Concurrent Data Processing Frameworks

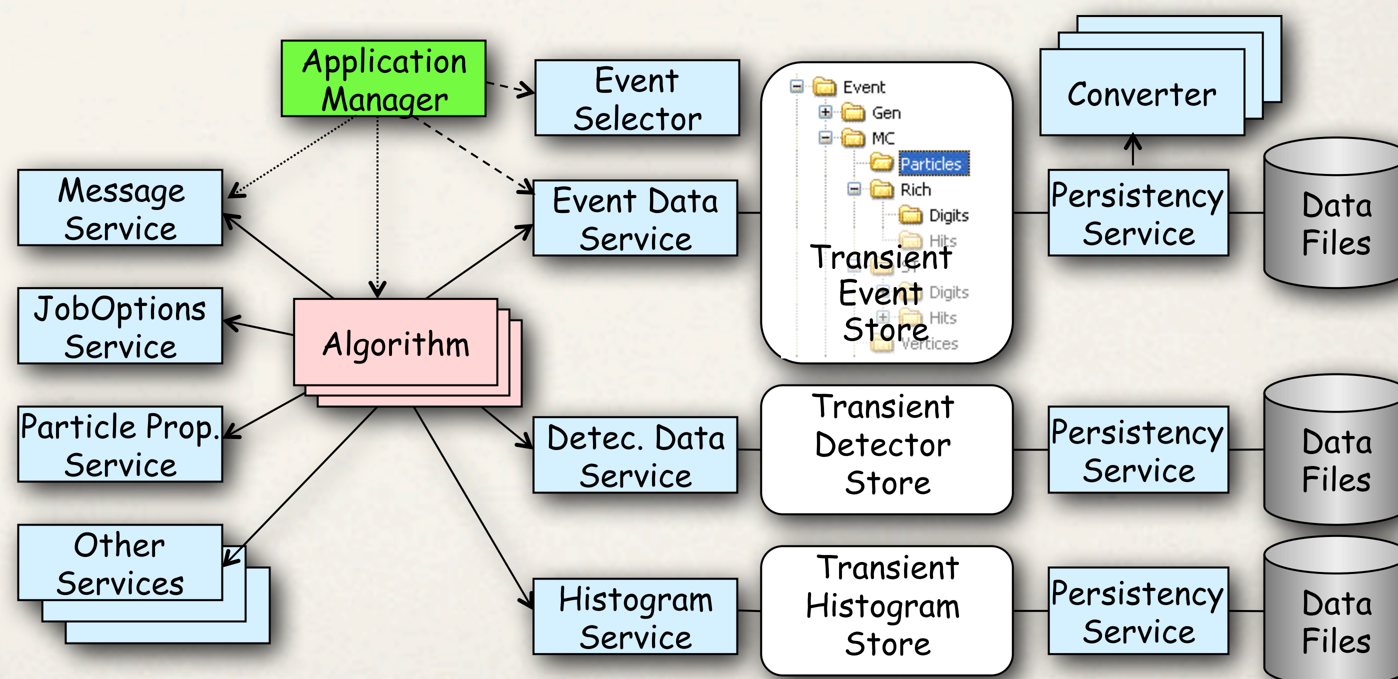
ALICE Offline Week - November 8, 2013

P. Mato/CERN

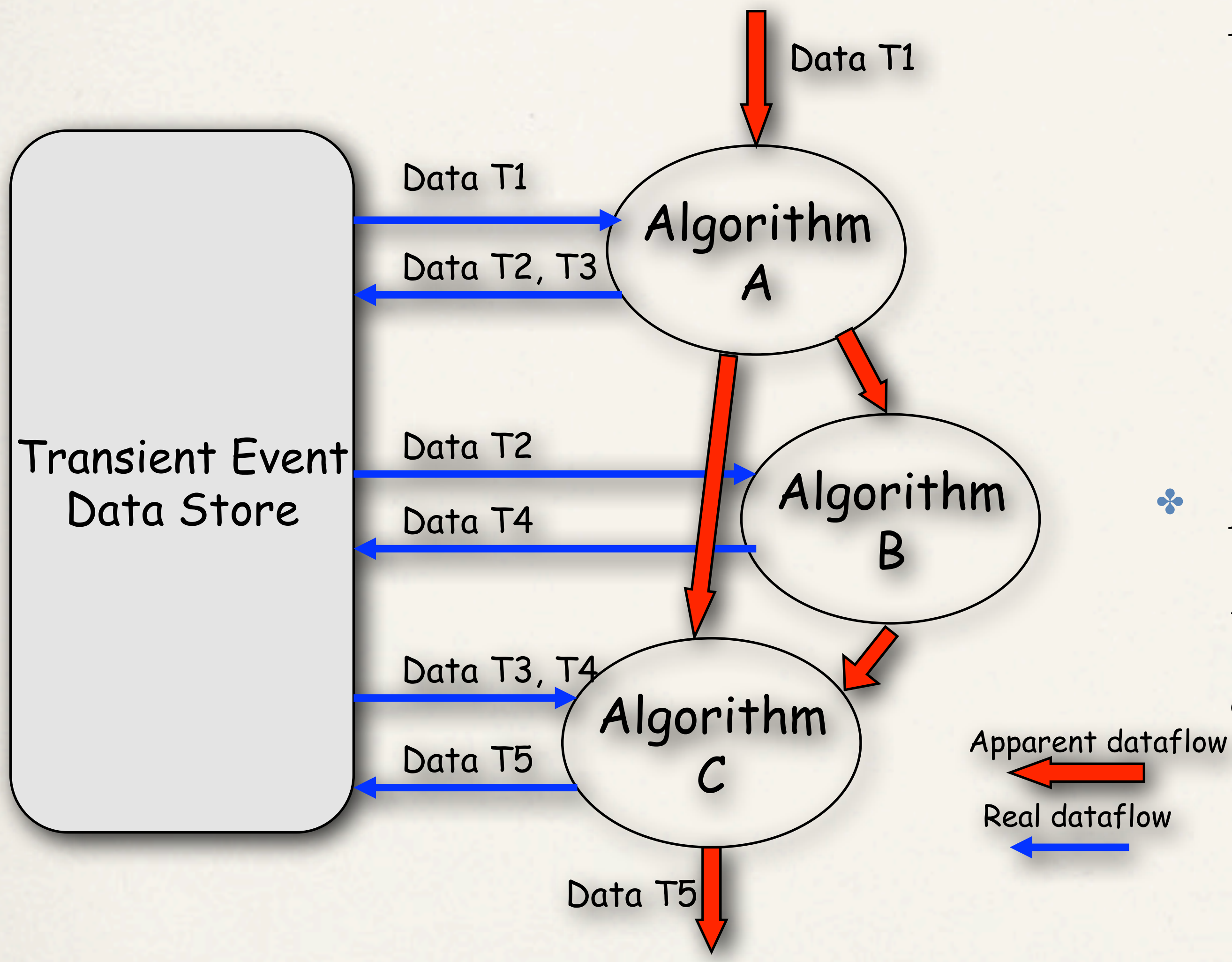
HEP Software Frameworks

- HEP Experiments develop Software Frameworks
  - General Architecture of the Event processing applications
  - To achieve coherency and to facilitate software re-use
  - Hide technical details to the end-user Physicists (providers of the *Algorithms*)
- Applications are developed by customizing the Framework
  - By composition of elemental *Algorithms* to form complete applications
  - Using third-party components wherever possible and configuring them

• Example the Gaudi Framework used by ATLAS and LHCb among others



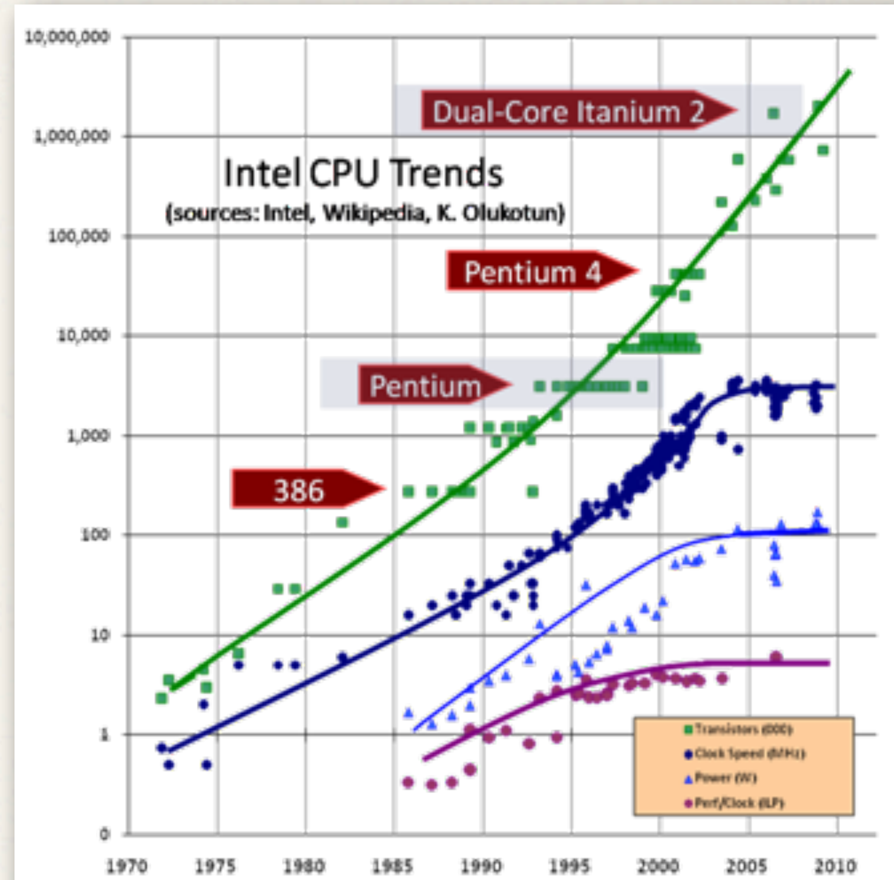
Algorithms and Data Flows



- The meat of the applications is coded by physicists in terms of *Algorithms*
  - They transform raw input *event data* into processed data
    - e.g. from digits -> hits -> tracks -> jets -> etc
- *Algorithms* solely interact with the *Event Data Store* ("whiteboard") to **get** input data and **put** the results
  - Agnostic to the actual "producer" and "consumer" of the data
  - Complete data-flows are programmed by the integrator of the application (e.g. Reconstruction, Trigger, etc.)

