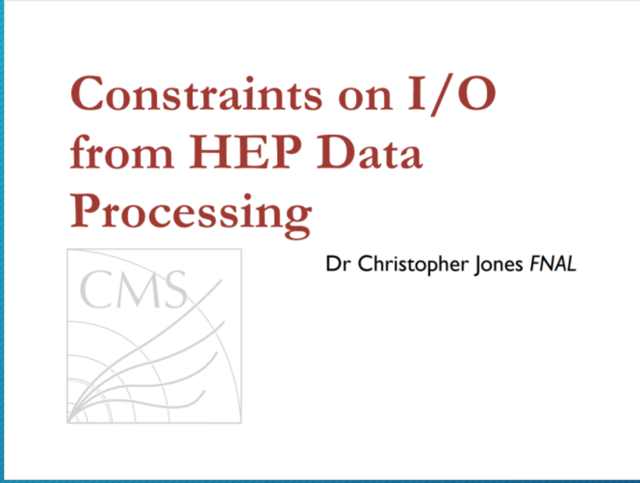

Constraints on I/O from HEP Data Processing

Dr Christopher Jones *FNAL*

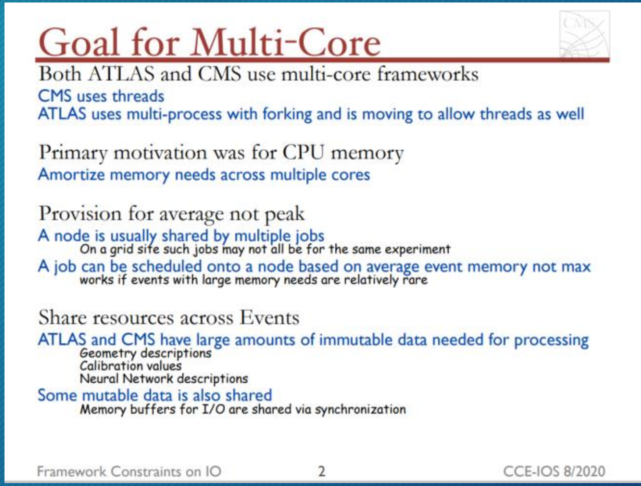
Follow Up: Processing and I/O

Peter van Gemmeren

Some ATLAS Similarities/Differences/Details

Multi Processing/Threading and Memory

- For ATLAS Reconstruction Memory is a tight resource:
  - Serial Job uses approx. 4 GB of memory
- Multiprocessing uses Copy on Write to share all common memory between processes:
    - E.g. geometry, calibration, initial condition…
    - But not: Event data, I/O buffer, data loaded after fork
  - Reduces memory to approx. 2 GB for each additional process
    - More 'tricks', Forking after processing 1 event, sharing I/O…
  - Can further reduce memory to about 1 GB, but have drawbacks.
- Multi-threading (new-ish for ATLAS) shares all memory
  - First tests show < 0.2 GB per additional thread
- Example machine 16 cores, 32 GB: cannot quite run 8 serial jobs, MP scales to all cores, but MT will enable full usage even with less memory/core or for even more memory hungry workflows (e.g. RAWtoALL).



