Introduction to Gaudi and Athena

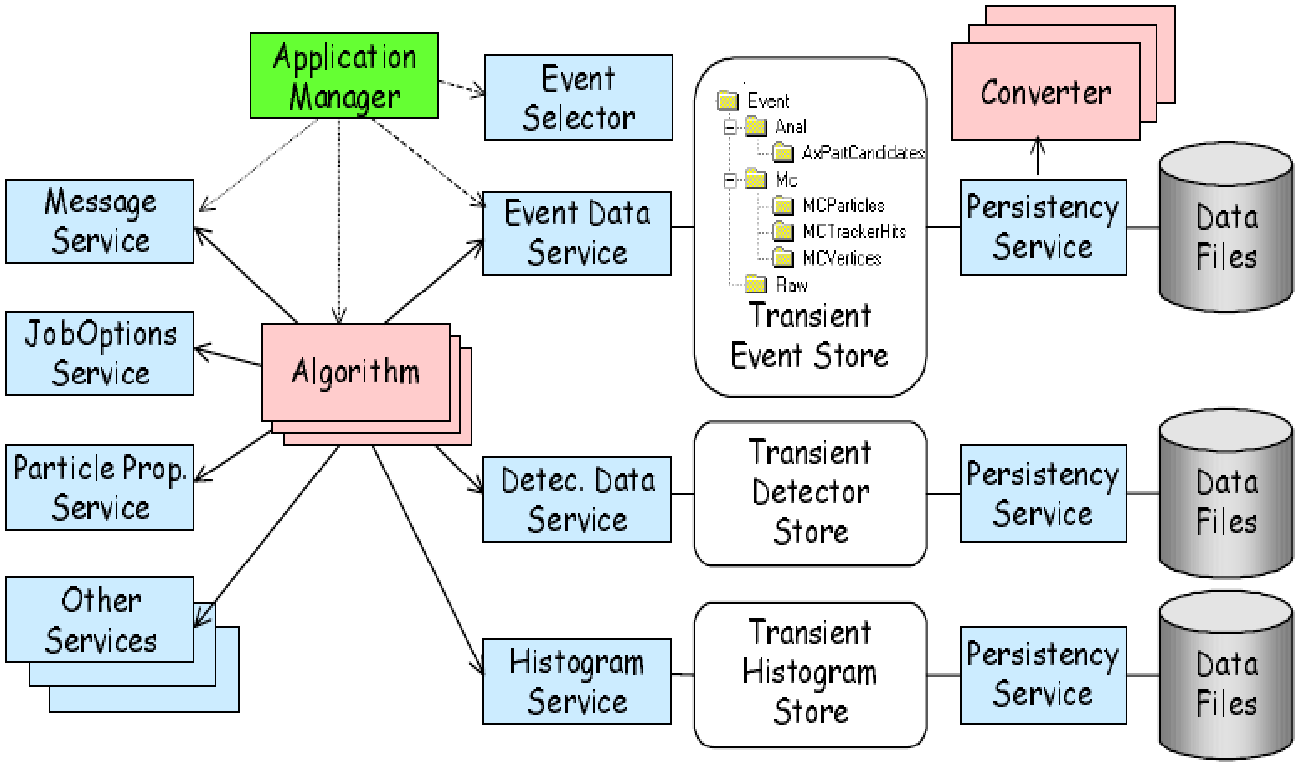
Vakho Tsulaia (LBL)

*Based on the presentations given by **Charles Leggett (LBL)** and **Paolo Calafiura (LBL)** in earlier editions of the ATLAS Software Development Tutorial*

ATLAS Software Development Tutorial
CERN, September 23-27, 2019

Building blocks of Gaudi

- **Algorithm**
  - Main building block of the **Event Loop**
  - Called once per event
- **AlgTool**
  - A plugin that helps an Algorithm perform some action
- **Service**
  - A plugin providing a common service to multiple components
  - **Examples:** Transient Data Store, Logging Service, Random Number Service



Interfaces, Plug-ins, Factories, etc.