CPU Technology Trends

- For the last ~20 years we have had an easy life in HEP software and computing
  - Year after year up to 2x increase in computing capacity tanks to the #transistor/chip (Moore's law) and higher clock frequencies
  - The same program that in year 1995 was needing 10 seconds, would need 1 second in 2002



- The "easy life" is now over
  - The available transistors are used for adding new CPU cores while keeping the clock frequency basically constant thus limiting the power consumption
- We need to **introduce concurrency** into applications to fully exploit the continuing exponential CPU throughput gains
  - Efficiency and performance optimization will become more important

Why Concurrency?

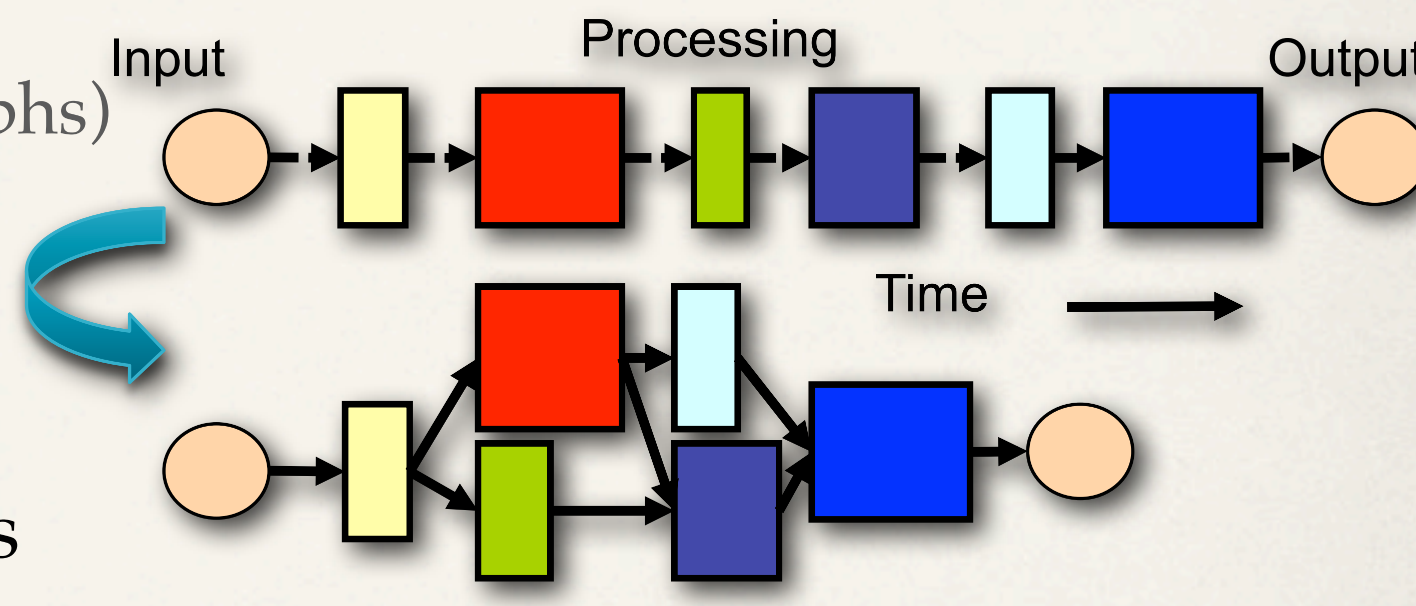
- We need to adapt current data processing applications to the new many-core architectures (~100 cores)
  - No major change is expected in the overall throughput with respect to trivial one-job-per-core parallelism with today core counts
- We must reduce the required resources per core to avoid real barriers when scaling to ~100 cores
  - I/O bandwidth
  - Memory requirements
  - Connections to DB, open files, etc.
- Reduce latency for single jobs (e.g. trigger, user analysis)
  - Run a given job in less time making use of all available cores
- Make possible the use of coprocessors
  - Lumping data from several events together

Concurrency at What Level?

- Concrete HEP algorithms can be parallelized with some effort
  - Making use of bare threads, OpenMP, MPI, OpenCL, Cuda, etc.
  - But difficult to integrate them in a complete application
  - Much more beneficial performance-wise to concentrate on the parallelization of the full application, not only on some parts (Amdahl's law)
- Developing and validating parallel code is very difficult
  - Very technical, difficult to validate and debug
  - 'Physicists' should be saved from this
  - Concurrency will impose some limitations on the way Algorithms are coded
- At the **Framework level** you have the full overview and control of the application
  - Controlling the access to critical shared states and resources
  - The framework may decide to run some parts of the code sequentially

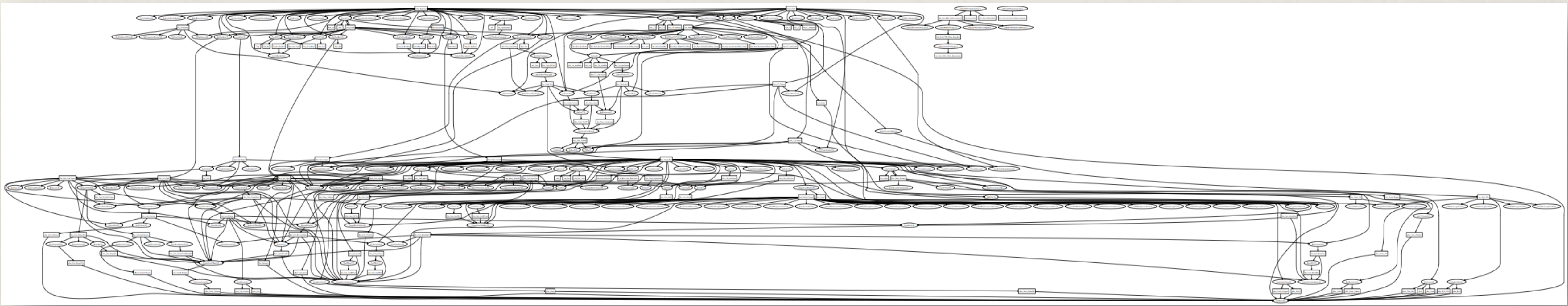
Concurrent 'Algorithm' processing

- Ability to **schedule** modules/algorithms concurrently
  - Full data dependency analysis would be required (no global data or hidden dependencies)
    - DAGs (Directed Acyclic Graphs)
  - Conditional execution of algorithms or sequences thereof

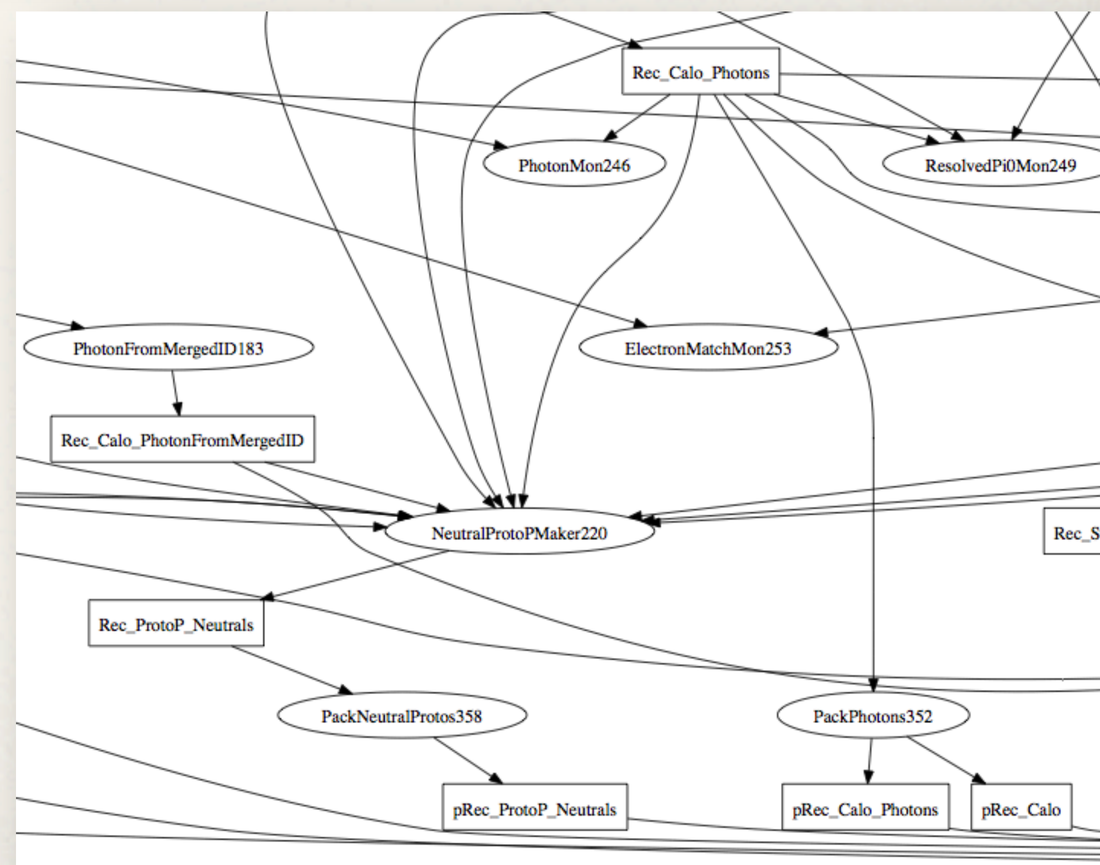


- Need to resolve the data-flow and control-flow dependencies automatically and dynamically
  - Run everything in parallel that isn't constrained by control flow or data flow
- Unfortunately with today's existing *Algorithms* we cannot use efficiently ~100 cores
  - Estimated concurrency factor rather low for CMS and LHCb (between 3 and 6)

