Future Computing Challenges for ATLAS

Charles Leggett

May 19 2016

Atlas Core Software Meeting / HPX

A Little History

- Athena: ATLAS event processing framework
  - an extension of GAUDI, which is shared by several experiments

- Athena is ~15 years old
  - serial processing design
  - future computing needs were expected to be met by faster clock rates
  - relatively small memory footprint

- Since then, clock rates have stalled, CPU core counts have exploded, memory prices have plateaued
  - ratio of mem / core has not kept up with ATLAS computing needs
  - memory usage for reconstruction have crept upwards to ~3GB
    - Run 3 / High luminosity will likely dramatically increase this

AthenaMP : Multi-Process Concurrency

- ATLAS has been able to address the problem for now via "trivial" event level parallelism
  - after initialization, mother process forks children, each child processes a separate event (serially)
  - significant memory savings via linux COW
    - large amounts of data are static after initialization, eg detector description, and can be shared by all the child processes
- Virtually no modifications required to user level code base
- Ultimately, while this works on multi-core machines, it's not good enough for many-core architectures.
  - Knights Landing, ARM, Tesla K80, GPU, etc
  - make use of large facilities such as Cori at LBL

Multi-Threaded Concurrency

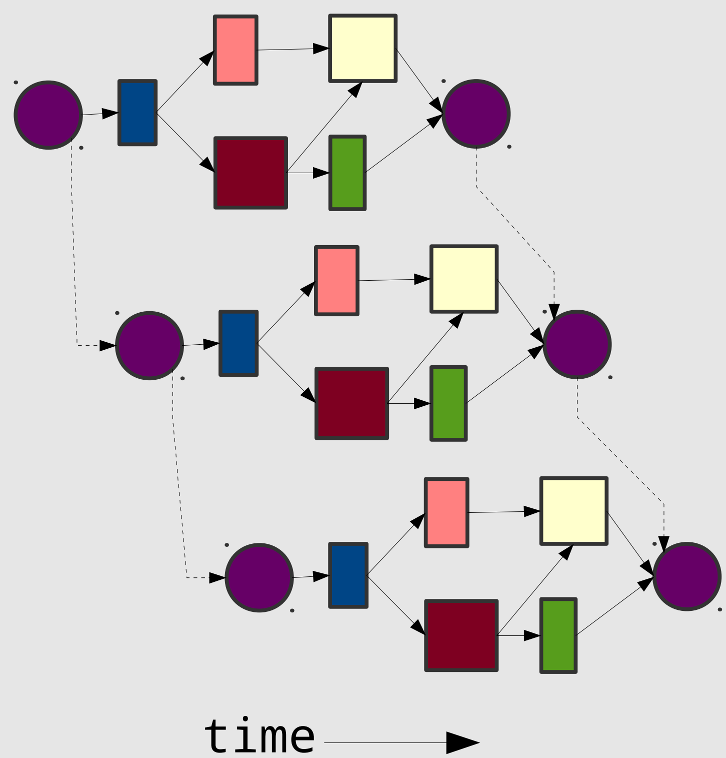
- In light of future hardware trends, and expected event processing requirements, decision was made to investigate a multi-threaded approach

- Thread Safety:
  - Athena was never designed to be thread safe
  - Need to be backward compatible: can't start over from scratch
    - don't have time/manpower to re-write millions of lines of legacy code
    - will need to be able to analyze Run 1 and 2 data for a long time - overhead and validation of 2 separate code bases impossible
  - Need to address as many thread safety issues at the framework level as possible, to shield end users from threading concepts

  - Thread safe programming is HARD. Can't expect the average physicist to be able to do it correctly.
    - but have to be flexible enough to enable thread access for those who understand the risks and need the benefits
- Leverage existing work on Gaudi Hive

