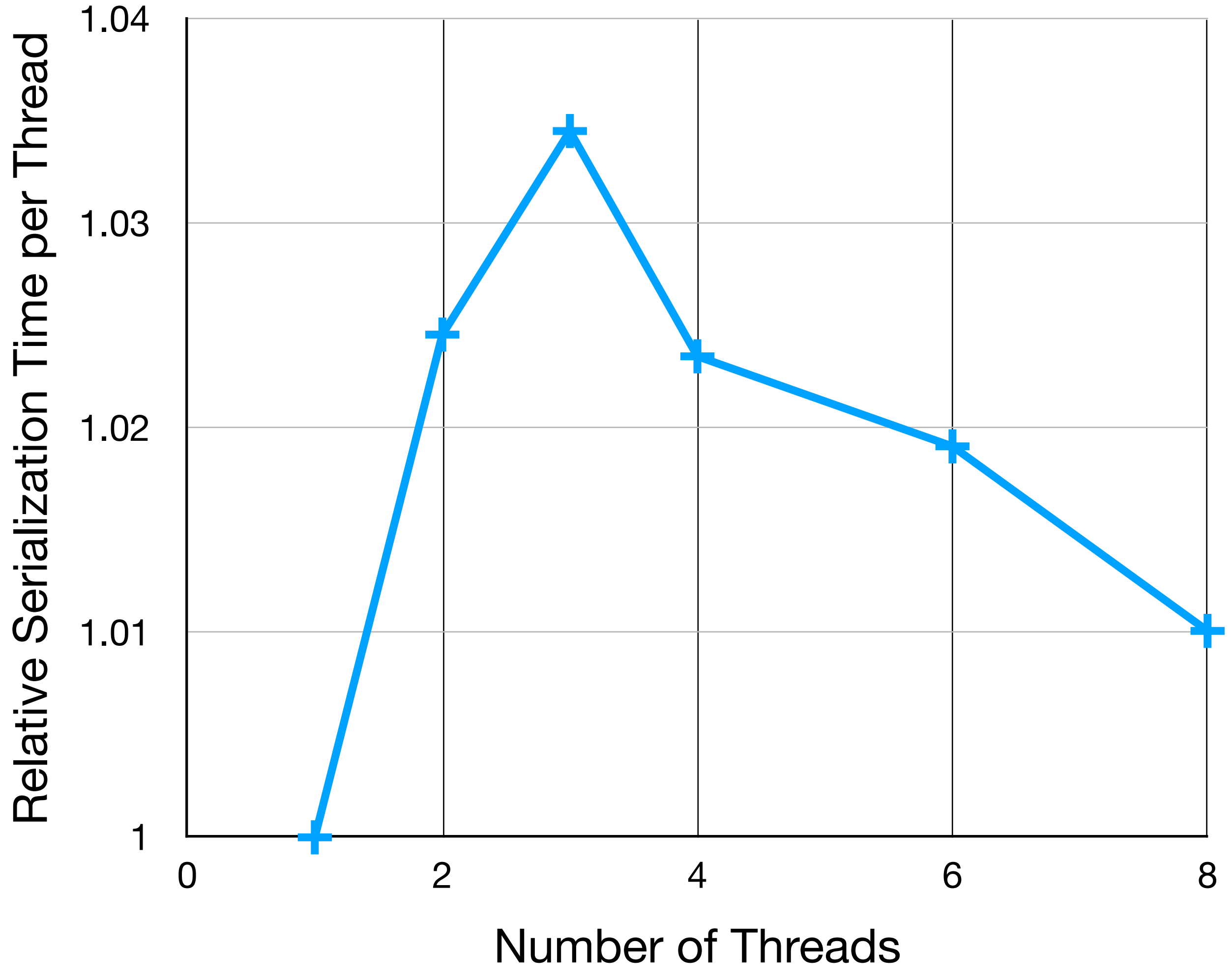
- Vary number of threads but only allow 1 concurrent Event