Example: LHCb Reconstruction



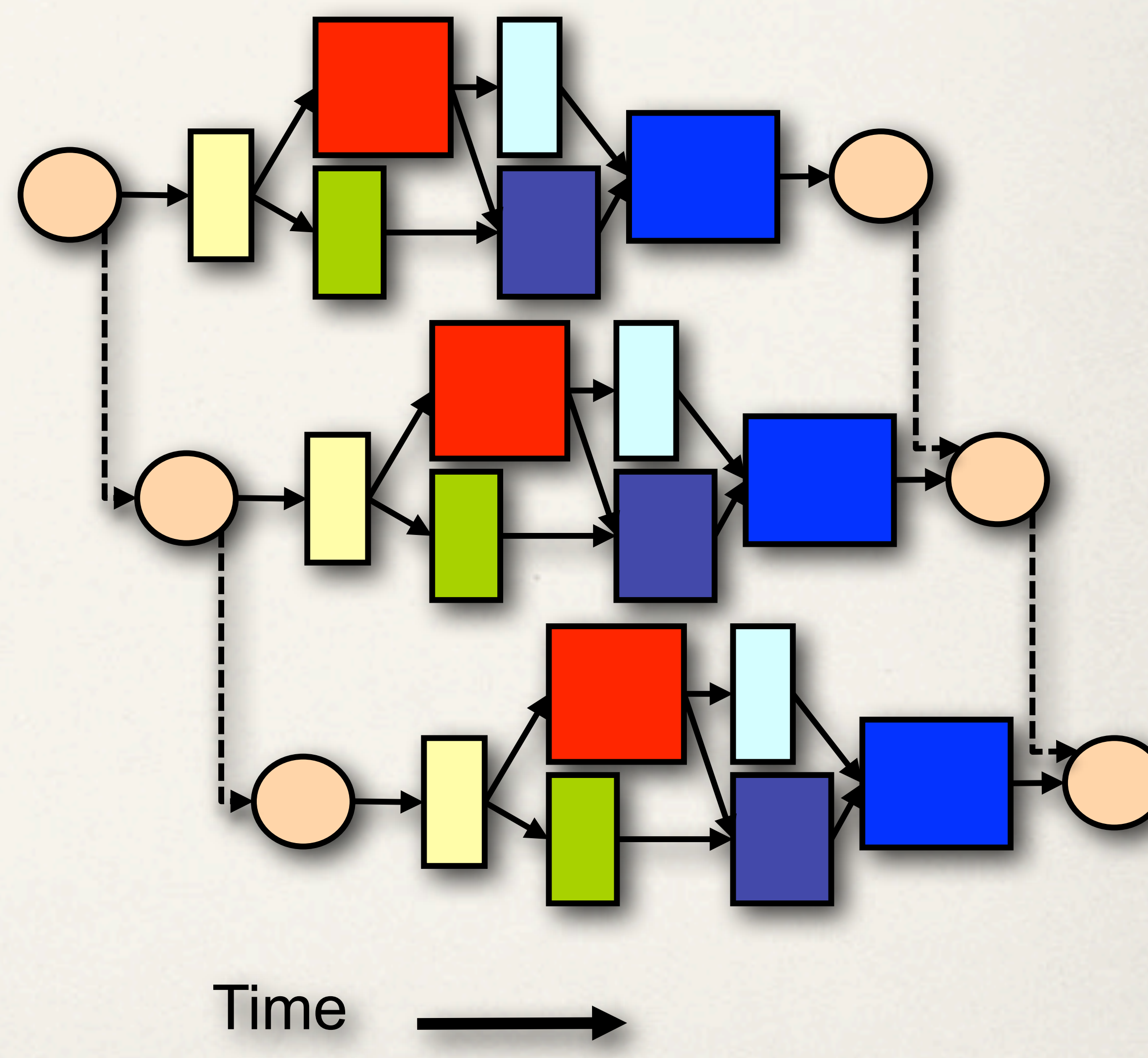
- DAG of Brunel (214 Algorithms)
  - Obtained by instrumenting the existing sequential code
  - Probably still missing 'hidden or indirect' dependencies



- This can give us an estimate of the potential for 'concurrency'
  - Assuming no changes in current reconstruction algorithms

Many 'Concurrent' Events

- Need to deal with the tails of sequential processing
  - There is always an *Algorithm* that takes very long (e.g. 20% in reconstruction) that produces data (e.g. fitted tracks) that are needed by many other
- Introducing *pipeline* processing
  - Exclusive access to resources or non-reentrant algorithms can be pipelined e.g. file reading/writing, DB access, etc.
- Current frameworks handle a single event at the time. They need to be evolved
  - Design a powerful and flexible *algorithm* scheduler
  - Need to define the concept of an *event context*



Prototype Project: GaudiHive

- Provide refurbished Gaudi framework which
  - Allows concurrent execution of algorithms
  - Supports simultaneous processing of multiple events
  - Requires minimal change of user code
- Phase 1
  - New framework components with sufficient functionality to support a small 'slice' of the LHCb reconstruction application (mini-Brunel)
    - Minimize everything that has an impact on current users of Gaudi
    - ~20 algorithms and associated tools (raw decoding and Velo tracking)
  - Ideal for understanding 'threading issues' and validating results
- Phase 2
  - Extern to the complete reconstruction (~200 algorithms)
  - Add remaining set of components and functionality
  - Document the "how-to migration"

How? Initiatives taken so far

- A new forum was established at the start of this year, the **Concurrency Forum**, with the aim of :
  - sharing knowledge amongst the whole community
  - forming a consensus on the best concurrent programming models and on technology choices
  - developing and adopting common solutions
- The forum meets bi-weekly and there has been an active and growing participation involving many different laboratories and experiment collaborations
- A programme of work was started to build a number of **demonstrators** for exercising different capabilities, with clear deliverables and goals
  - 16 projects are in progress started by different groups in all corners of the community
- In the longer term this may need to evolve into other means for measuring progress and steering the future work programme

http://concurrency.web.cern.ch

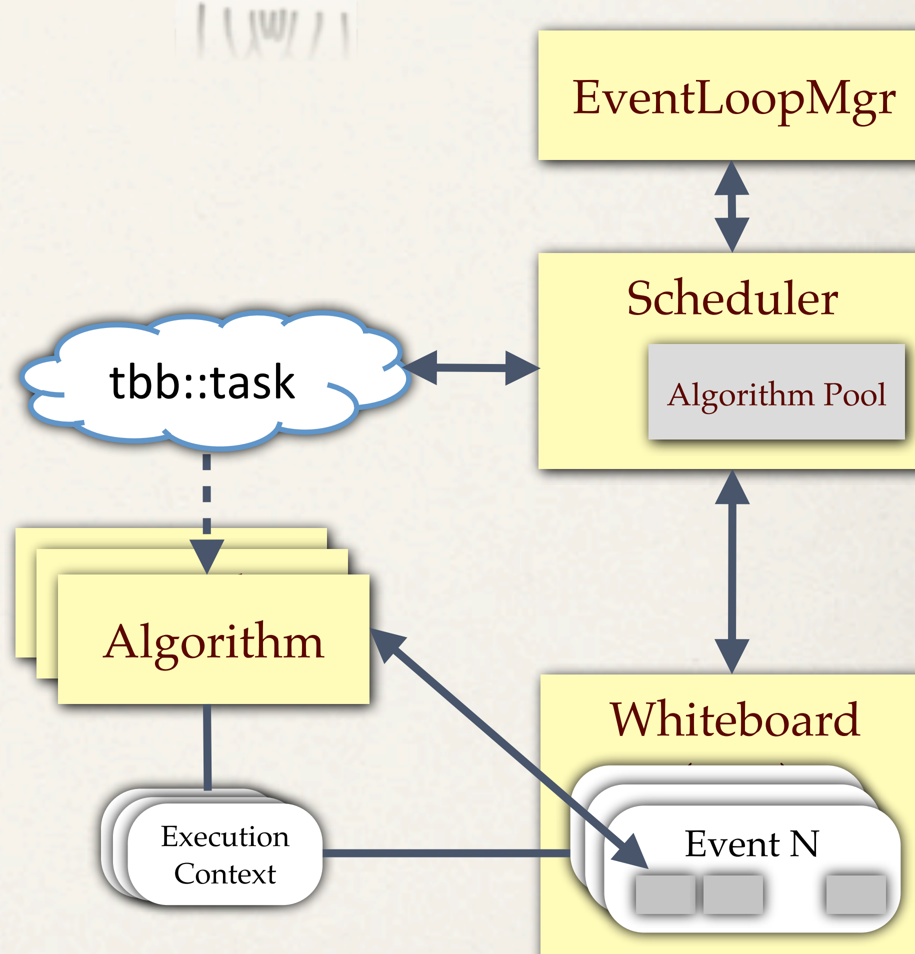
TBB Technology

- Intel® Threading Building Blocks (TBB) has been identified as a good match for implementing concurrency at the Framework level
- C++ library with a rich and complete approach to express parallelism
  - Concurrent containers: concurrent_vector, concurrent_hash_map, ...
  - Algorithms: parallel_for, pipeline, task, ...
  - Other: atomic data types, memory allocators, ...
- Provides a "task-based" programming model that abstracts platform details and threading mechanisms for scalability and performance
- Positive evaluations reported at the **Concurrency Forum**
  - Easy to build and very portable
  - Lower CPU overhead than other libraries evaluated
  - Missing functionalities are generally easy to add