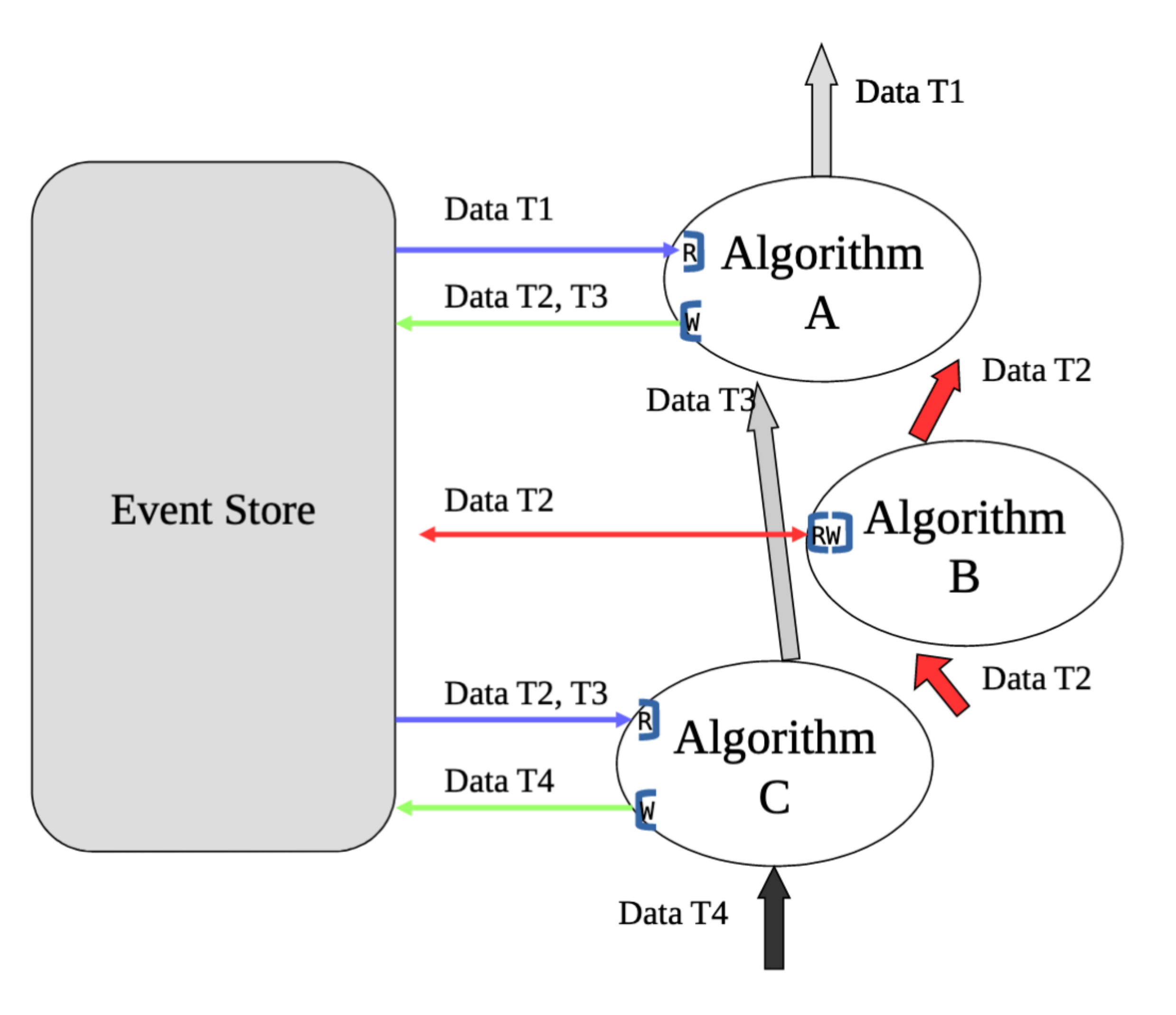
- Gaudi component model originally inspired by Microsoft COM
- Components implement **an interface** and use other components through **an interface**
- Components get packaged into Shared Object Libraries (DSO) and declared in a special component manifest files
- A dedicated Gaudi Service (**PluginSvc**) uses manifest to locate which DSO contains requested component dictionary, dl-opens it and creates an instance using a factory method

Job execution stages (simplified)

- **Configuration**
  - Parsing of configuration scripts (**Job Options**)
  - *More on configuration later this week*
- **Initialization**
  - `::initialize()` methods of components (Services, Algorithms) get called
- **Event processing loop**
  - `::execute()` methods of Algorithms get called
- **Finalization**
  - `::finalize()` methods of components (Services, Algorithms) get called

Event processing in serial Athena

- Algorithms run in sequences defined at the configuration time
- Algorithms consume (read) and produce (write) data objects shared via an Event Store
- Objects in the Event Store define data dependencies between algorithms
- Algorithms don’t call each others methods



Status codes

- `StatusCode` is a standard return code used by Athena / Gaudi components
- Values are `StatusCode::SUCCESS` and `StatusCode::FAILURE`
- **Must be checked** on return of function. Unchecked `StatusCode` results in termination of the Athena job

```
StatusCode sc;
sc = SomeFunction(par);
If ( sc.isFailure() ) {
  ATH_MSG_FATAL("SomeFunction failed for par : " << par);
  return sc;
}

ATH_CHECK(OtherFunction(par));
```

Check StatusCode

Recommended way of checking StatusCode

Data access in Athena

- `StoreGateSvc` – **Transient Object Store** implementation in Athena
  - Manages data objects in memory (owns them)
  - Provides interface for **recording** and **retrieving** objects to/from the object store
  - Type-centric Naming Service. Each object has a **unique key**
- Several instances of Store Gate in typical Athena job
- Event Data:
  - **Event Store**
    - Contains data corresponding to one event. Gets cleared after every event
- Non-Event Data
  - **Detector Store** (immutable data, e.g. detector geometry)
  - **Metadata Store** (in-file metadata)
  - **Condition Store** (time-varying calibration data retrieved from Condition Database)

Data Handles

- Athena components access data in the Event Store via **Smart Handles**
- Two types of handles are defined for general use: **ReadHandle** and **WriteHandle**
- The handles are **templated** on the type of object being referenced
- They hold the corresponding **StoreGate ke**y
- They act like pointers to the referenced data, caching the pointer to the object in the Event Store

Usage of Handles

- Declare corresponding **Handle Keys** as private data members of your Algorithm/AlgTool

```
class MyAlg : public AthAlgorithm {
…
private:
   SG::ReadHandleKey<MyObj1>  m_myObj1Key {this, ”MyObj1", ”DefaultKey1", ”MyObj1 Handle Key"};
   SG::WriteHandleKey<MyObj2> m_myObj2Key {this, ”MyObj2", ”DefaultKey2”, ”MyObj2 Handle Key"};
```

Default value

Property name

- This automatically sets up handle key properties of the Algorithm, which can be set in job options

```
topseq += MyAlg (..., myObj1 = ‘OtherKey1’)
Topseq.MyAlg.MyObj2 = ’OtherKey2’
```

Usage of Handles (contd.)