Goal for Multi-Core

Both ATLAS and CMS use multi-core frameworks

CMS uses threads

ATLAS uses multi-process with forking and is moving to allow threads as well

Primary motivation was for CPU memory

Amortize memory needs across multiple cores

Provision for average not peak

A node is usually shared by multiple jobs

On a grid site such jobs may not all be for the same experiment

A job can be scheduled onto a node based on average event memory not max

works if events with large memory needs are relatively rare

Share resources across Events

ATLAS and CMS have large amounts of immutable data needed for processing

Geometry descriptions

Calibration values

Neural Network descriptions

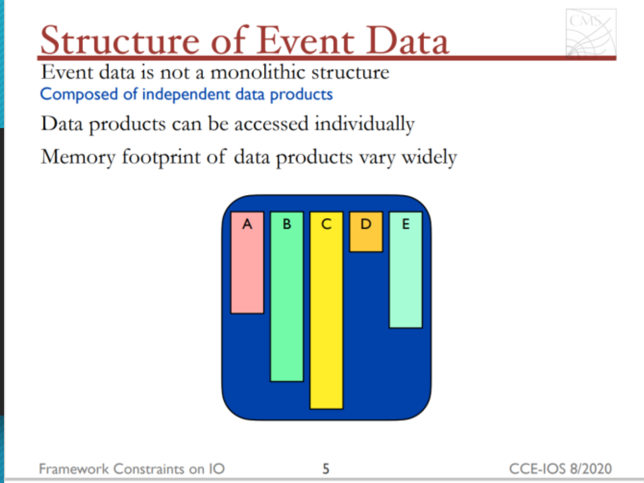
Some mutable data is also shared

Memory buffers for I/O are shared via synchronization

Framework Constraints on IO 2 CCE-IOS 8/2020

Event Data Structure, Processing

- Very similar for ATLAS
- Reading different data products (A, B, …) for the same or different events can be concurrent.
- Reading the same data product for different events is serialized:
  - typically in ROOT they are stored in the same compression basket and reading event N will decompress events N-d to N+e.
  - Attention is paid to thread requesting object from next cluster while others still processing previous data
- All algorithms on the event have to complete before writing, but no enforcement of output ordering.



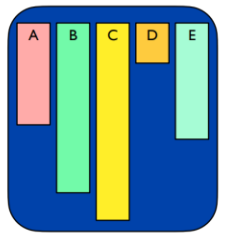
Structure of Event Data

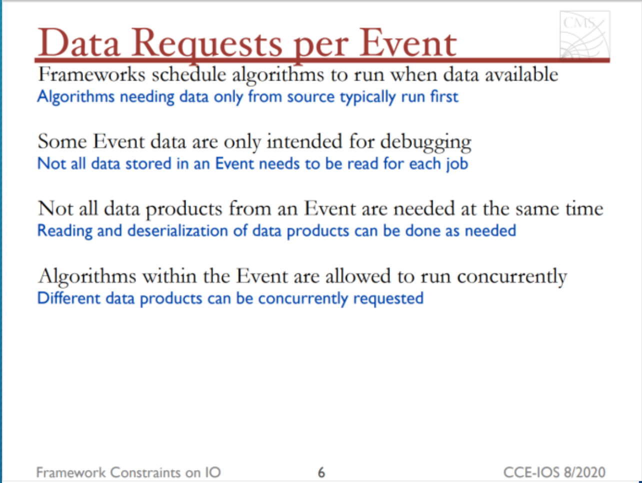
Event data is not a monolithic structure

Composed of independent data products

Data products can be accessed individually

Memory footprint of data products vary widely





Data Requests per Event

Frameworks schedule algorithms to run when data available

Algorithms needing data only from source typically run first

Some Event data are only intended for debugging

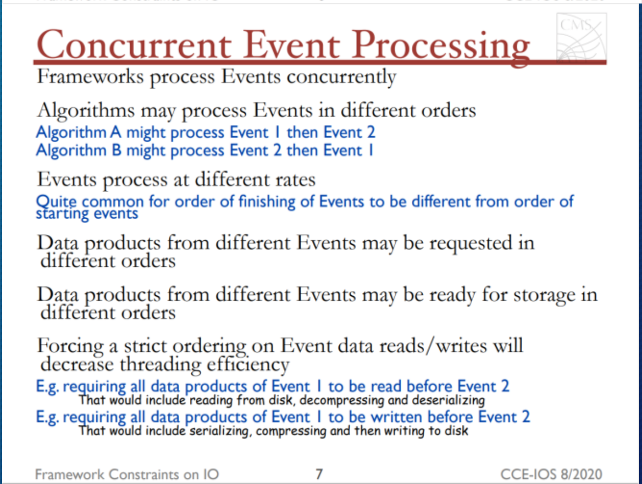
Not all data stored in an Event needs to be read for each job

Not all data products from an Event are needed at the same time

Reading and deserialization of data products can be done as needed

Algorithms within the Event are allowed to run concurrently

Different data products can be concurrently requested



Concurrent Event Processing

Frameworks process Events concurrently

Algorithms may process Events in different orders

Algorithm A might process Event 1 then Event 2

Algorithm B might process Event 2 then Event 1

Events process at different rates

Quite common for order of finishing of Events to be different from order of starting events