http://concurrency.web.cern.ch

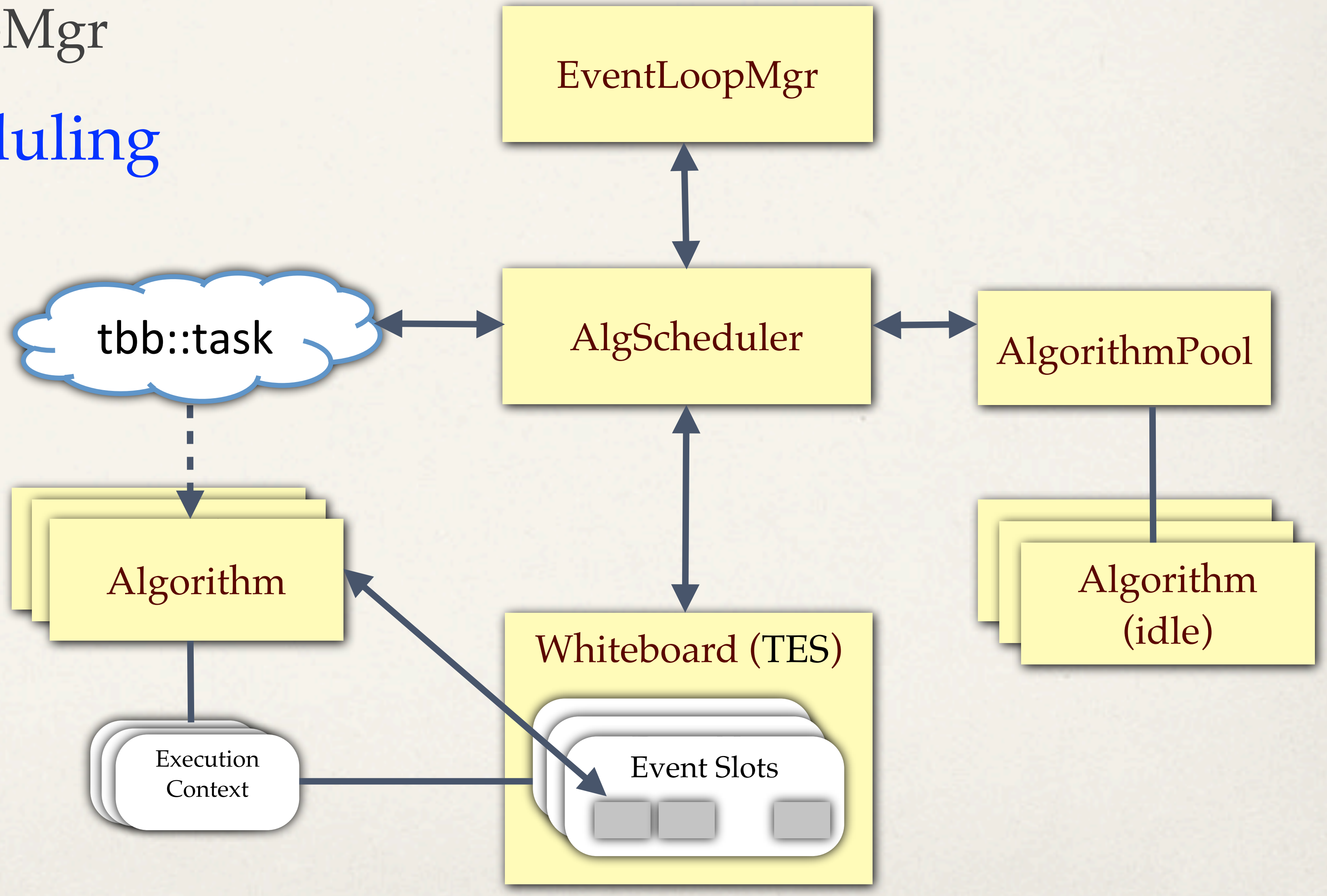
Prototype: GaudiHive

- So far a ‘toy’ Framework implemented using TBB
  - No real algorithms but CPU crunchers
  - Timing and data dependencies from real workflows
- Schedule an *Algorithm* when its inputs are available
  - Need to declare *Algorithms*’ inputs
  - The tbb::task is the pair (Algorithm*, EventContext*)
- Multiple events managed simultaneously
  - Bigger probability to schedule an *Algorithm*
  - Whiteboard integrated in the Data Store
  - Which has been made thread safe
- Several copies of the same algorithm can coexist
  - Running on different events
  - Responsibility of AlgoPool to manage the copies
- Some services have been made thread-safe
  - E.g. TBBMessageService



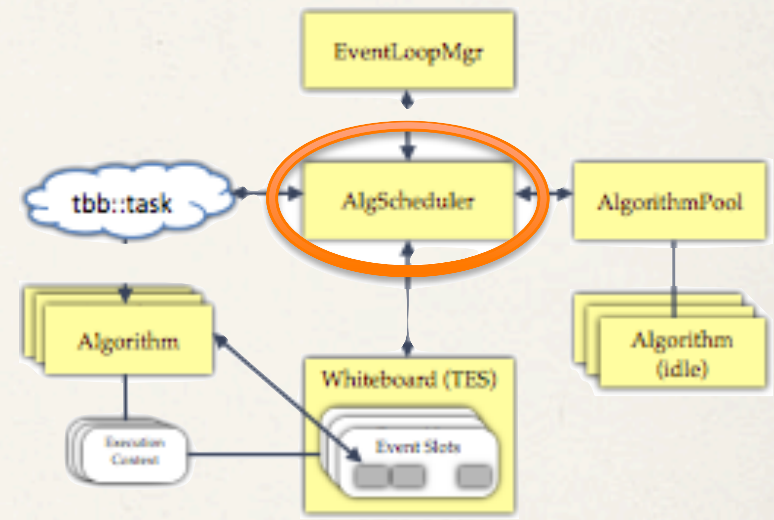
Components Overview

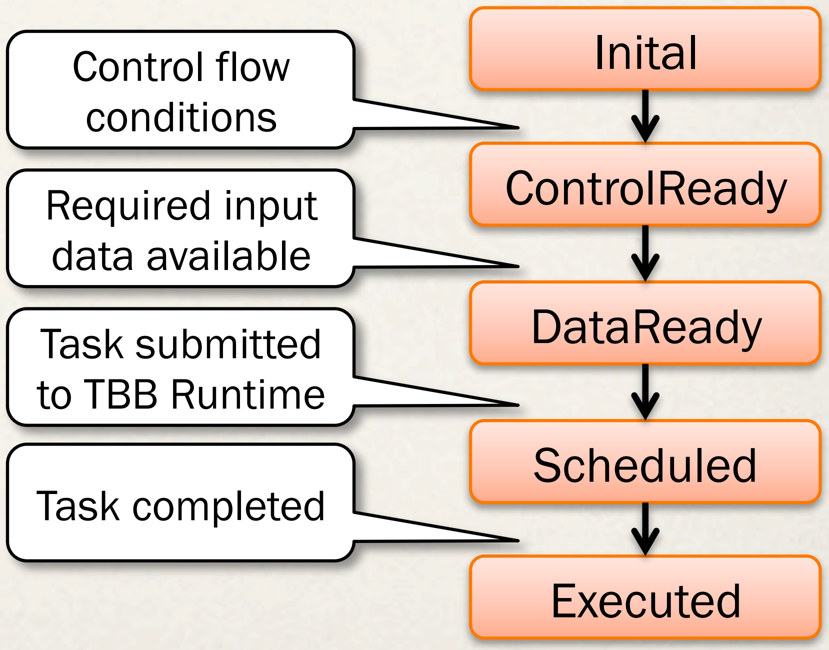
- New components added to Gaudi to support concurrency
  - E.g. Scheduler, Whiteboard, AlgorithmPool
- Existing components upgraded
  - E.g. ToolSvc, EventLoopMgr
- Adopted forward scheduling
  - Schedule an algorithm as soon as its input data are available
- Other other scheduling strategies available as a plug-in



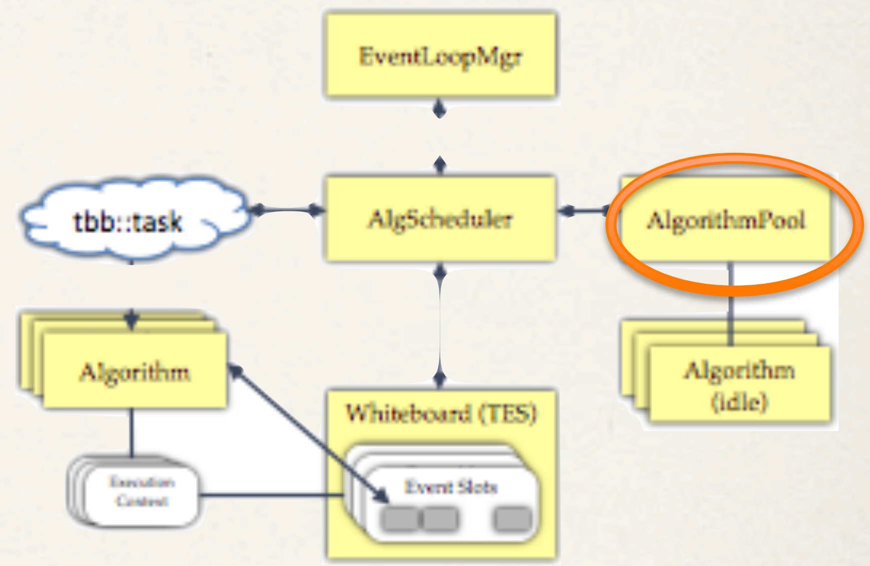
The Forward Scheduler

- Keeps the state of each algorithm for each event
- Simple finite state machine
- Receives new events from loop manager
- Interrogates whiteboard for new DataObjects
- Pulls algorithms from AlgorithmPool if they are available
- Encapsulate them in a tbb::task for execution





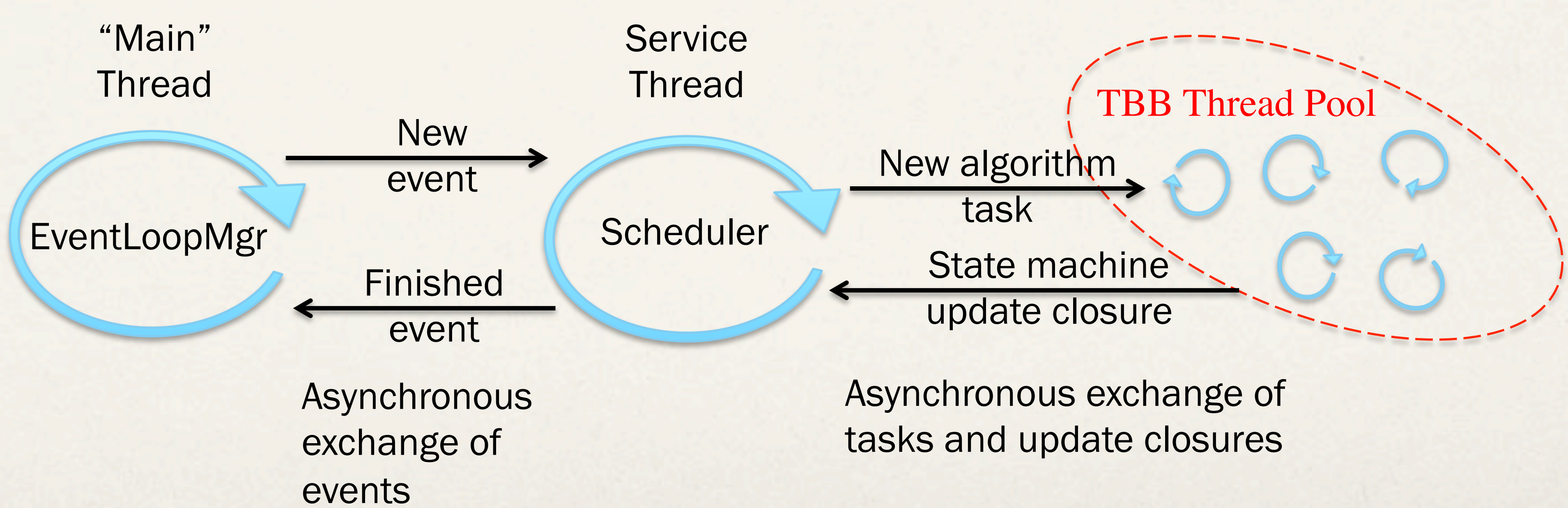
Algorithm Pool



- Contains algorithms and coordinates them
- Gives away instances to run, retrieves finished algorithms
- Clones algorithms (via AlgManager)
- Number depends on code re-entrancy:
  non re-entrant (1 copy only)
  non re-entrant (use n copies)
  fully re-entrant (re-use same instance n times)
- Allows for exclusive resource checking
  e.g. if 2 algos using a non re-entrant external library, only one at the time can run.
- Algorithms' and resources' thread-safety can be tackled one by one

