Preparing HEP Software for Concurrency

Lessons Learned from the Concurrent Gaudi Project

M. Clemencic, B. Hegner, P. Mato, D. Piparo

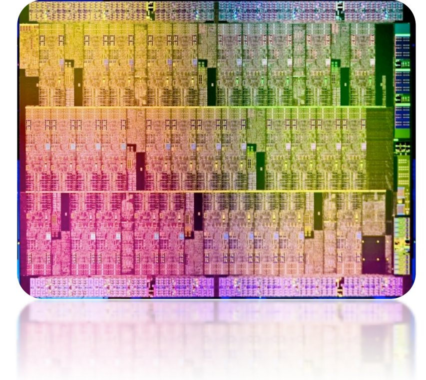
CERN

CHEP 2013

See contribution Introducing Concurrency in the Gaudi Data Processing Framework by B. Hegner on Monday for all the performance figures!

Motivation

- **Clear trend towards parallelism** in microprocessor technology
  - Many cores, larger vector units
  - Fat/general purpose but also thin/specialised (e.g. RISC)
  - Accelerators



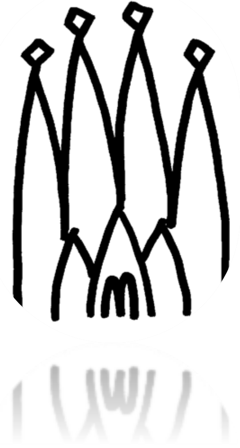
- Ambitious physics programs for the future, e.g. LHC experiments

Present HEP software solutions will not scale for the next decade

- Evolve data processing algorithms and toolkits (e.g. simulation and analysis)
- **Evolve software frameworks**

Goal of the Parallel Gaudi Project

Gaudi: general data processing framework, adopted by Atlas and LHCb and other non LHC experiments (e.g. Minerva)



Provide refurbished **Gaudi framework to support parallelism**

- **At event level**: multiple events treated simultaneously
- **At algorithm level**: execute multiple algorithms concurrently
- **Within algorithms** (intra-algorithm parallelism)

Pragmatic attitude: **start from slice of a real LHCb reconstruction**

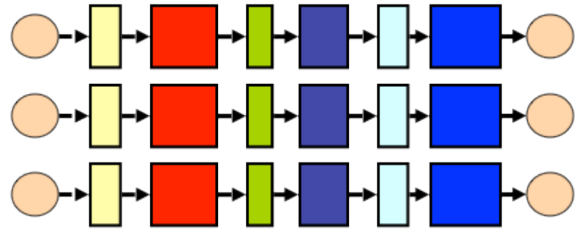
- 20 algorithms for raw unpacking and velo tracking

Highly innovative but evolutionary approach

- **Operate on large, existing and successful codebase**
- **Minimal changes of interfaces and existing code**
- **Provide new components**

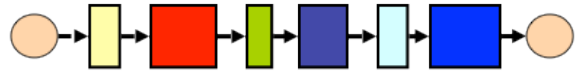
Nomenclature: **Data processing units**, implemented as C++ classes are called **Algorithms** (**modules** in CMS' jargon)

