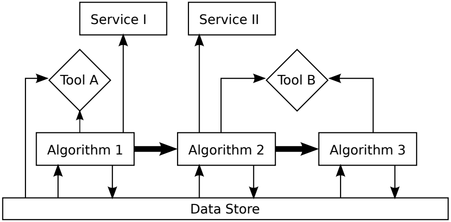
Gaudi Components for Concurrency

Marco Clemencic, Daniel Funke, Benedikt Hegner,
Pere Mato, Danilo Piparo and Illya Shapoval

CERN PH-SFT / KIT ITI

ACAT 2014

Classical Data Processing Frameworks



- **Algorithm:**
  - consumes and produces data objects from/to data store
  - steers further processing depending on data
- **Tool:**
  - computation that can be re-used by several algorithms
  - may consume and produce data objects
- **Service:**
  - provide fundamental framework functionality to all algorithms and tools
  - is managed by the context of the framework

Classical Data Processing Frameworks (contd.)

- were designed for sequential processing
- benefited from steadily increasing CPU clock speeds

However, in recent years

- clock speeds have stopped increasing
- amount of collected physics data still does
- with higher collision energies, processing time per event increases

Addressing the Challenge

One job per core does not scale:

- limited memory amount/bandwidth
- particularly for many-cores not feasible

Instead, fine-level parallelism needs to be exploited

- **inter-event:** one process handles several events in parallel
- **intra-event:** executing independent algorithms within one event concurrently
- **intra-algorithm:** simultaneous processing of many physical objects

Addressing the Challenge

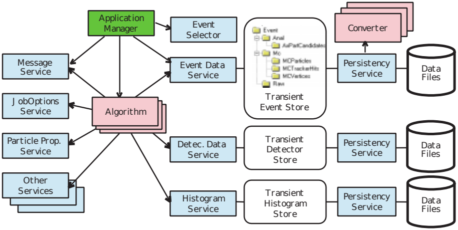
One job per core does not scale:

- limited memory amount/bandwidth
- particularly for many-cores not feasible

Instead, fine-level parallelism needs to be exploited

- **inter-event:** one process handles several events in parallel
- **intra-event:** executing independent algorithms within one event concurrently
- **intra-algorithm:** simultaneous processing of many physical objects

The Gaudi Framework



[6]

- generic data processing framework
- provides clear interfaces
- easily extendable and adaptable to experiments
- used by LHCb, ATLAS, FCC, HARP, Fermi, ...

The Concurrent Gaudi Project

Goal: enable inter- and intra-event-level parallelism in the Gaudi framework

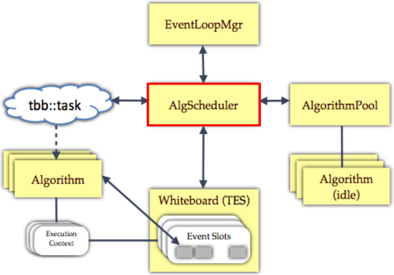
Milestones:

Nov. 2012:
- parallel demonstrator using simulated workloads [IEE NSS 1]

Oct. 2013: *Rel. v0.5*
- parallel execution of LHCb VELO reconstruction [CHEP 2,3]

now *Rel. v0.6*
- evolved workarounds to production quality solutions
- added features essential for parallel scheduling