Service Threads

- An additional "service" thread (outside the TBB pool, which contains "worker" threads) is spawned:
  - Host the scheduler method to update the state machine when an algorithm has run. If no work is available, it sleeps.
- The "main" thread manages the event loop ("little more than an event factory")
  - While the scheduler processes the events, it sleeps.
- Other service threads existed and continue to exist (e.g. conditions watchdogs)



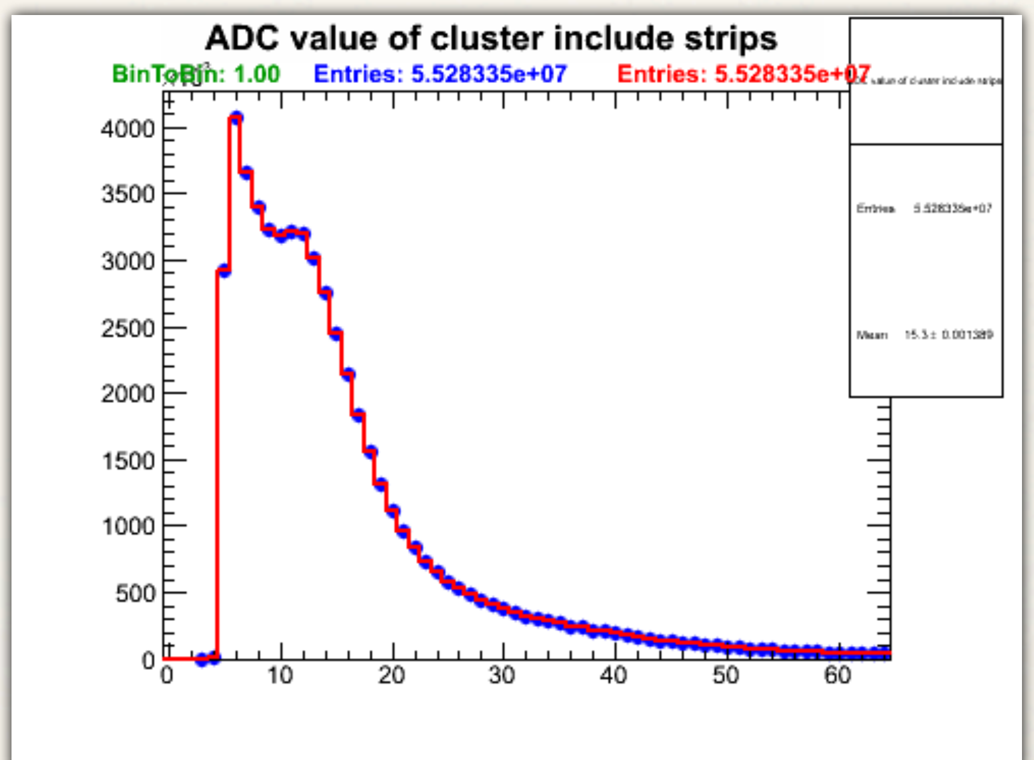
User Code Changes: Executive Summary

- Algorithm dependencies
  - Data dependencies: announced by the algorithms themselves
- Global data structures
  - A few objects served as back-door communication channels bypassing the official (event data) channel
- Fix assumptions of only one event at a time
  - Meaning of many global incidents radically changed (e.g. BeginEvent)
  - Raw Data Conversion Caches and their cleanup

Output Validation

- Only successfully tested software is working software
- Our test case: LHCb standard set of data quality monitoring histograms
- Necessary but not sufficient to guarantee production quality results
- Check histograms for serial and concurrent version (high number of simultaneous events and algorithms)

Example of data monitoring histogram: ADC counts.



All standard histograms identical bin by bin

Does it help with memory consumption?

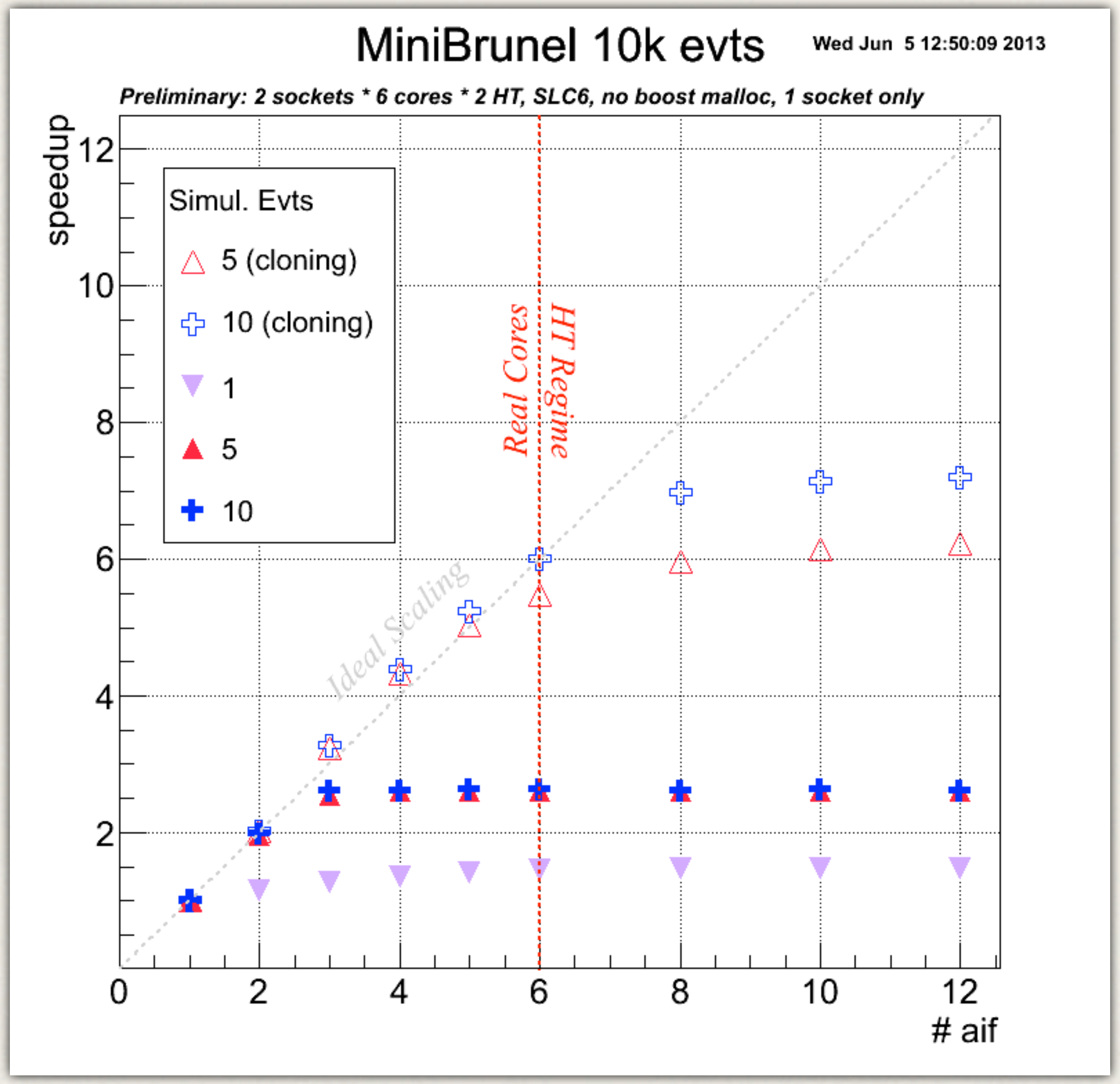
- Running mode:
  - 1 clone per event in flight of 3 longest running algorithms
  - Full TBB thread pool (24 threads)
  - Limit algorithms in flight to 6

- Resident Set Size at the end of the event loop (no finalisation):
  - Serial Gaudi (no new components) ….. 478 MB
  - Concurrent Gaudi 1 evt in flight …….... 480 MB
  - Concurrent Gaudi 2 evts in flight ……. 485 MB
  - Concurrent Gaudi 10 evts in flight …... 514 MB
  - Note: Not full LHCb events but MiniBrunel events.

6 algorithms running simultaneously

Memory: multi-threaded solution is cheap!

Scaling on One Processor



Multiple events in flight
Clone 3 most time consuming algs (1 copy per event in flight)

Linear scaling of speedup up to number of physical cores

10 events in flight already enough for peak performance (thanks to HT)