- Initialize your handle keys from the algorithm's `initialize()` method

```
StatusCode MyAlg::initialize() {
…
 ATH_CHECK( m_myObj1Key.initialize() );
 ATH_CHECK( m_myObj2Key.initialize() );
```

- To use the handles in `execute()` of your algorithm, create handle objects as local variables, initializing them from the keys

```
StatusCode MyAlg::execute() {
…
 SG::ReadHandle<MyObj1> myObj1 (m_myObj1Key);
 SG::WriteHandle<MyObj2> myObj2 (m_myObj2Key);
```

Usage of Handles (contd.)

- To record the object to the store:

```
StatusCode MyAlg::execute() {
…
 SG::WriteHandle<MyObj2> myObj2 (m_myObj2Key);
 ATH_CHECK( myObj2.record (std::make_unique<MyObj2>()) );
```

- For in-depth documentation of the data access in Athena, see the [“Event data access in AthenaMT”]() TWiki page

Accessing components

- Algorithms/AlgTools access Services and AlgTools via **Handles**

- Declare handles as data members of your Algorithm/AlgTool (ToolHandle **must** be private!)

```
class MyAlg : public AthAlgorithm {
…
private:
   ToolHandle<IHiveTool>       m_hiveTool {this, "PrivateTool", "PriHiveTool", ”private tool"};
   ServiceHandle<IGeoModelSvc> m_geoModelSvc {this, "GeoModelSvc", "GeoModelSvc"};
```

- Retrieve them at initialize …

```
StatusCode MyAlg::initialize() {
…
   ATH_CHECK(m_hiveTool.retrieve());
   ATH_CHECK(m_geoModelSvc.retrieve());
```

… and then treat them like pointers

```
m_hiveTool->methodA();
…
m_geoModelSvc->methodB();
```

Configuring a component

- In order to be configurable via the Configuration layer, an Algorithm/AlgTool must **declare one or several properties**

Property name

Default value

Doc string

```
class MyAlg : public AthAlgorithm {
…
private:
   Gaudi::Property<bool>  m_flag  {this, “Flag”,  true, “some description”};
   Gaudi::Property<float> m_float {this, “Float”, 3.14, “looks like PI”};
```

- You can set the property values in **job options** script

```
topseq += MyAlg (..., Flag = False)
Topseq.MyAlg.Float = 3.1416
```

Const-correctness of ToolHandles

- In general, you should think of ToolHandles as objects, not pointer to objects, even though you will access the methods of the Tool with pointer-like semanthics.
- ToolHandles are preserving constness: **it is not possible to call a non-const method of a Tool from a const ToolHandle**
- The following will generate a compile time error
  - If a class has a ToolHandle as a data member, calling a non-const method of the Tool from a const function of the class
  - Iterating over a ToolHandleArray with a const_iterator, and calling non-const methods
  - Extracting a `T*` pointer from a `const ToolHandle<T>`

Run Athena

- Once you have your job options script ready, the next step is to run the job:

```
$ athena.py [OPTIONS] jobOptions.py
```

… before doing that, though, first you need to setup runtime 😬 (see next slides) …

- This is **not** how we run Athena in production
- For the production jobs we use special wrapper scripts called **Job Transforms**
  - Provide interface to the production system components (e.g. **Pilot**)
  - One transform can launch either a single Athena job or a chain of jobs, each implementing different data processing stages (e.g. digitization, reconstruction, trigger simulation)
  - **Not covered in this tutorial**

Releases

- Athena repository in GitLab has number of branches
  - master – main development branch, now leading to 22.0.X series of releases
  - 21.0, 21.1, 21.2, 21.3 – several production release branches
- For several branches (e.g. master, 21.X) we automatically build nightly releases
  - Each nightly build has corresponding Git tag
    Example: `nightly/master/2019-09-21T2133`
  - Nightly builds are installed on CVMFS and kept around for ~1 month
- Status of nightly builds can be monitored here
- Once there is a need for that, we build **numbered releases** of a particular branch
  Examples: 21.0.100, 22.0.4
  - Each of these releases has a corresponding tag in GitLab
  - Unlike nightly builds, the numbered releases are there to stay "forever"

Runtime

- Once you start a new session, say, on `lxplus`, first thing you usually do is to run this command

```
$ setupATLAS
```

- After that it is recommended to pick up “right” version of git

```
$ lsetup git
```

- Now you are all set to setup runtime against some recent nightly release

```
$ asetup Athena,master,r21
```