Gaudi Components for Concurrency



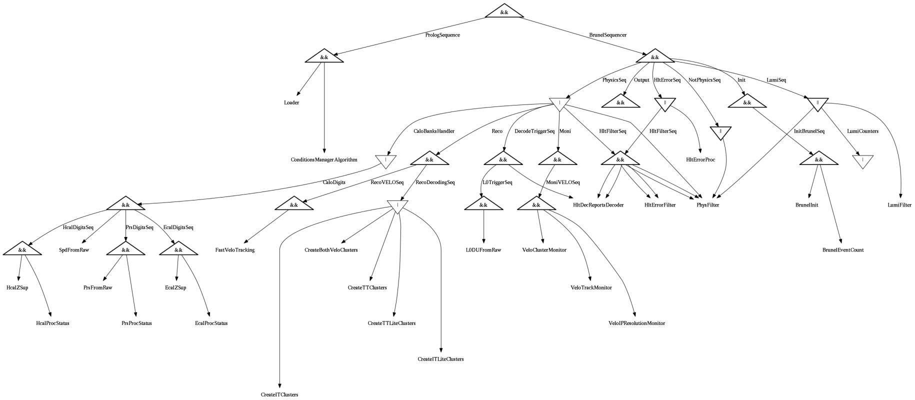
[2]

- events processed in loop and handed over to scheduler
- scheduler acquires algorithm instances from pool and submits them to Intel TBB runtime
- each concurrently processed event has a dedicated slot in the whiteboard (multi-slot event store) to retrieve/store data items

Additional components for:

- concurrent message logging
- shared resource protection
- timeline of multi-threaded algorithm execution

Scheduling



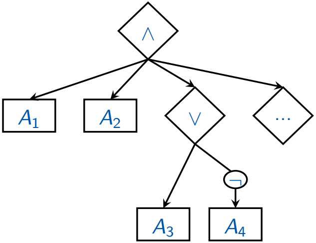
Sequential Gaudi:

- algorithms are arranged in sequences
- each algorithm produces binary decision that may:
  - set decision of sequence (AND/OR)
  - early return of sequence
- sequences can be composed
- algorithms can be part of several sequences

Scheduling (contd.)

Concurrent Gaudi:

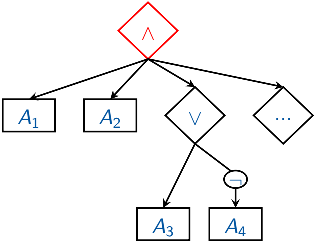
- the **control-flow** is extracted from the sequences
- executability of remaining algorithms is updated with every algorithm decision
- lazily evaluated sequences limit potential for parallelism
  $\Rightarrow$ optimistic execution should be preferred



Scheduling (contd.)

Concurrent Gaudi:

- the **control-flow** is extracted from the sequences
- executability of remaining algorithms is updated with every algorithm decision
- lazily evaluated sequences limit potential for parallelism
  ⇒ optimistic execution should be preferred

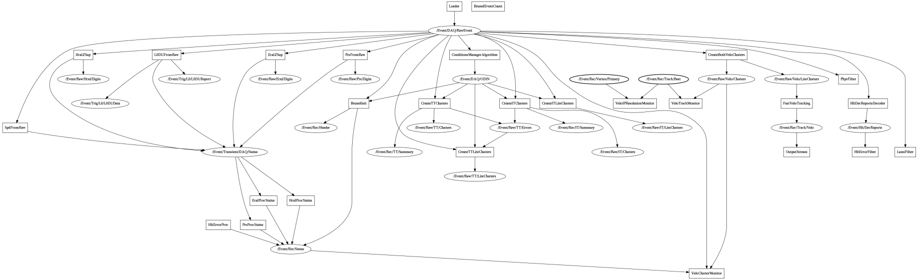


Example

Assuming an early return AND-sequence,
if A_1 produces false, $A_2 \ldots A_4$ not required to executed

Scheduling (contd.)