What's Next

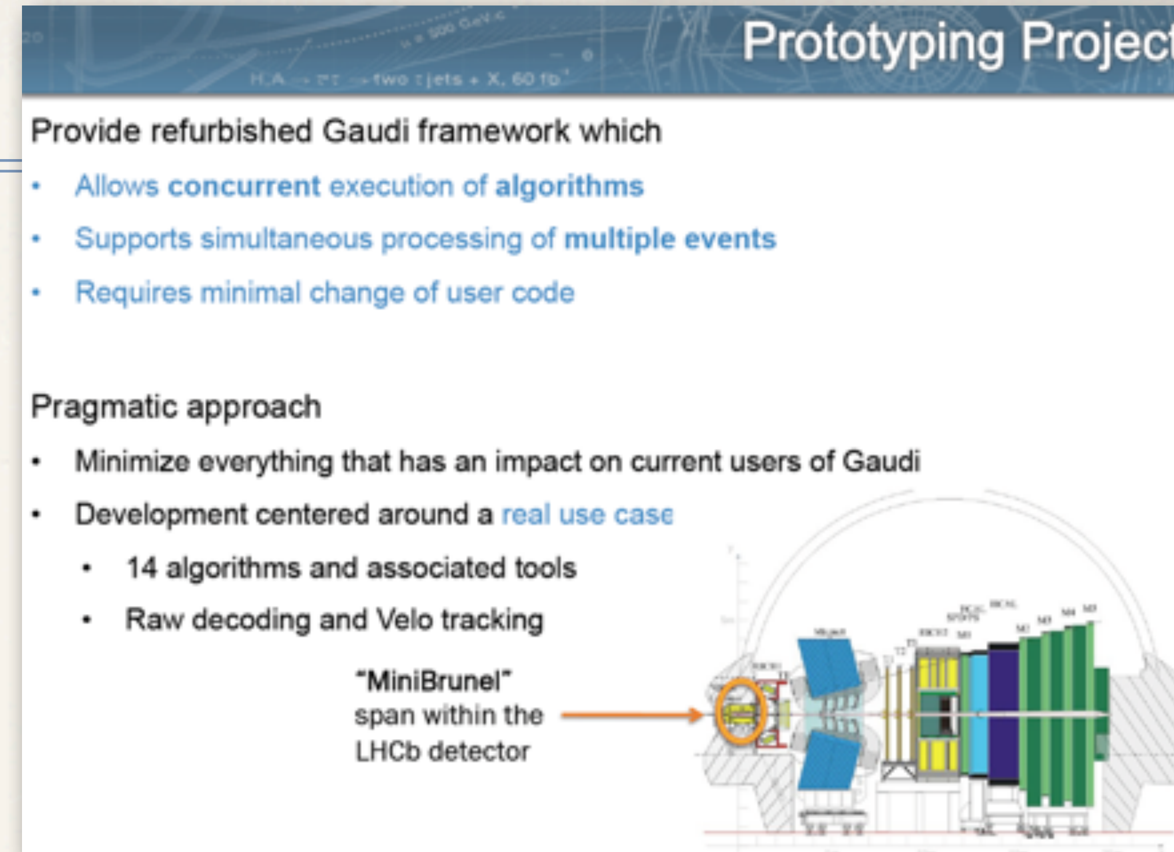
- Initial development of a concurrent framework prototype
  - Smooth evolution for the Gaudi framework
  - Supporting concurrency at all levels (intra-algorithms, algorithms, events)
  - Minimal changes to 'user' code
- Outcome of real-world test very successful
  - Sequential and Concurrent Mini-Brunel yield identical physics output
  - Concurrent MiniBrunel scales linearly on a single die
  - Negligible increase of memory consumption
- Future activities
  - Extend the test scenario to a bigger LHCb example (full reconstruction)
  - ATLAS is caching up with Mini-Reco
  - Complete the set of thread-safe classes and implementation patterns
  - Develop compete benchmarks

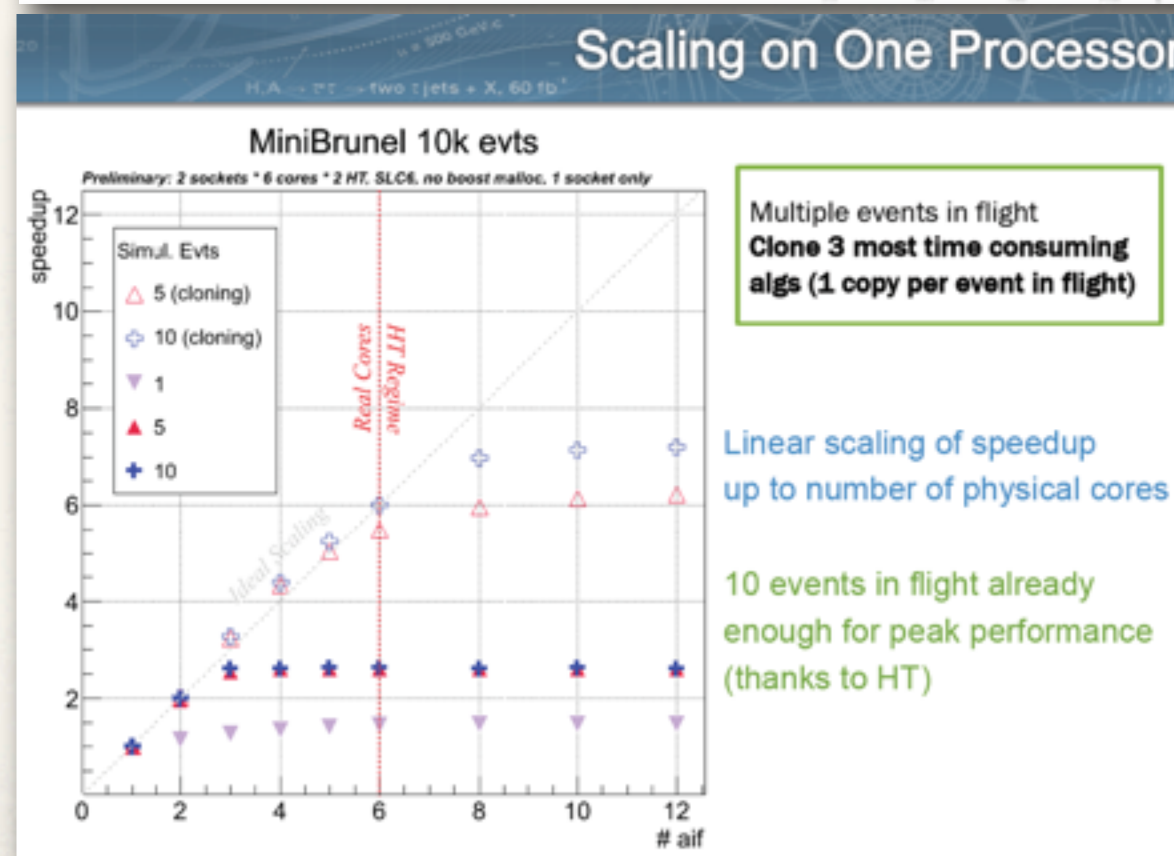
Other Data Processing Frameworks (presented at CHEP)

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

GaudiHive

- Refurbished Gaudi framework for concurrency
    - Supporting concurrency at all levels
- Finished all developments necessary for the test case
    - Framework: components for MT execution (Scheduler, EventLoopManager) and integration with TBB runtime
    - "User" code: input declaration, thread-safety fixes, compatibility with >1 event simultaneously processed
- Outcome very successful
    - Serial and concurrent Mini-Brunel yield identical physics output
    - Concurrent Mini-Brunel scales linearly on a single die
    - Negligible increase of memory consumption