Nightly

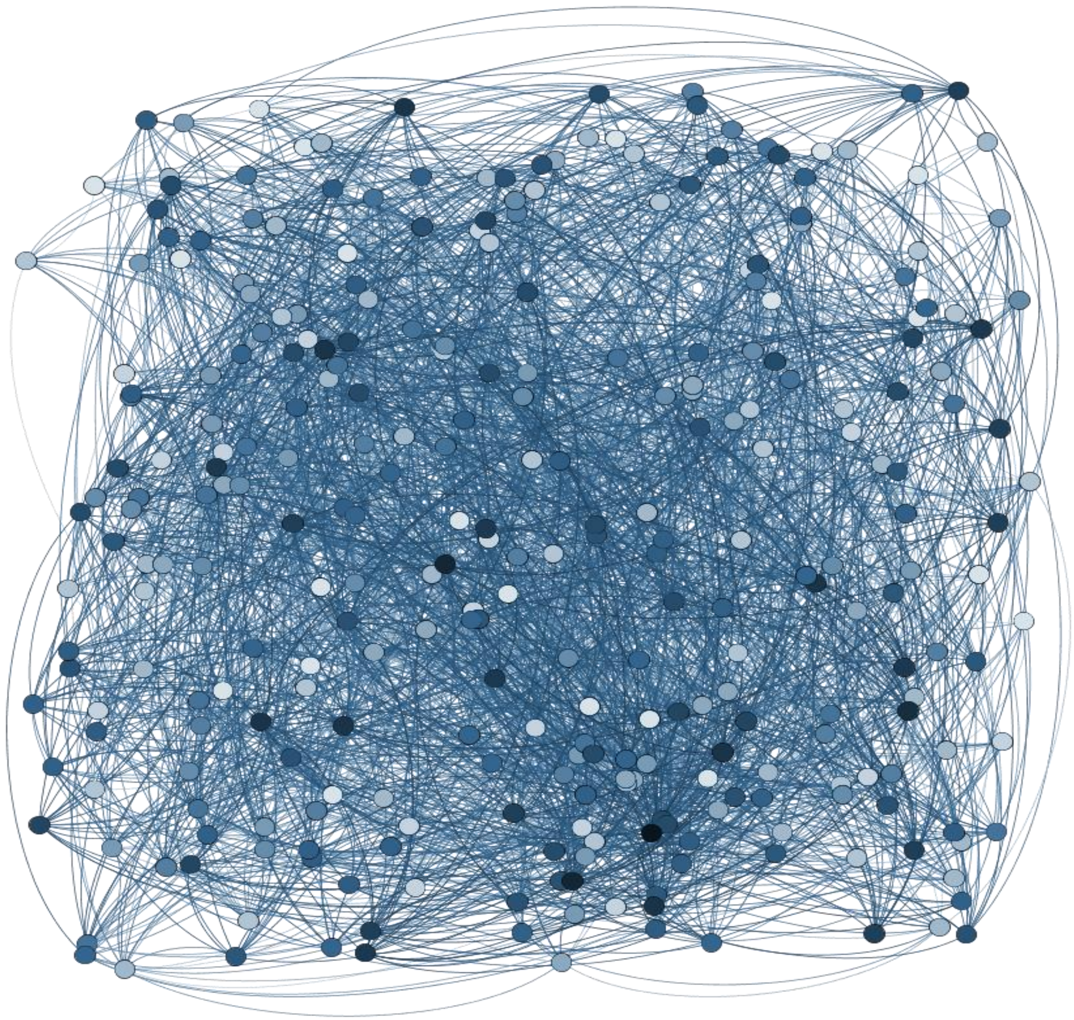
Branch

Project

- … or some numbered release

```
$ asetup Athena,22.0.4
```

Athena MT



Gaudi MT

Parallel programming models
Problem decomposition viewpoint

Framework overhead

- Task-parallel model
  - Leverages **intra**-event concurrency
- Data-parallel model
  - Leverages **inter**-event concurrency

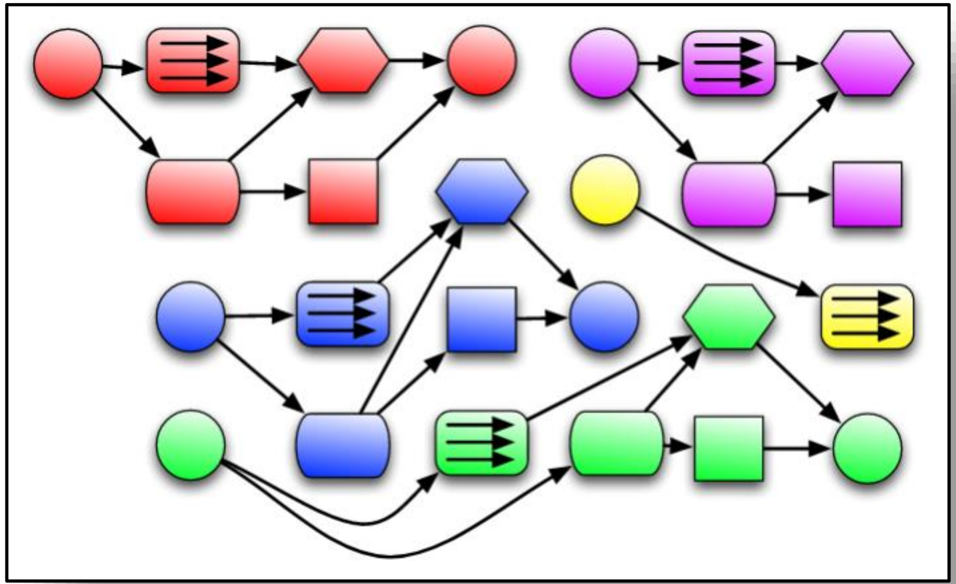
Task-based programming
An abstraction of hardware architecture

- Faster task startup and shutdown
  - tthread/ttask = 18 onLinux (*)
  - tthread/ttask = 100 onWindows (*)
- Better load balancing
- Favors structured work partitioning and task precedence management
- Matching parallelism to resources
- Matching tasks to resources

(*) According to Intel Threading Building Blocks Team
tthread – time to launch a thread

AthenaMT

- Multi-threaded successor of Athena, designed to make better use of existing and upcoming hardware, while minimizing the memory footprint
- Uses the same component model as serial Athena with the same states (initialize, execute, finalize)
- Data flow driven: algorithms declare what data they consume and produce
- Each `Algorithm::execute()` happens in its own thread from shared thread pool
- Event pipelining: multiple event can be executed concurrently
- Algorithm cloning: multiple instances of the same algorithm can execute with different event contexts