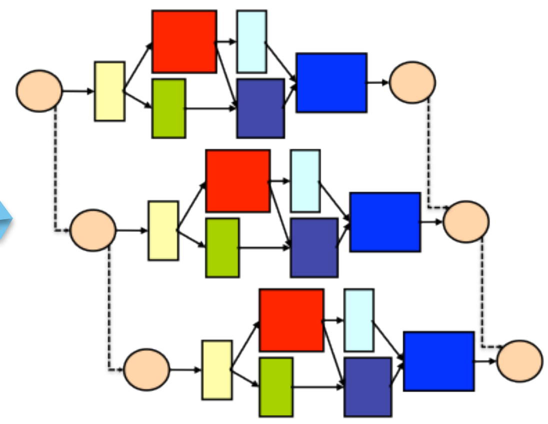
Our Goal



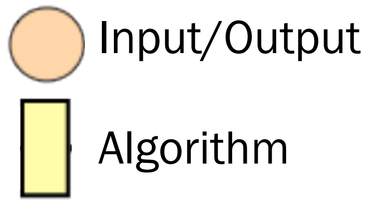
Simultaneous Multiple events

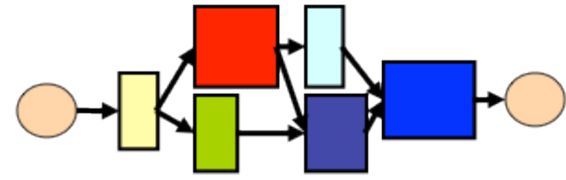
Sequential execution





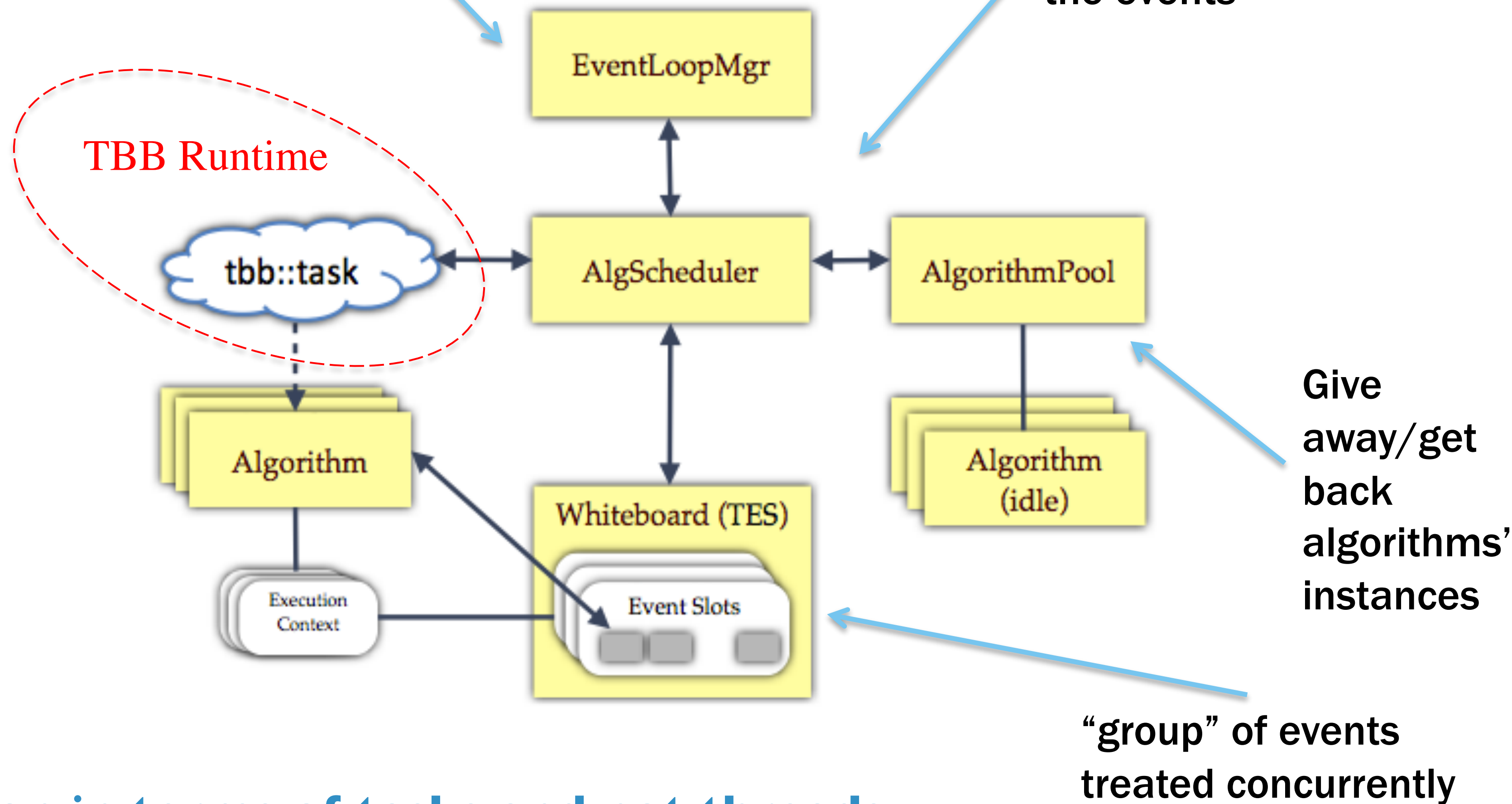
Parallelism at all levels





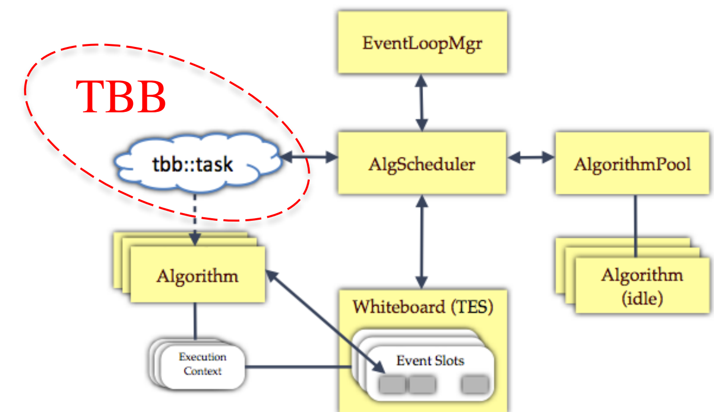
Parallelism within an event

Concurrent Gaudi's Design, in a Nutshell



- **Reason in terms of tasks and not threads**
  - Task: **closure of algorithm + its input data** = our unit of work
- Intel Threading Building Blocks chosen as runtime
- C++11 standard adopted

1. **Resource protection, thread safety and correctness**
2. Treatment of multiple events simultaneously