Algorithms require and produce data objects



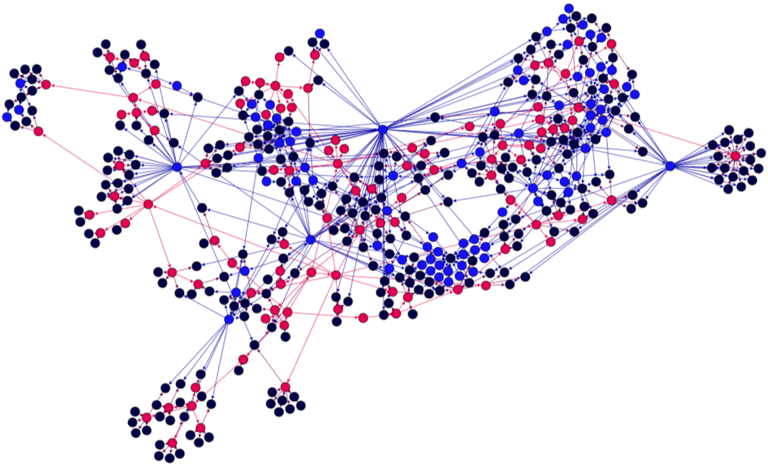
- establishes **data flow** between algorithms
- data flow implicitly contained in control flow structure of **sequential Gaudi**

Unifying Control and Data Flow

Concurrent Gaudi:

- data dependencies need to be explicitly stated
- control and data flow can be expressed in a unified graph
  - graph contains algorithm, data and decision nodes
  - two edge types for control flow and data dependencies
- information for scheduler about parallelizable flows within the sequence

Unifying Control and Data Flow



brunel2012magdown workflow

Unifying Control and Data Flow

Graph analysis can yield insights on the execution flow:

- unfulfillable data dependencies of algorithms
  unreachable data node connected to algorithm
- superfluous control flow constructs
  paths of decision nodes of in-degree = out-degree = 1
- critical paths and maximal concurrency level
- priorities for algorithm execution
  out-degree of node

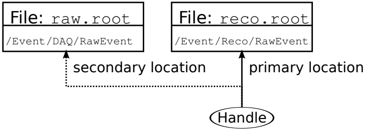
Declaring Data Dependencies

All interaction with data store must be made through data handles

- smart pointers that properly register read/written data object with framework
- thus, allow **automatic** deduction of data dependencies between algorithms

Data handles provide:

- declaration syntax familiar to Gaudi developers
- transparent use of alternative locations for a data object
- customization of properties in configuration file



Declaring Data Dependencies (contd.)

Data handles provide locking mechanism for update operations

Caveats:

- only truly thread-safe if:
  - `update` operation is commutative
  - no other mutable data is used for update
    e.g. updates depending on another status
- performance penalty to pay

Declaring Data Dependencies (contd.)

Data handles provide locking mechanism for update operations

Caveats:

- only truly thread-safe if:
  - `update` operation is commutative
  - no other mutable data is used for update
    e.g. updates depending on another status
- performance penalty to pay

Updating data objects poses non-trivial problems to non-deterministic execution
$\Rightarrow$ just re-ordering of sequences might have unexpected effects

Ideal: everything in the data store is **`const`**

Declaring Tools

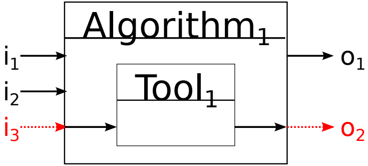
Algorithms may use

private tool owned by algorithm exclusively

public tool owned by framework, shared by several algorithms

Interaction with tools via tool handles provides:

- **automatic** propagation of tools in- and output to algorithm
- declaration syntax familiar to Gaudi developers
- optional configurability of private tools in configuration file



Context-aware Data Access

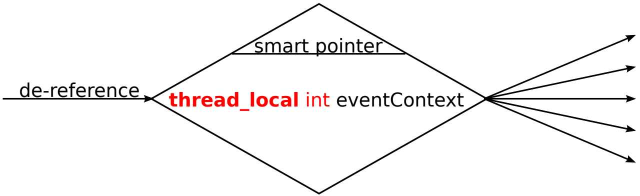
With concurrently processed events, event-specific data must

a) be stored in the data store

- thread-safe and context-aware
- event-context transparently set by framework through thread local index

b) use Gaudi's context-aware smart pointer

- smart pointer de-references to object associated with processed event
- thread local index set by framework