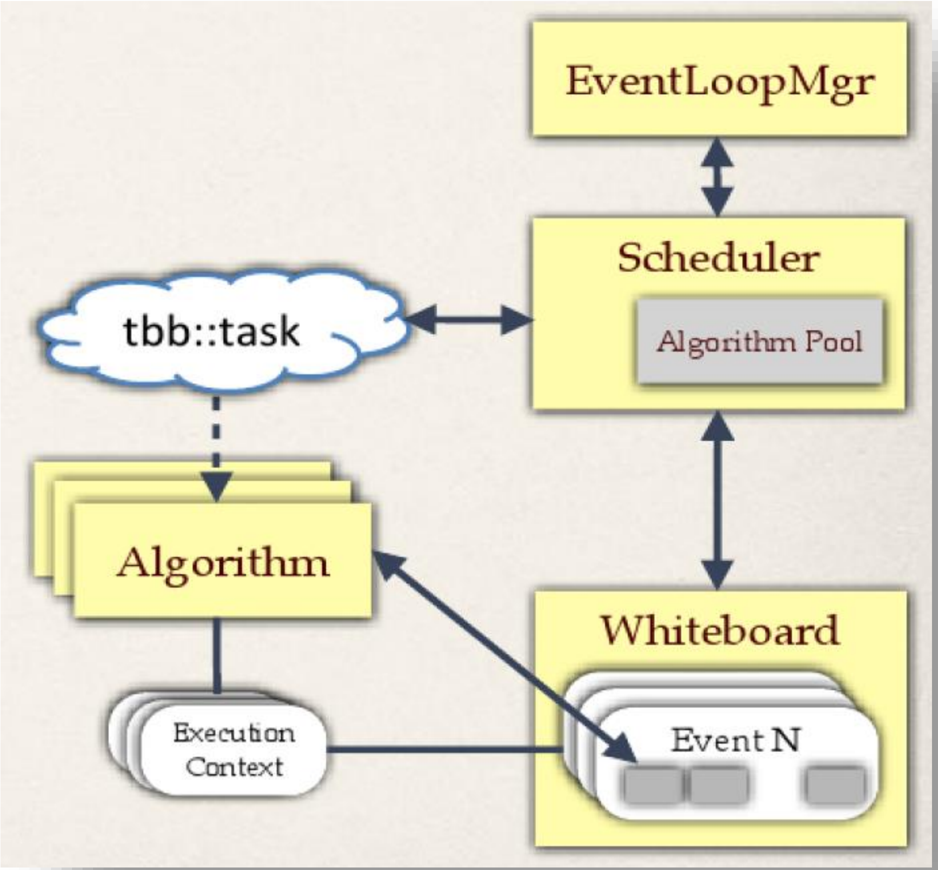
Event processing:
- each event is a different color
- Each shape is a different algorithm

AthenaMT implementation

- AthenaMT uses **Intel Threaded Building Blocks (TBB)** for thread management
  - C++ templates and generic programing
  - Abstracts threading model by allowing operations to be treated as **tasks**
  - Supports **task stealing**: tasks on busy cores automatically reassigned to less busy ones
  - Library of algorithms and data structures that ease use of threads
    - `parallel_for, parallel_reduce, parallel_scan, parallel_do`
    - `concurrent_hash_map, concurrent_vector, concurrent_queue`
- While the TBB layer is hidden from Athena users, they can still use it for finer grained parallelism within Algorithms
  - In general not recommended

