Constraints on I/O from HEP Data Processing

Dr Christopher Jones *FNAL*

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

Interval of Validity

Data shared across Events can change

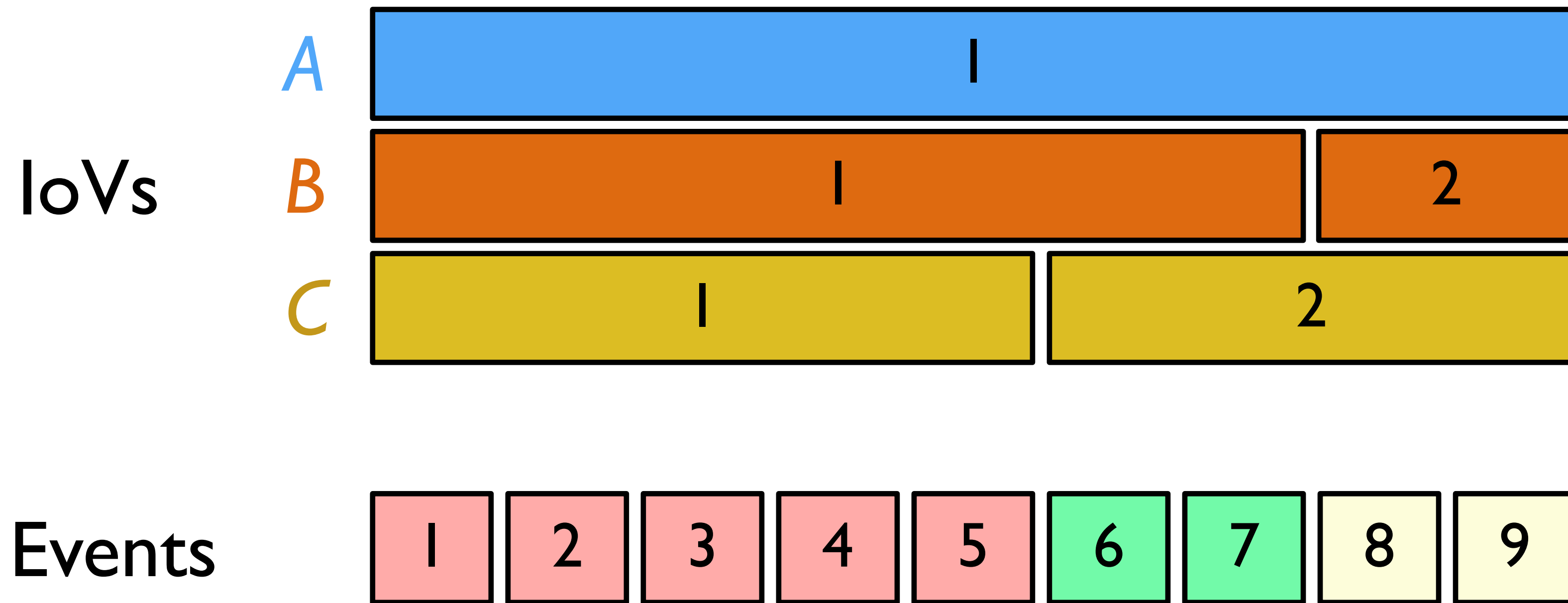
Interval of Validity (IoV)

- The range of time (i.e. span of Events) for which a given piece of data is valid

To minimize memory use want to minimize # open IoVs

- Puts a constraint on which groups of Events to process concurrently
- Within an IoV based group the processing order of the Events does not matter

IoV Example



Optimal Event processing groups based on IoVs

1-5
6-7
8-9

Order of Events in source can drive processing order

Want Events on disk from same processing group to be near each other
Alternatively low cost random access ability to read Events in best order

Structure of Event Data

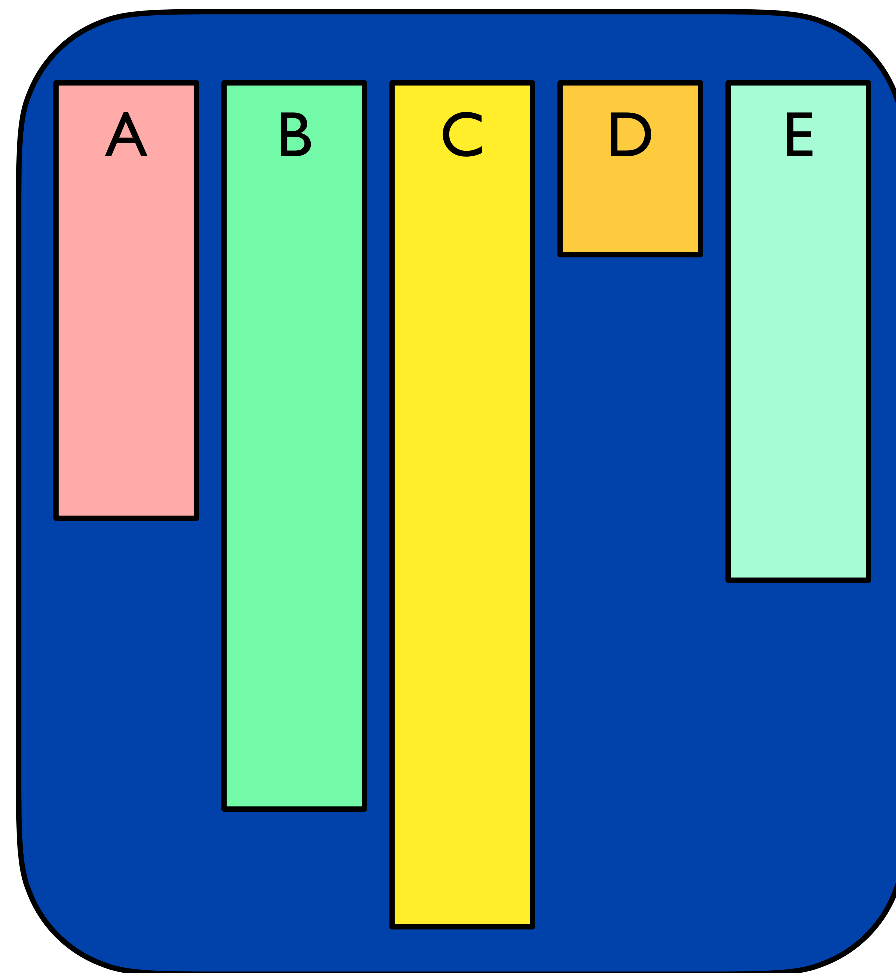
CMS

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

CMS

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

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

ROOT Storage

Stores data products mostly independently

ROOT uses the term *Branches*

The same data products for multiple Events are stored together

They are compressed together

E.g. all Tracks for a group of Events will be stored on disk contiguously

The number of Events grouped can be different for each data product

ROOT uses the term *Basket*

All data products must store Events in the same order

No data products can process next Event until all data products finish present Event

ROOT uses the term *Tree* which is a collection of related *Branches*

Data associated to a group of sequential events can be flushed

Form a *Cluster* on disk

Can random access data products

Can independently request a data product from a particular Event