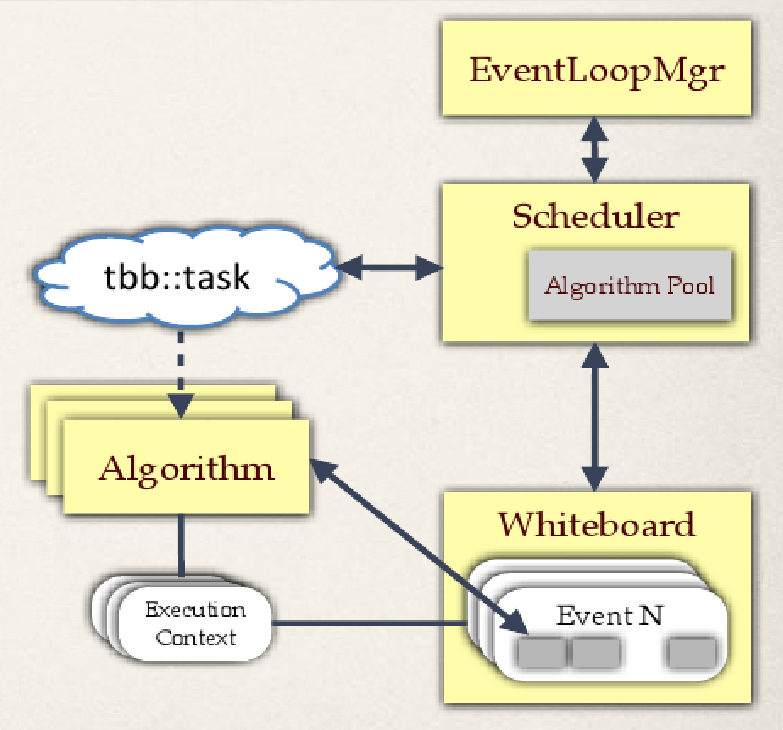
AthenaMT / Gaudi Hive

- Gaudi Hive: multi-threaded, concurrent extension to Gaudi
  - backward compatible, uses Intel tbb for thread management
- Data Flow driven
  - Algorithms declare their data dependencies
    - build a directed acyclic graph - can be used for optimal scheduling
  - Scheduler automatically executes Algorithms as data becomes available.
- Multi-threaded
  - Algorithms process events in their own thread, from a shared Thread Pool.
- Pipelining: multiple algorithms and events can be executed simultaneously
  - some Algorithms are long, and produce data that many others need (eg track fitting). instead of waiting for it to finish, and idling processor, start a new event.
- Algorithm Cloning
  - multiple instances of the same Algorithm can exist, and be executed concurrently, each with different Event Context.
  - cloning is not obligatory, balancing memory usage with concurrency.
  - support for re-entrant Algorithms



AthenaMT / Gaudi Hive Operation

- Configuration, Initialization, Finalization are performed serially in "master" thread
  - only `Algorithm::execute` is concurrent
- Algorithms are scheduled when data becomes available
  - Algorithms must declare their inputs at initialization or dynamically with DataHandles
  - data only exchanged via whiteboard
  - tbb::task wraps the pair (Algorithm*, EventContext)
- Algorithms can be non-cloneable (singleton), cloneable, or re-entrant
  - more clones = more memory, but greater opportunity for concurrency
    - cardinality is tunable at runtime
  - re-entrant is best, but hardest to code



- tbb layer is normally hidden from users, but Algorithms can explicitly use tbb constructs (*parallel_for*, *concurrent_queue, etc)* for finer grained parallelism
  - plays well with the Scheduler
- Component model allows Scheduler to be replaced as needed

Current AthenaMT Status

- Several sub-detector testbeds have been implemented, and show very promising performance/memory saving results

- Integrated into our regular code base
  - separate AthenaMT nightly build to enable certain features

- Some user level code changes will still have to be made
  - data dependency declaration through use of Data Handles
  - all shared components (Services) must be thread safe
    - will generally be done by "experts" and not "regular" users
  - some modifications to access patterns of data in the Event Store
  - user education required to avoid and remove "thread hostile" code
    - many dangerous patterns can be caught early via static analyzers and gcc compiler plugins