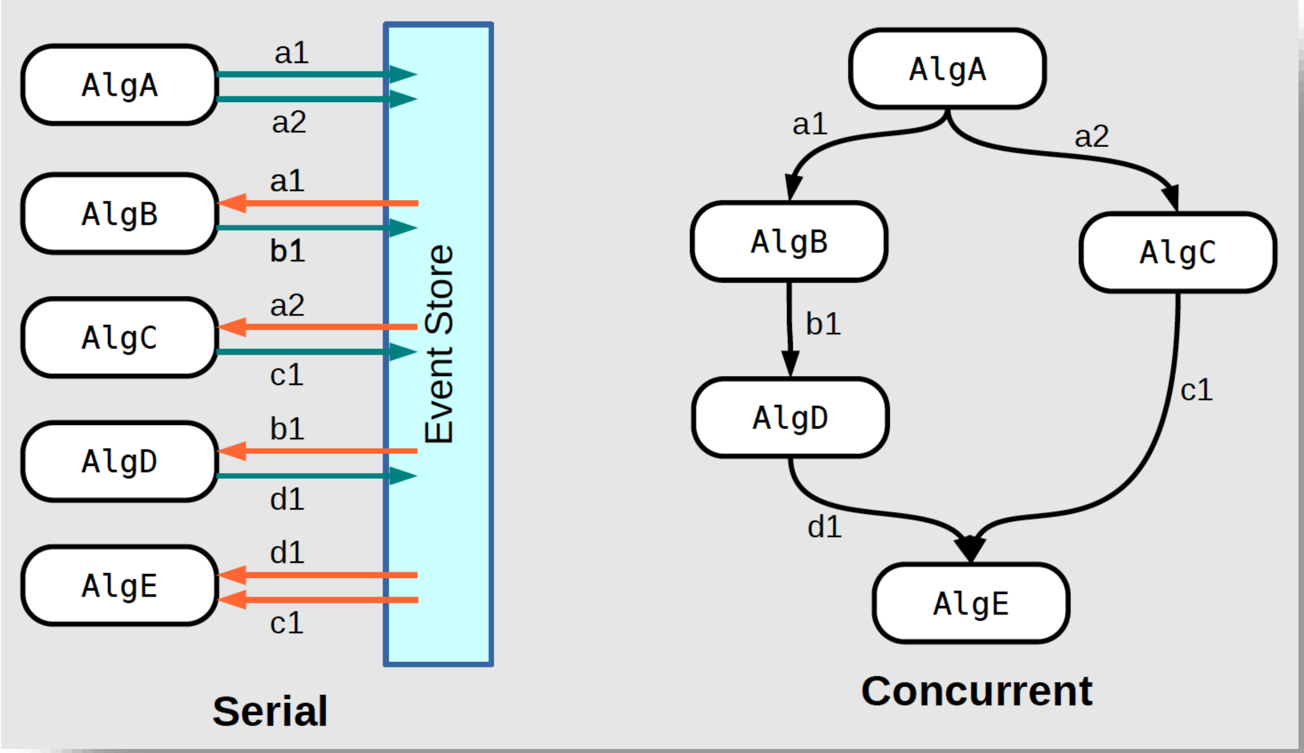
Athena MT operation

- Configuration, Initialization and Finalization are performed serially in the “master” thread
  - Only `Algorithm::execute()` is concurrent
- Algorithms are only scheduled when their input data becomes available
- Several instances of the same Algorithm can coexist
  - **Cloning:** create new instance if can be scheduled, and all other instances are busy
- Multiple events can be executed concurrently
  - Separate instance of Data Store per concurrent event