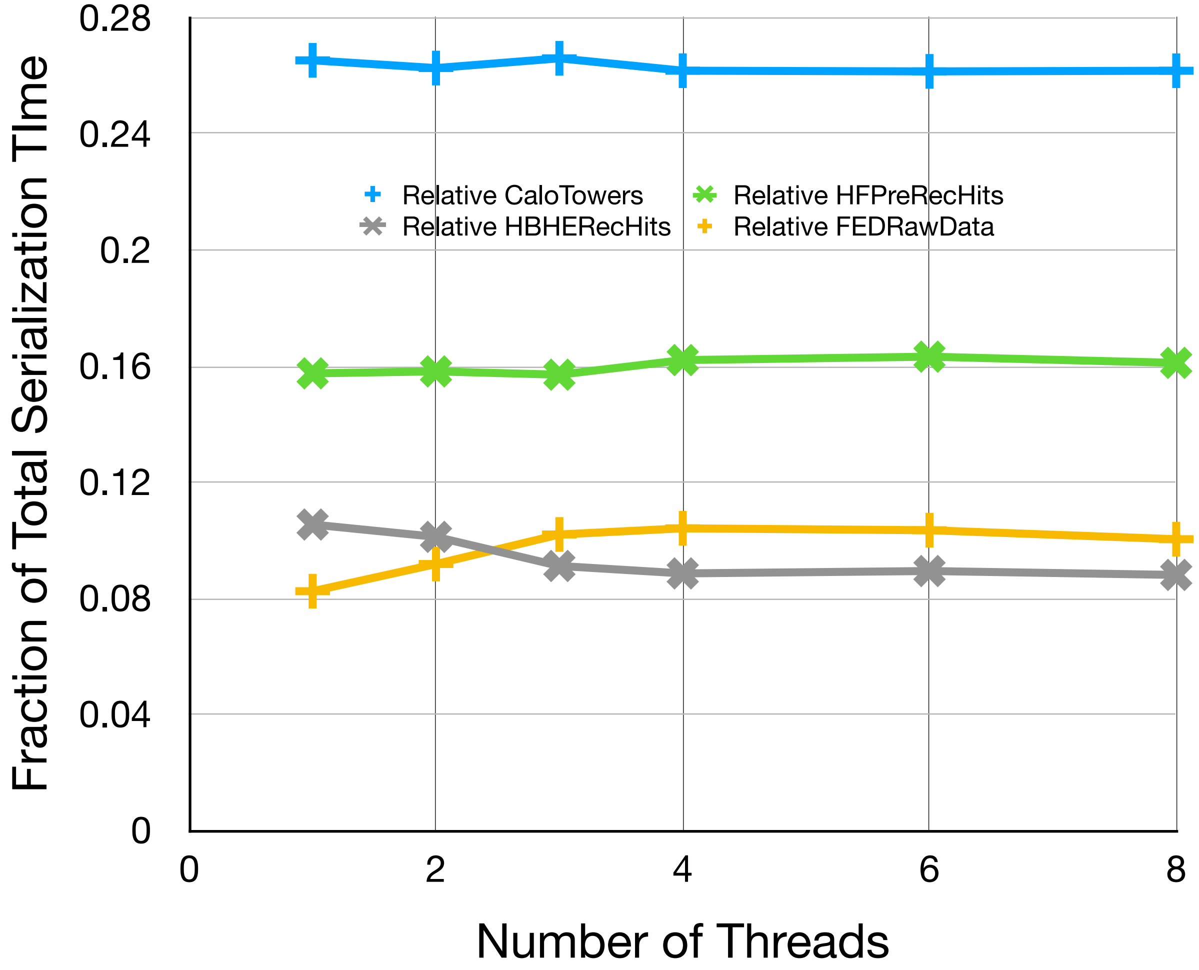
- Throughput
  - Have good throughput for 2 threads and then diminishing returns
  - Expected behavior given 4 Data Products dominate serialization time



Root Serialization with 1 Event and Multiple-Threads (cont)

- Total Serialization Time
  - No obvious trend for serialization time as a function of threads
  - Still see the different thread scaling for the 2 Data Products





Conclusion

- The testing framework seems adequate for performance measurements
  - Scaling limits are well below realistic rates

- No signs of major synchronization problems in ROOT serialization
  - Further investigation would require much higher thread counts

- For realistic CMS data files concurrent serialization provides some benefit
  - The large variations in Data Product serialization times limits the concurrency gain

- Code can be found here
  - https://github.com/Dr15Jones/root_serialization