Data products from different Events may be requested in different orders

Data products from different Events may be ready for storage in different orders

Forcing a strict ordering on Event data reads/writes will decrease threading efficiency

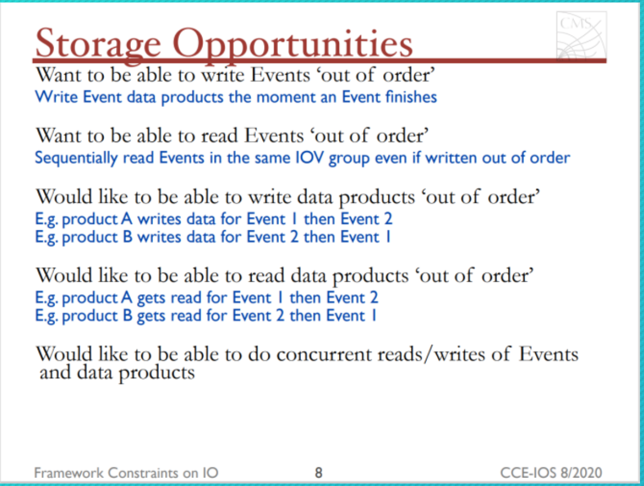
E.g. requiring all data products of Event 1 to be read before Event 2

That would include reading from disk, decompressing and deserializing

E.g. requiring all data products of Event 1 to be written before Event 2

That would include serializing, compressing and then writing to disk

Storage Opportunities



Storage Opportunities

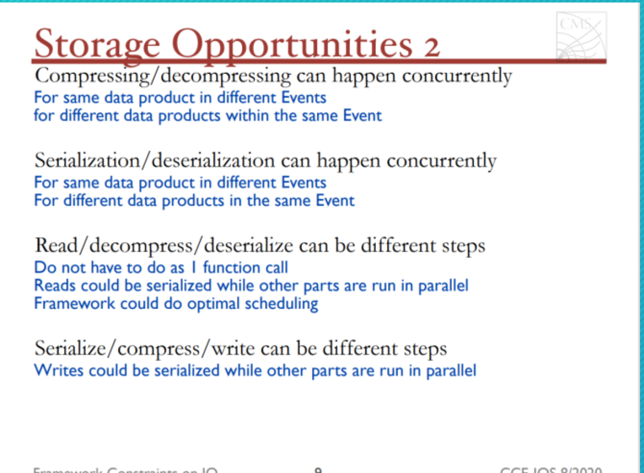
Want to be able to write Events 'out of order'
Write Event data products the moment an Event finishes

Want to be able to read Events 'out of order'
Sequentially read Events in the same IOV group even if written out of order

Would like to be able to write data products 'out of order'
E.g. product A writes data for Event 1 then Event 2
E.g. product B writes data for Event 2 then Event 1

Would like to be able to read data products 'out of order'
E.g. product A gets read for Event 1 then Event 2
E.g. product B gets read for Event 2 then Event 1

Would like to be able to do concurrent reads/writes of Events and data products



Storage Opportunities 2

Compressing/decompressing can happen concurrently
For same data product in different Events
for different data products within the same Event

Serialization/deserialization can happen concurrently
For same data product in different Events
For different data products in the same Event

Read/decompress/deserialize can be different steps
Do not have to do as 1 function call
Reads could be serialized while other parts are run in parallel
Framework could do optimal scheduling

Serialize/compress/write can be different steps
Writes could be serialized while other parts are run in parallel

- Very similar for ATLAS
- Cluster size matters for concurrent [de-]compression
- High-light:
  - Read/decompress/deserialize can be different steps
    - Do not have to do as 1 function call
    - Reads could be serialized while other parts are run in parallel
    - Framework could do optimal scheduling
  - Serialize/compress/write can be different steps
    - Writes could be serialized while other parts are run in parallel
- This would be also highly benefitial for in memory data sharing, but AFAIK is very complex.