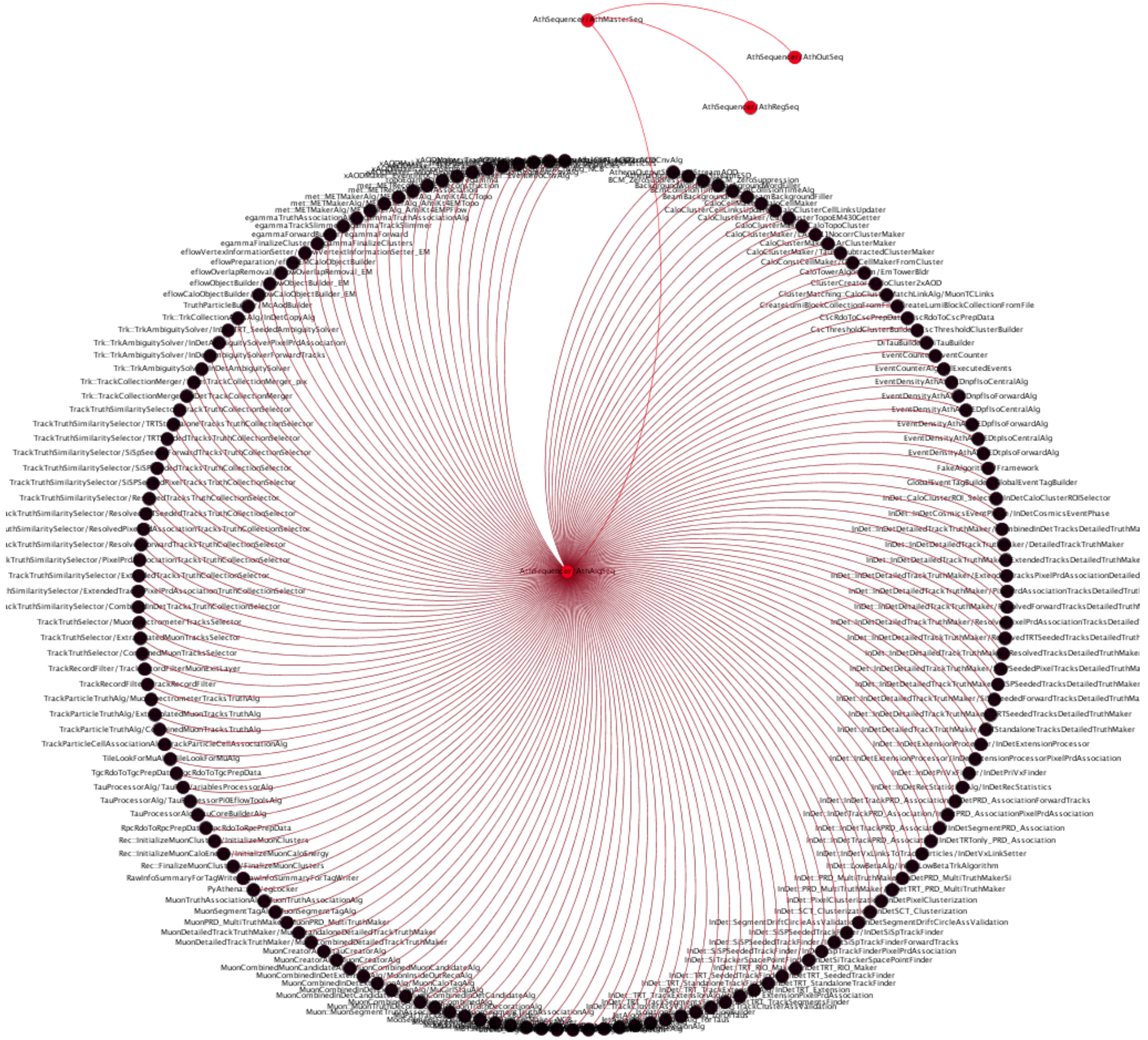


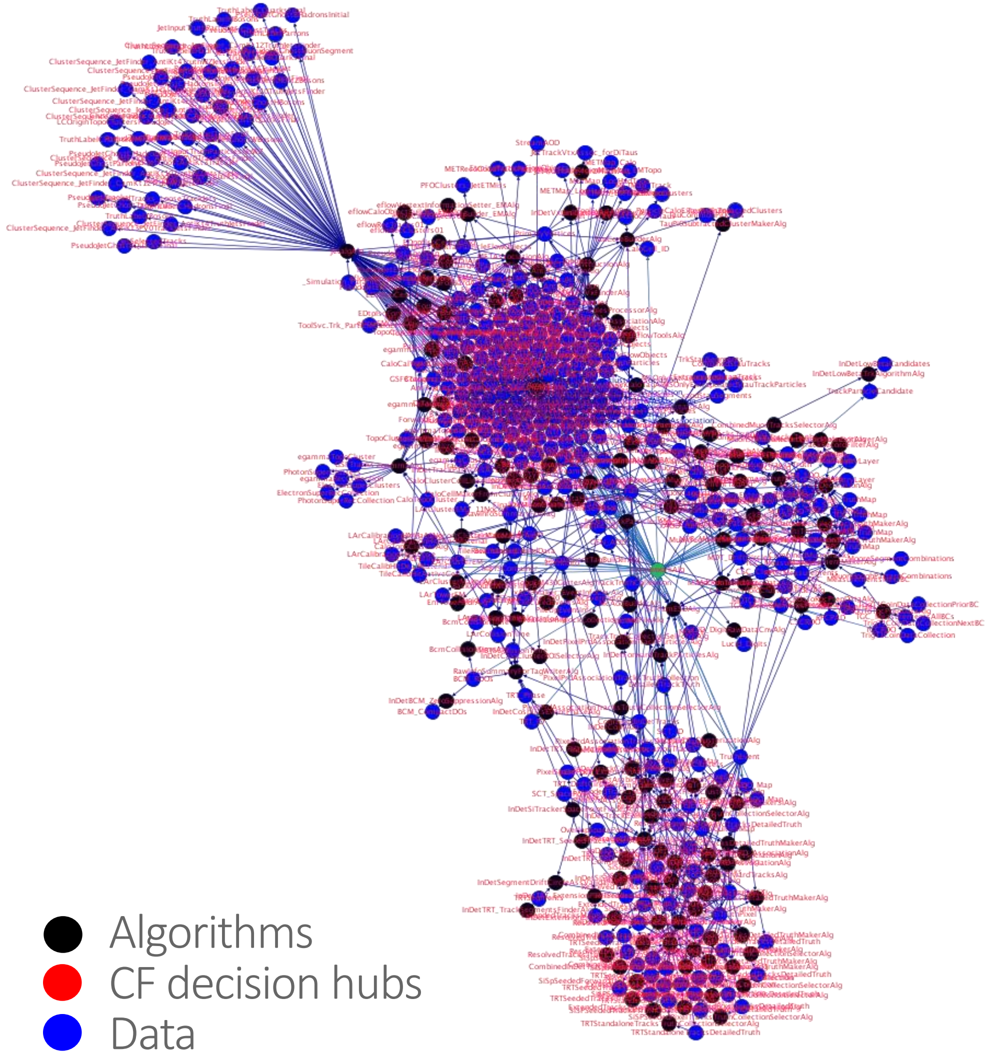
Scheduler

- Framework component that executes an Algorithm in an unused thread at the appropriate time
- Number of ways to decide which Algorithm to execute next
  - Precedence rules are created from a combination of **Data Flow** and **Control Flow**
  - Algorithms can be scheduled from any event in the queue
- Additional information can be provided to the Scheduler to help it determine most efficient way to get to the end



ATLAS MC data reconstruction precedence rules: CF (left) and DF (right)





Algorithms
CF decision hubs
Data

EventContext

- An object which provides information about the currently executed event
- Is set by the Event Loop Manager / Scheduler, and pushed into the Algorithm

```
#include “GaudiKernel/EventContext.h”

class EventContext {
public:
 size_t evt() const;
 size_t slot() const;
 bool valid() const;
 const EventIDBase& eventID() const;
 const T* getExtension() const;
}
```

Number of events executed in job

Index of concurrent slot

Context is valid only within Algorithm::execute

Event#, Run#, LB#, Timestamp

Extendable content

Algorithm cloneability

- **Non-cloneable**: default
  - Only one instance exist in Algorithm Pool
  - Shared between all events. The Scheduler will **block** if algorithm needs to run on some event, but is in use in another event
  - Low memory use
- **Cloneable:**
  - Enabled explicitly by user via `MyAlg::isCloneable() { return true;}`
  - No clones created by default. Extra clones must be specifically requested by user
  - High memory use
- **Reentrant** (special base class `AthReentrantAlgorithm`)**:**
  - **One instance,** shared by all threads
  - `execute()` is const and signature includes `EventContext`
  - Scheduler will **not block**, but use same instance for all threads
  - **Must be thread safe**

Component requirements

- **Algorithms**:
  - Must declare data dependencies (see later in this talk).
    - Data dependencies are union of Algorithm's and used AlgTools'
  - Communicate data to other Algorithms **only** via Whiteboard
  - **Must not cache** data between events
  - Should limit cloning to longer Algorithms, that are more likely to execute simultaneously
- **AlgTools** must be **stateless** and **private**
  - Must declare data dependencies
  - **Must not cache** data between events, unless in an `EventContext`-aware fashion
  - Can have event related data passed to them via accessors
  - Can access Whiteboard
- **Services** must be **thread safe** and **stateless**
  - May need to be made aware of the `EventContext`

Declaring data dependencies

- Algorithms (and AlgTools) must declare their Input and Output data dependencies to the framework
- Algorithms inherit the data dependencies of their sub-algorithms and Tools
  - Tools must be declared as `ToolHandles` by the Algorithms
- Data should be declared in the form of `VarHandleKeys`, using an initializer list syntax in the header of the Algorithm/AlgTool

```
class MyAlg : public AthAlgorithm {
…
private:
   SG::ReadHandleKey<MyObj1>  m_myObj1Key {this, "MyObj1", "DefaultKey1", "MyObj1 Handle Key"};
   SG::WriteHandleKey<MyObj2> m_myObj2Key {this, "MyObj2", "DefaultKey2", "MyObj2 Handle Key"};
```

Examining job configuration

- You can dump Control Flow and Data Flow configuration of the job into log fie

```
from AthenaCommon.AlgScheduler import AlgScheduler
AlgScheduler.ShowControlFlow( True )
AlgScheduler.ShowDataDependencies( True )
```

Examining job configuration (contd.)