3. Scheduling
4. Caches and their classification



Resource Protection and Correctness

Algorithms executed simultaneously: resources to be protected

Possible strategies: e.g. transactional memory, atomic operations, locks (as a last resort)

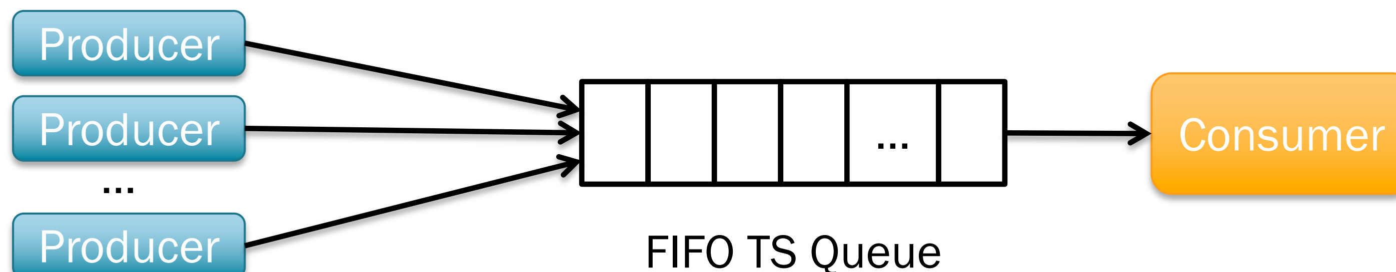
An example:

Problem: **Print on screen messages coming from algorithms running in parallel**

Solution: Serialisation via **thread safe queue of actions**

- Many producers, one consumer in our usecases
- Consumer pulls work until queue empty
- Item of work: function closure
- **Message passing like approach**
- **Closure: message text +print function**