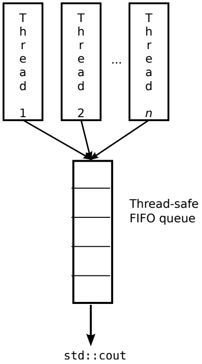
Muli-threaded Message Logging

Logging to `std::cout` is not thread-safe:

- interleaved output from different threads
- corrupted output buffer

`TBBMessageSvc` resides in own thread:

- output messages buffered in thread-safe queue
- no interleaving, order of messages preserved
- drop-in replacement for `MessageSvc`
- can be used in sequential mode to offload logging



Adoption by Existing Experiments

Concurrent features do **not** interfere with production sequences
$\Rightarrow$ sequential Gaudi can run unaltered

- data and tool handles can be gradually adopted algorithm by algorithm
  $\Rightarrow$ advantage of execution graph analysis even without concurrent processing
  e.g. identification of inconsistencies, superfluous algorithms, ...
- existing functionalities of the framework were instrumented to ease migration
  - classical tool retrieval method via `tool<T>( ... )` method
    $\Rightarrow$ properly registers tool usage with parent algorithm/tool
    for **automatic** dependency propagation
  - transparent use of context-aware smart pointer

Adoption by Existing Experiments (contd.)

However, parallel processing does not come for free!

Some things need to be re-[implemented, designed]:

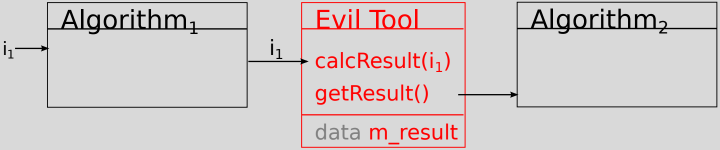
- use of caches within algorithms and tools
- thread-unsafe updates to data objects in the data store
- abuse of public tools for back-channel communication
- ...

Adoption by Existing Experiments (contd.)

However, parallel processing does not come for free!

Some things need to be re-[implemented, designed]:

Example already a problem now



implicit dependency between Algorithm_1 and Algorithm_2
$\Rightarrow$ can **not** be automatically deduced by framework

Adoption by Existing Experiments (contd.)

However, parallel processing does not come for free!

Some things need to be re-[implemented, designed]:

- use of caches within algorithms and tools
- thread-unsafe updates to data objects in the data store
- abuse of public tools for back-channel communication
- . . .

Again, incremental approach:

- revise algorithms one at a time
- enable parallel processing workflow by workflow

Adoption by LHCb

Decision taken to merge concurrency components into production Gaudi

- gradual adaption of data and tool handles
- immediate benefit of static configuration checking
- paving the road to go parallel
- user feedback will help to distill further best practices for adoption

Future Circular Collider

FCC develops new experiment software based on Gaudi

- design with concurrency in mind
  - algorithms access data store only via data handles
  - tools are declared at configuration time
  - data store is used for algorithms' intermediate results
  - services are re-entrant or context-aware
  - const-correctness is enforced
- challenge of integrating external packages in thread-safe manner

Summary and Outlook

Features developed in Concurrent Gaudi Project are ready to be used by existing and future experiments [5]

Existing experiments

- can apply incremental adoption strategy
- immediately benefit from static configuration checking
- pave the road to go parallel

Further developments:

- support adaption of concurrency by experiments
- leverage asynchronous writes to the data store
- explore use of accelerators

References

1. P. Mato, Evolving LHC Data Processing Frameworks for Efficient Exploitation of New CPU Architectures, IEEE NSS 2012, November
2. D.Piparo, Preparing HEP Software for Concurrency - Lessons learned from the Concurrent Gaudi Project, CHEP 2013, October
3. B. Hegner, Introducing Concurrency in the Gaudi Data Processing Framework, CHEP 2013, October
4. Concurrency for HEP Twiki:
   `https://twiki.cern.ch/twiki/bin/view/C4Hep/WebHome`
5. Gaudi Hive git repository:
   git clone −b dev/hive http://cern.ch/gaudi/GaudiMC.git
6. Barrand G. et al., GAUDI - A software architecture and framework for building LHCb data processing applications, CHEP 2000