- Data dependencies

```
AvalancheSchedulerSvc                              0      INFO Data Dependencies for Algorithms:
  BeginIncFiringAlg
      none
  IncidentProcAlg1
      none
  xAODMaker::EventInfoCnvAlg
    o INPUT   ( 'EventInfo' , 'StoreGateSvc+McEventInfo' )
    o OUTPUT  ( 'SG::AuxElement' , 'StoreGateSvc+EventInfo' )
    o OUTPUT  ( 'SG::AuxVectorBase' , 'StoreGateSvc+PileupEventInfo' )
    o OUTPUT  ( 'xAOD::EventInfo' , 'StoreGateSvc+EventInfo' )
    o OUTPUT  ( 'xAOD::EventInfoContainer' , 'StoreGateSvc+PileupEventInfo' )
  SGInputLoader
      none
  HiveAlgA
    o INPUT   ( 'HiveDataObj' , 'StoreGateSvc+b2' )
    o INPUT   ( 'xAOD::EventInfo' , 'StoreGateSvc+EventInfo' )
    o OUTPUT  ( 'HiveDataObj' , 'StoreGateSvc+a1' )
    o OUTPUT  ( 'HiveDataObj' , 'StoreGateSvc+a2' )
  HiveAlgB
    o OUTPUT  ( 'HiveDataObj' , 'StoreGateSvc+b1' )
    o OUTPUT  ( 'HiveDataObj' , 'StoreGateSvc+b2' )
  HiveAlgC
    o INPUT   ( 'HiveDataObj' , 'StoreGateSvc+a1' )
    o OUTPUT  ( 'HiveDataObj' , 'StoreGateSvc+C1' )
    o OUTPUT  ( 'HiveDataObj' , 'StoreGateSvc+c2' )
```

Examining job configuration (contd.)

- Control Flow

```
==================== Control Flow Configuration ==================

  AthMasterSeq [Seq]  [Sequential]  [Prompt]
    AthAlgEvtSeq [Seq]  [Sequential]
      AthBeginSeq [Seq]  [Sequential]  [Prompt]
        BeginIncFiringAlg [Alg]  [n= 0]
        IncidentProcAlg1 [Alg]  [n= 1] [unclonable]
      AthAllAlgSeq [Seq]  [Concurrent]
        AthAlgSeq [Seq]  [Concurrent]  [PASS]
          xAODMaker::EventInfoCnvAlg [Alg]  [n= 0]
          SGInputLoader [Alg]  [n= 4]
          HiveAlgA [Alg]  [n= 4]
          HiveAlgB [Alg]  [n= 4]
          HiveAlgC [Alg]  [n= 4]
          HiveAlgD [Alg]  [n= 4]
          HiveAlgE [Alg]  [n= 4]
          HiveAlgG [Alg]  [n= 4]
          HiveAlgF [Alg]  [n= 4]
          HiveAlgV [Alg]  [n= 4]
        AthCondSeq [Seq]  [Concurrent]
      AthEndSeq [Seq]  [Sequential]  [Prompt]
        EndIncFiringAlg [Alg]  [n= 0]
        IncidentProcAlg2 [Alg]  [n= 1] [unclonable]
    AthOutSeq [Seq]  [Concurrent]
    AthRegSeq [Seq]  [Concurrent]
```

Logging

- MessageSvc - a service used by Athena for logging purposes
- In AthenaMT, MessageSvc queues messages from each source
  - Atomic output, so log is readable
- By default you'll get slot# and event# (only if there is a valid EventContext)
  - MessageSvc supports several flags for output formatting (not covered in this talk)

```
AthenaHiveEventLoopMgr          3 3 INFO ===>>> start processing event #3, run #1 on slot 3, 0 events processed so far <<<===
AthenaHiveEventLoopMgr              INFO ===>>>  done processing event #0, run #1 on slot 0, 1 events processed so far <<<===
AthenaHiveEventLoopMgr          4 4 INFO ===>>> start processing event #4, run #1 on slot 4, 1 events processed so far <<<===
HiveAlgA                        0 0 DEBUG execute HiveAlgA
HiveAlgB                        0 0 DEBUG execute HiveAlgB
HiveAlgB                        0 0 INFO context: s: 0 e: 0 for 0x17dc0000
HiveAlgB                        1 1 DEBUG execute HiveAlgB
HiveAlgB                        0 0 INFO sleep for: 15 ms
HiveAlgB                        1 1 INFO context: s: 1 e: 1 for 0x17dc4800
```

Event#

Slot#

Job Control

- Run an MT job like any other Athena job, and specify number of threads and/or number of concurrent events via Command-Line Interface
- Two knobs to tweak
  - `--threads=N` number of threads in Thread Pool
  - `--concurrent-events=N` number of concurrent events
- In general, easiest to just specify `--threads=N`, which will set both parameters to the same number
- Setting `--threads=1` will run **AthenaMT with 1 thread, not serial Athena**