Distil a pattern starting from a concrete issue: pragmatic approach



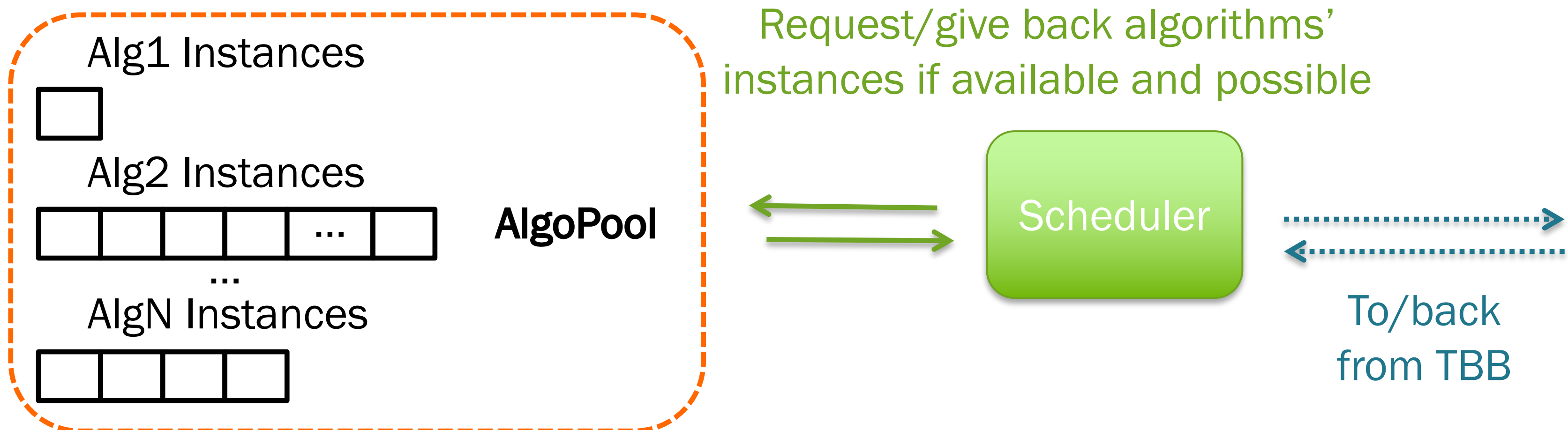
Limit Contention by Design: AlgorithmPool

Problem: algorithms that can access thread unsafe resources (e.g. I/O, libraries) shall not run in parallel

Proper design can minimise contention!

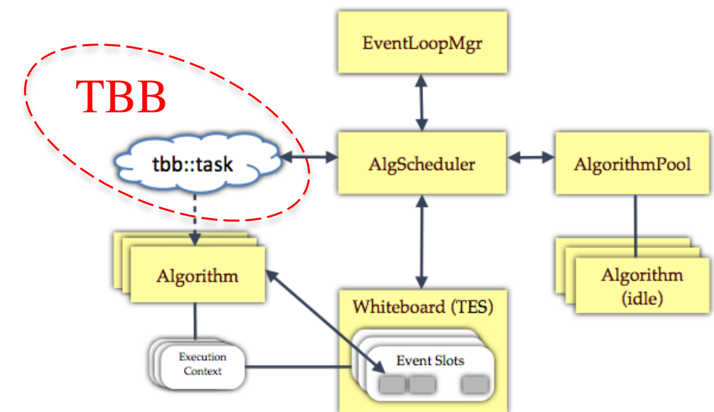
Solution: The AlgorithmPool:

- Contains algorithms in queues and coordinates them
- **Gives away instances to run, retrieves ran algorithms**
- **Exclusive resources:** e.g. N algs declaring to use the same thread unsafe library, only one instance at the time is given away
- **Lock-free solution**



Issues, Strategies and Solutions

1. Resource protection, thread safety and correctness
2. **Treatment of multiple events simultaneously**
3. Scheduling
4. Caches and their classification



Multiple Events Simultaneously

- **Increase problem's size**: more opportunities for parallelism
  - E.g. to increase probability of scheduling an algorithm
- **Need for an "execution context"** carrying information about event being processed
- Major change in philosophy, lots of assumptions broken!
  - E.g. access to event store: from/to which event data should be read/written?
- Two possibilities:
  - **Adapt all code to percolate execution context,** passing by value
  - **Leverage thread local storage for implicit execution context passing***

Minimal intervention: go for the latter

* Against purely task oriented approach: make assumption about underlying threads

The Whiteboard Pattern

Problem: accommodate several event stores

Solution: A natural extension of event data store for multi event case

- **Contains several event stores**
- **Implements same interface of event store**, thread safe.
  - Replace access to event store with access to whiteboard transparently
  - **No need to change users' code**
- Forwards calls to internal stores according to execution context
  - Event number – Event Store mapping
- Bookkeeping of data stores' contents for schedulers

Cloning of Entities

Problem: Performance degraded by long running algorithms: "block" scaling forcing a coarse granularity

Solution: Clone algorithms, i.e. have identical instances able to run simultaneously on several events

- If all algorithms were re-entrant: no need for cloning
  - >10 years of development with no re-entrancy in mind
- **Cloning: tradeoff between additional memory footprint and modification of ~all existing code**
- **Delicate issue**: "deep copying"
  - E.g. Members which are pointers pointing to the same entity