Backup

Declaring Data Dependencies

Use data handles to access data store from algorithms and tools:

Code

```
class MyAlgorithm : public GaudiAlgorithm{

private:
  DataObjectHandle<LHCb::Tracks> m_tracks;
  DataObjectHandle<LHCb::Tracks> m_filteredTracks;

public:
    MyAlgorithm( ... ) : GaudiAlgorithm( ... ) {
      declareInput("Tracks", m_tracks,
                   LHCb::TrackLocation::Default);
      declareOutput("FilteredTracks", m_filteredTracks,
                    "Analysis/FilteredTracks");
    }

    void execute() {
      LHCb::Tracks * tracks = m_tracks.get();
    }
};
```

Declaring Data Dependencies – Examples

Code: Python configurability

```
myAlg = MyAlgorithm('AnalysisFilter')
myAlg.Inputs.Tracks.Path = 'Skim/Tracks' # use pre-filtered tracks
```

Code: `DataObjectHandle` interface

```
template<typename T>
class DataObjectHandle : public MinimalDataObjectHandle {

  ...
  bool exist();
  T* get();
  T* getIfExists();
  T* getOrCreate();
  void put (T* object);

  void lock();
  void unlock();

}
```

Data Dependencies – Concurrency Features

Locking mechanisms for "thread-safe" access to data objects

Code: `DataObjectHandle` interface

```
void MyAlgorithm::updateStatus(const Status & status){

  m_status.lock();

  GlobalStatus* gStatus = m_status.getOrCreate();
  if(!gStatus.contains(status.key())){
    gStatus->insert(status);
  } else {
    gStatus->update(status);
  }

  m_status.unlock();
}
```

⇒ transitional migration tool, many caveats involved

Declaring Tools

Declare tools used by algorithm at configuration time:

Code

```
class MyAlgorithm : public GaudiAlgorithm{

private:
  ToolHandle<ITrackExtrapolator> m_extrapolator;
  ToolHandle<IMaterialLocator>    m_materialLocator;

public:
    MyAlgorithm( ... ) : GaudiAlgorithm( ... ) {
      declarePrivateTool(m_extrapolator, "TrackLinearExtrapolator");
      // optionally make it a property
      declareProperty("TrackExtrapolator", m_extrapolator);

      declarePublicTool(m_materialLocator, "DetailedMaterialLocator");
    }
};
```

Declaring Tools - Example

Code: Python configurability

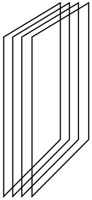
```
myAlg = MyAlgorithm('AnalysisFilter')
myAlg.TrackExtrapolator.Iterations = 1 # rough estimate
```

Scheduling

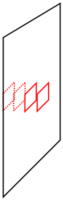
Different scheduling strategies transparently available:

- **Parallel Sequential** mimic multi-process approach
  $\Rightarrow$ but with reduced memory footprint

multi-process



multi-thread



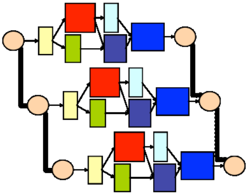
context specific state

Scheduling

Different scheduling strategies transparently available:

- **Parallel Sequential** mimic multi-process approach
  $\Rightarrow$ but with reduced memory footprint
- **Forward** schedule executable (control-flow) algorithms as soon as their input becomes available (data-flow)

Only forward scheduler exploits intra-event parallelism



[2]

Scheduling

Different scheduling strategies transparently available:

- **Parallel Sequential** mimic multi-process approach
  $\Rightarrow$ but with reduced memory footprint
- **Forward** schedule executable (control-flow) algorithms
  as soon as their input becomes available (data-flow)

Only forward scheduler exploits intra-event parallelism

Future plans:

- backward schedule only algorithms required to produce final result
- use accelerators: bunch up events to make load-off profitable