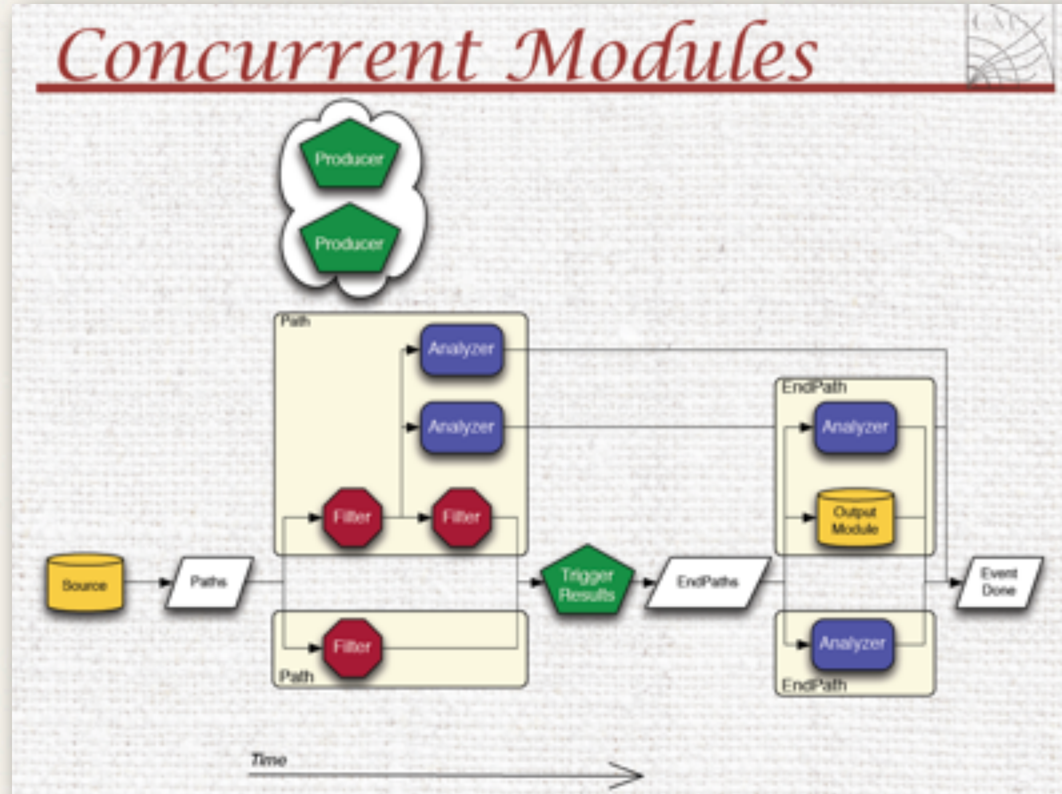


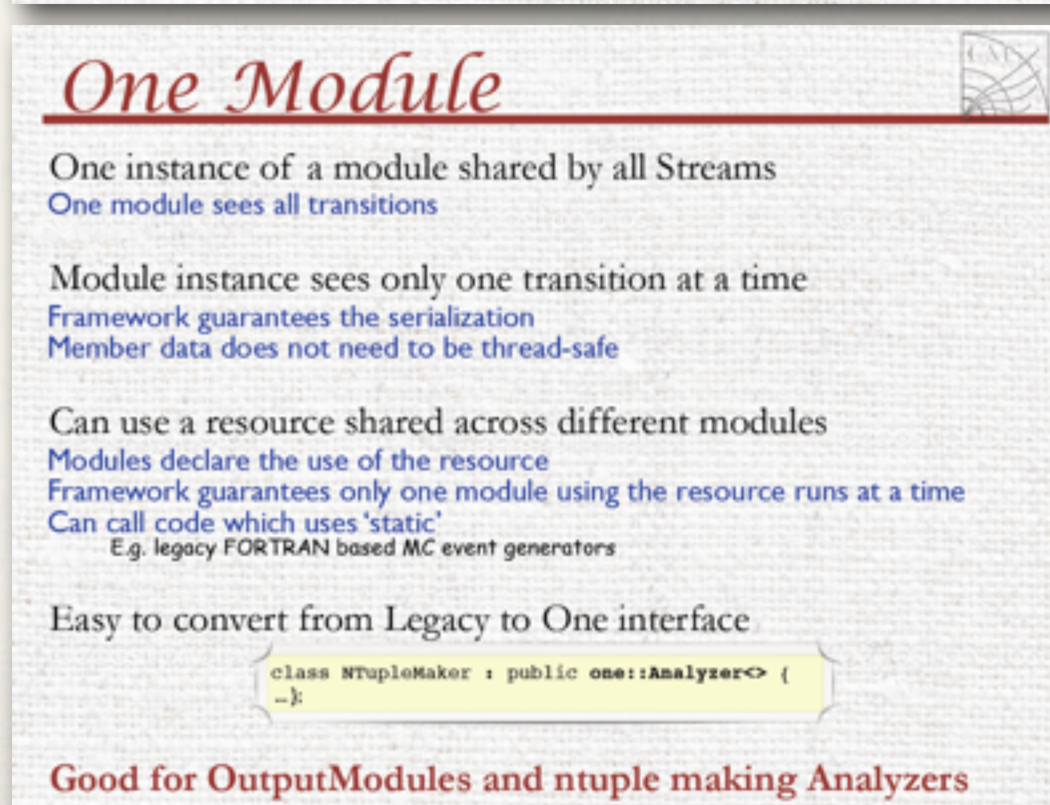


CMS Threaded Framework

E. Sexton-Kennedy, C. Jones

- Better scaling of system resources as core count increases
  - memory, I/O buffers, files, ...
- Minimize changes to existing framework and user facing interfaces
- The design allows many different levels of concurrency
  - Events, modules and sub-module
  - TBB based
- Thread-safety
  - Thread-unsafe code is allowed via 'One' module variety
  - Framework guarantees serialization
- Need tools to find thread-safety issues
  - Clang static analyzer, Helgrind

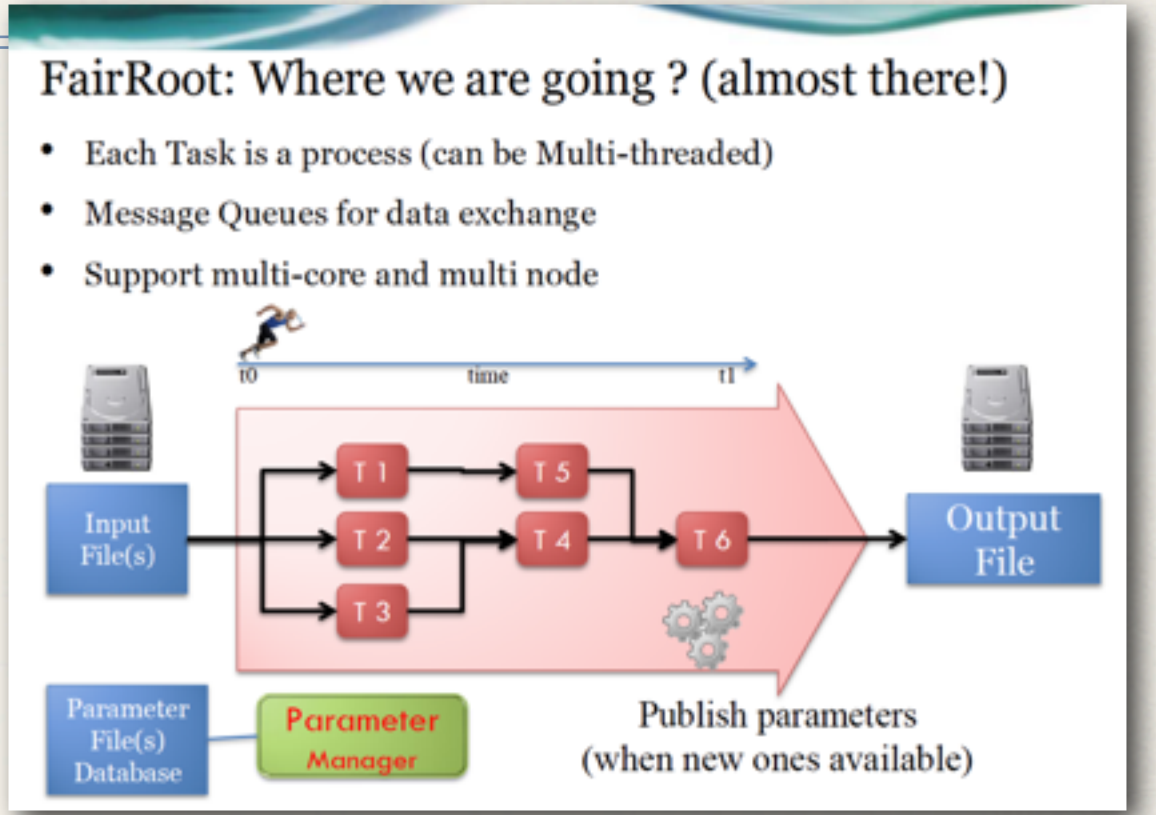


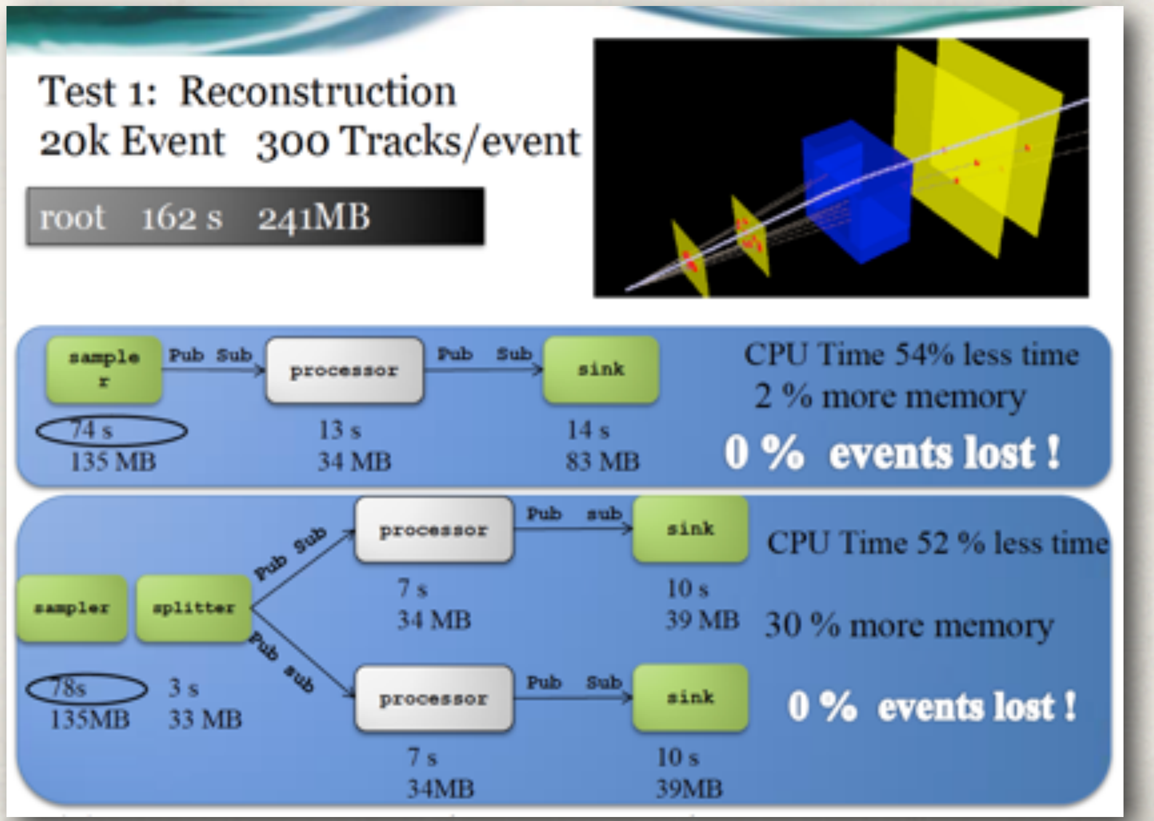


M. Al-Turany et al.

FairROOT data streaming

- Introduced pipelined data processing to the current FairROOT Framework
- Multithreaded concept or a message queue based one?
  - Message based systems to decouple producers from consumers
  - Work spread over several processes and machines
- ZeroMQ provides efficient transport options
  - No need to re-invent the wheel
- The Framework delivers some components which can be connected to each other in order to construct a processing pipeline(s).