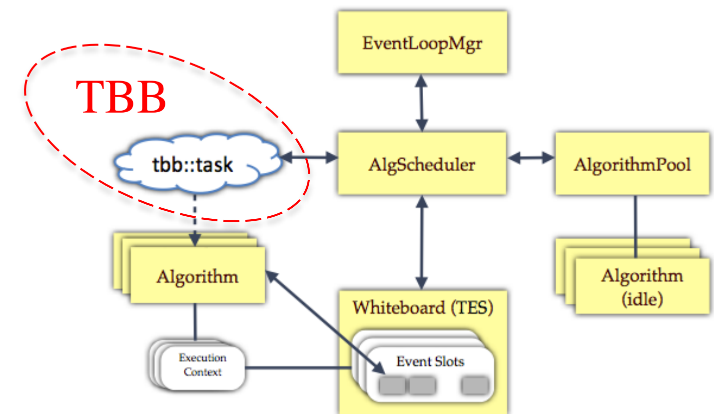
Cloning: a general pattern valid not only for algorithms

See contribution Introducing Concurrency in the Gaudi Data Processing Framework

for all the performance figures!

Issues, Strategies and Solutions

1. Resource protection, thread safety and correctness
2. Treatment of multiple events simultaneously
3. **Scheduling**
4. Caches and their classification



Scheduling Strategies

The scheduler is at the heart of a parallel framework

Gaudi allows at runtime to choose implementation of interfaces:

→ **different implementations of the IScheduler interface**

Framework invariant wrt scheduling choice!

- **Forward scheduler** (our primary choice, tested, profiled)
- **Parallel sequential scheduler** (being tested)
- **Round Robin scheduler** (being tested)
- Other being thought through, e.g. **Backward scheduler**

Compare these scheduling strategies among themselves, with multi-process and one-job-per-core approaches

Forward Scheduler 1/2

The idea: schedule an algorithm as soon as its input data is available

1) Check list of data object in the whiteboard
2) Check which algorithms can be run given the available data
3) Submit algorithms which are ready to the runtime (goto 1)

Needs:

- **Input data dependencies** declared
- Treat **multiple events simultaneously** (for reasonable performance)
- **Algorithms cloning** (for top performance)

Pros:

+ Scheduling intrinsically immune to deadlocks
+ Possible to lump data from different events for computations on accelerators
+ Possibility to run same algorithm repeatedly on the same core: cache friendly

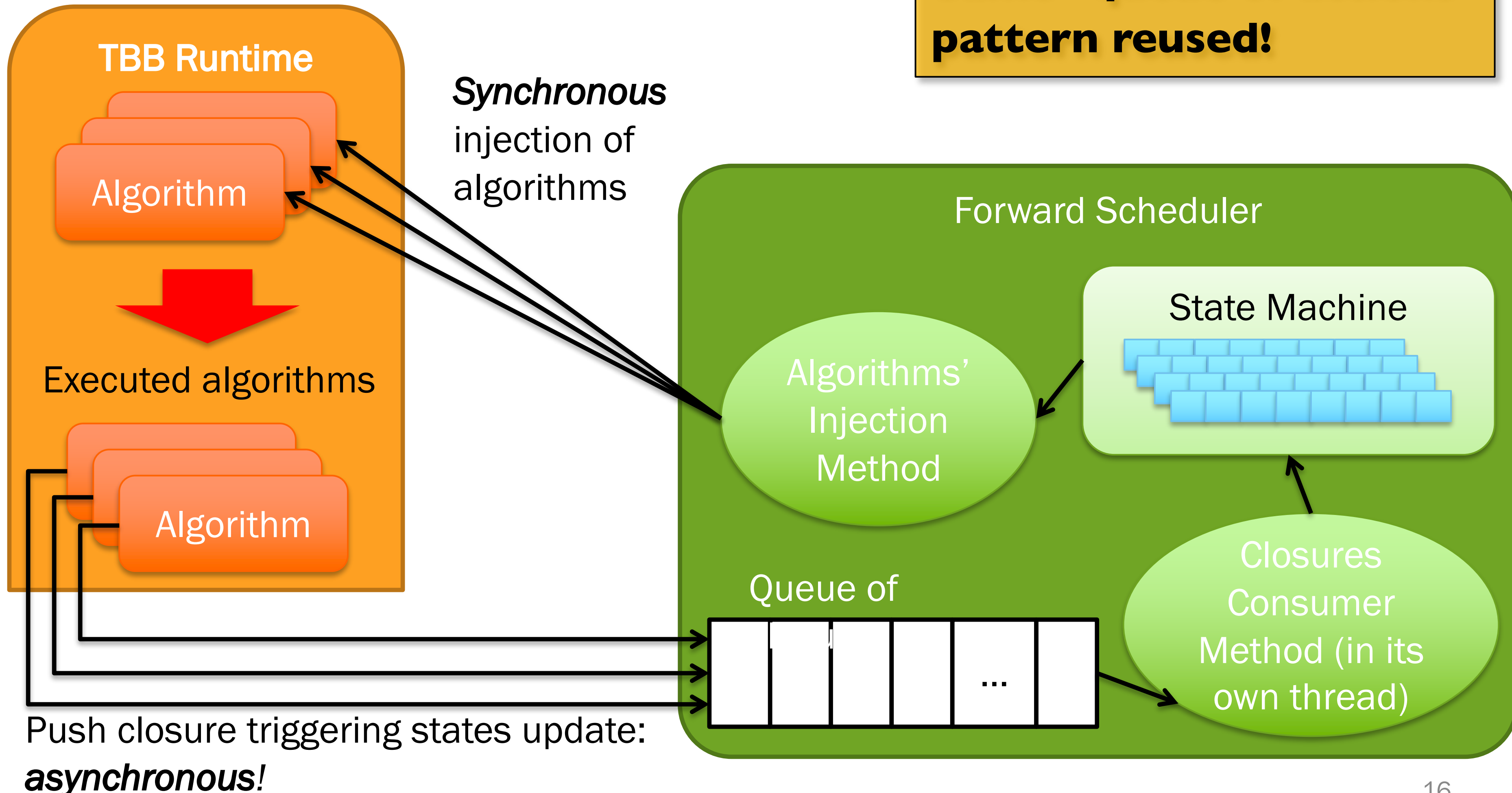
Cons:

- Needs all data dependencies to be explicit

Forward Scheduler 2/2

- Absorbs asynchronous finishing of submitted tasks with a queue
- State machine: keep track for each algorithms a state
  - E.g. Steering of scheduling, stall detection

Same "queue of actions" pattern reused!



Parallel Sequential Scheduler

The idea: run in parallel on several events simultaneously the full sequence of algorithms (similar to MP approach)

Needs:

- Treat **multiple events simultaneously**
- **Algorithms cloning** (for reasonable performance)

Pros:

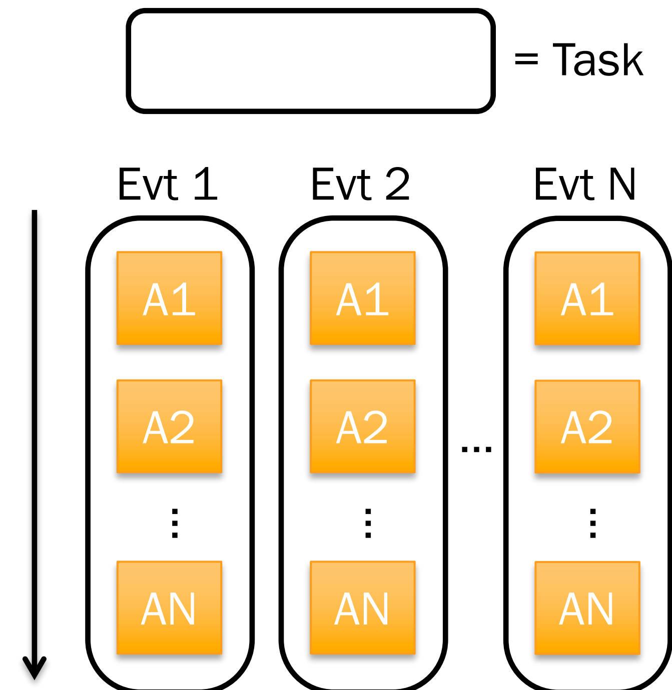
\+ No need for data dependencies (order preserved)

\+ No "output merging process": whiteboard used

\+ More memory saved than COW

Cons:

\- Much less flexible than forward scheduling



Round Robin Scheduler

The idea: using one core only, run the same algorithm on all events in a round-robin fashion

Needs:

- Treat **multiple events simultaneously**

Pros:

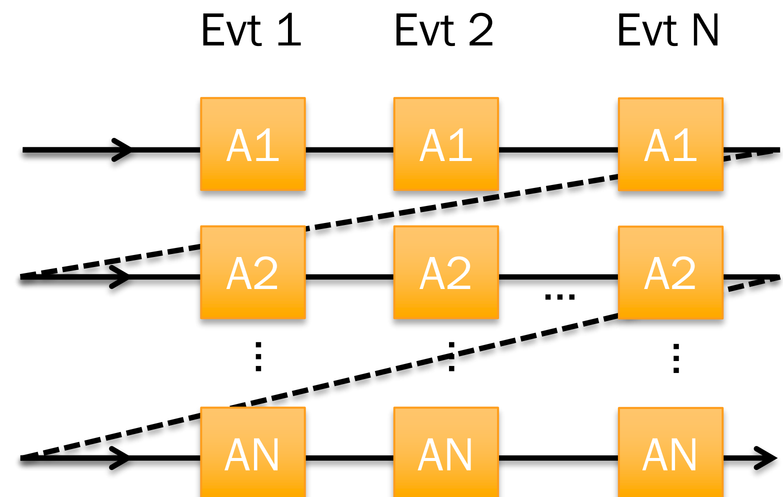
+ One core only (e.g. same wms' of today)
+ Optimised instruction cache usage (if algorithms not too long)
+ Possible to lump data from different events for computations on accelerators
+ Predictable memory access pattern (pre-fetching)

Cons:

-- One core only!

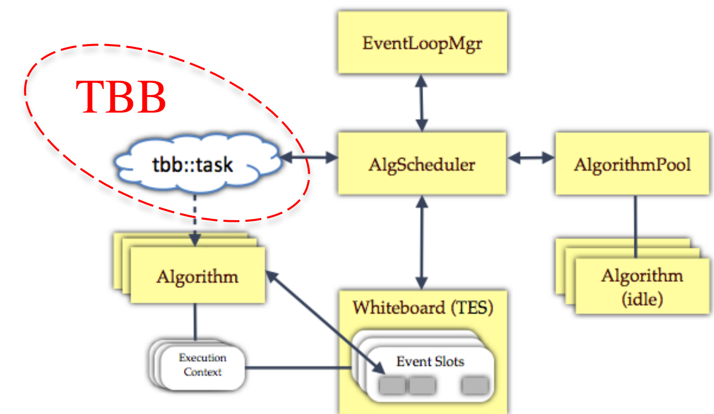
- Cannot mask the offload latency

A good checkpoint between sequential and full MT execution!



Issues, Strategies and Solutions

1. Resource protection, thread safety and correctness
2. Treatment of multiple events simultaneously
3. Scheduling
4. **Caches and their classification**



Caches and their classification

Problem: Code base full of caches – Case by case study

Methods of classification

- Validity span: for an event, a run or the whole application
- Local caches: within algorithms
  - Usually harmless, intelligence needed: e.g. counters, histograms
- Global shared caches: outside algorithms
  - **E.g. communication channels among algorithms to bypass event store**

Possible solutions:

- **Check whether they are actually needed!**
- **Caches initialised once**: put object in the event store
- **Result caching**: provide one separate cache per event processed simultaneously

Conclusions

Knowledge distilled in several fields as general strategies and patterns

- **Schedulers:** different available, easy switching
- **Resources' protection**
- **Conservation of existing codebases and migrations**
  - **Closer look to code: opportunity for improvements everywhere**

Achieve **parallelism in Gaudi with a pragmatic approach**:

- Consider a **real life use case**: LHCb reconstruction
- **Solve problems one by one**
- **Potential reuse of our findings** in our contexts, e.g. other frameworks
- **Pursue this effort**: all algorithms of LHCb reconstruction

See contribution Introducing Concurrency in the Gaudi Data Processing Framework

for all the performance figures!

Backup

Useful Material

Project Page on the Concurrency Forum Site:

http://concurrency.web.cern.ch/GaudiHive

Main Twikipage:

https://twiki.cern.ch/twiki/bin/view/C4Hep

Git Repository Web Interface:

http://lcgapp.cern.ch/git/GaudiMT/

Jira:

https://sft.its.cern.ch/jira/browse/CFHEP