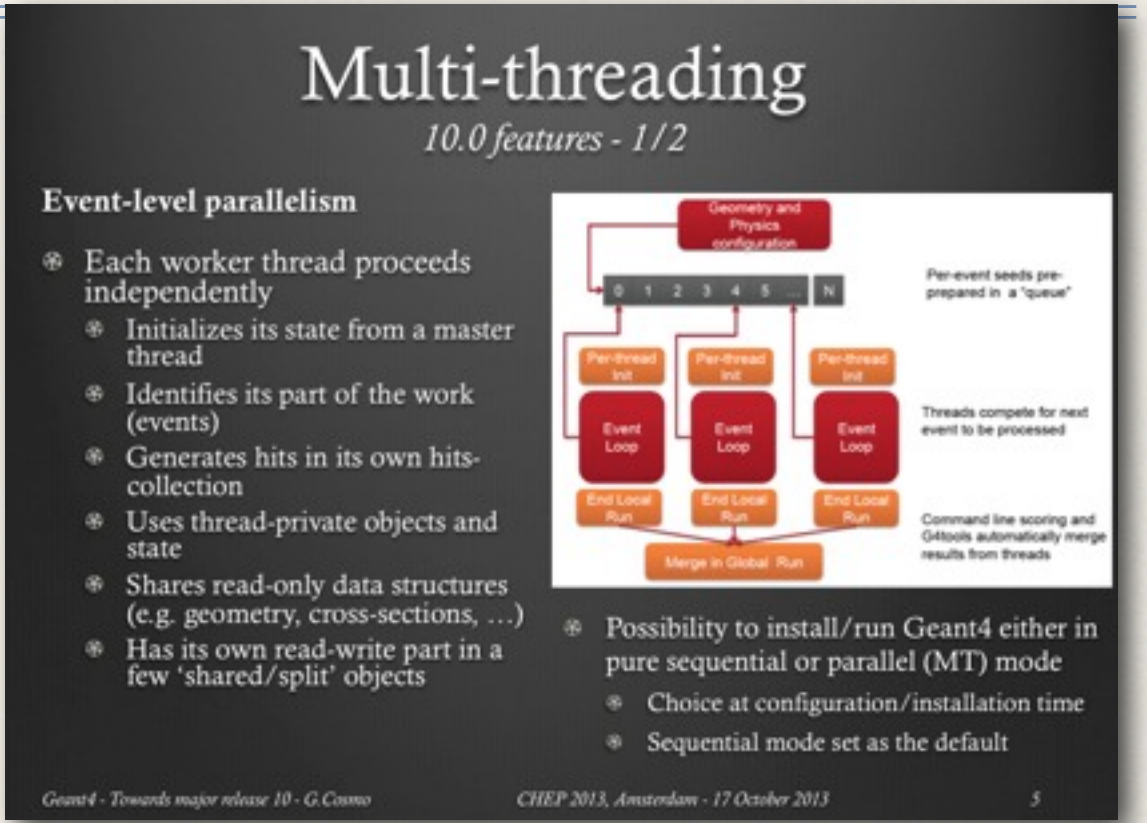


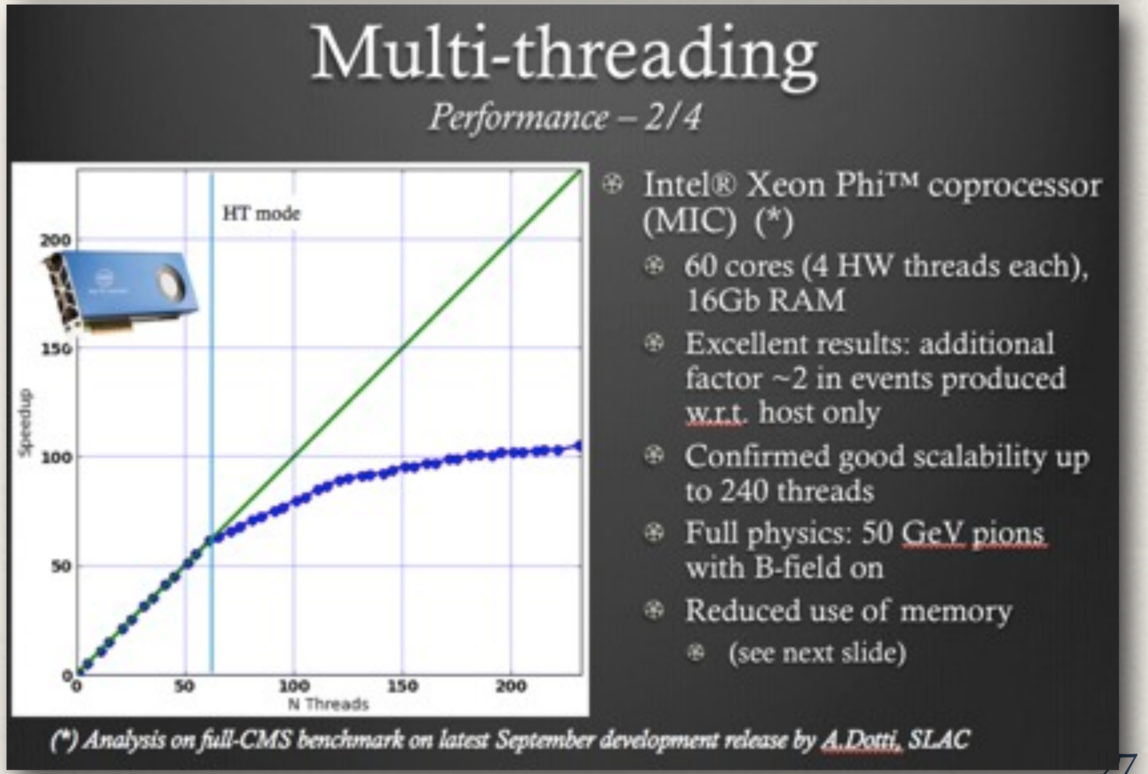


Geant4 MT

G. Cosmo et al.

- Adaptations to thread-safety for event-level parallelism
  - Capitalizing the work started back in 2009
  - Final release version 10 expected for December 6th
- Showing good efficiency w.r.t. excellent linearity vs. number of threads (~95%)
  - From 1.1 to 1.5 extra gain factor in HT-mode on HT-capable hardware
- No measured CPU degradation vs. sequential runs

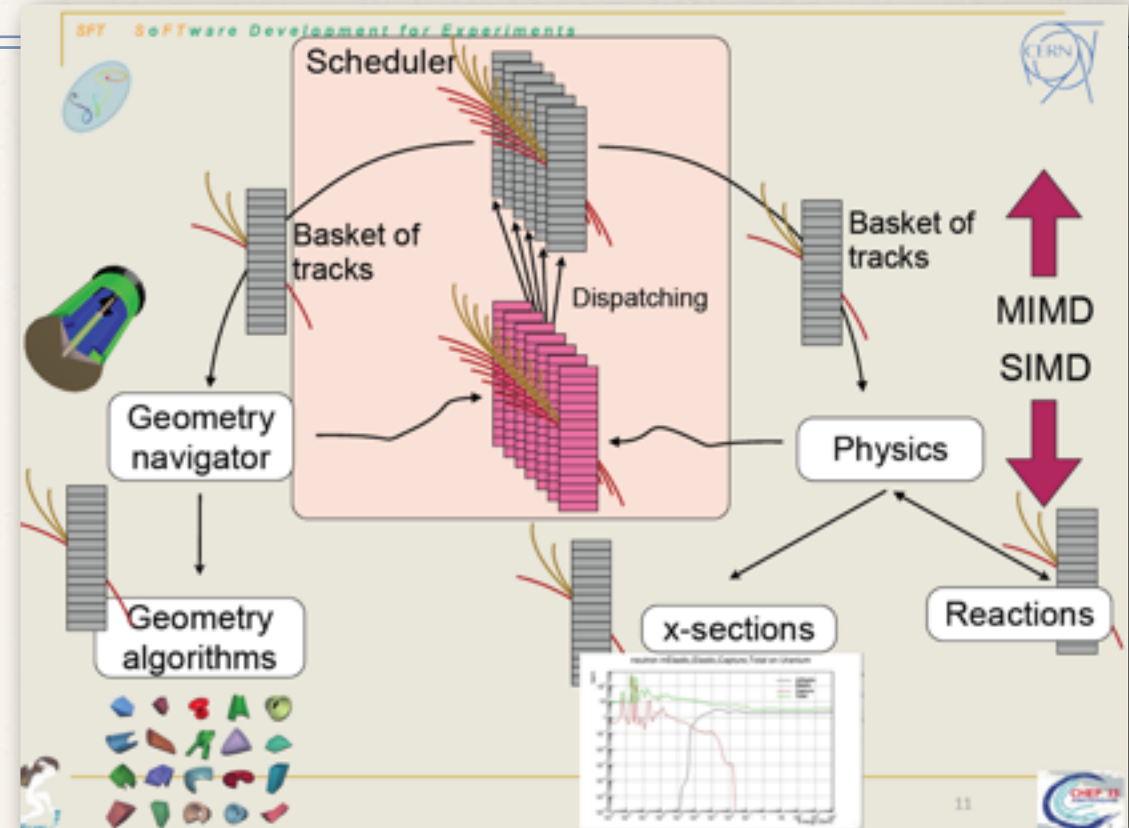


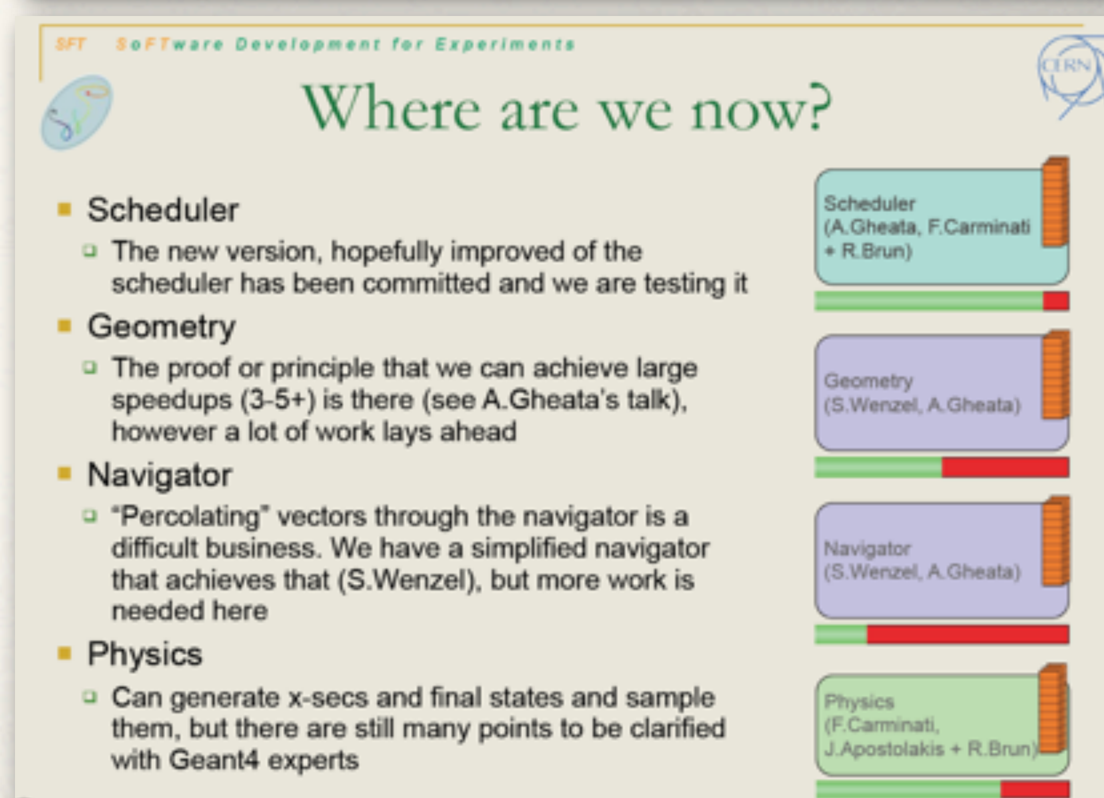


Geant V Prototype

F. Carminati et al.

- Simulation is the ideal primary target for investigation for its relative experiment independence and its importance in the use of computing resources
  - Scheduling the transport of ‘baskets’ of particles
- The Geant Vector project aims at demonstrating substantial speedup (3-5+) on modern architectures
- The work is done in close collaboration with the stakeholders and with Geant4

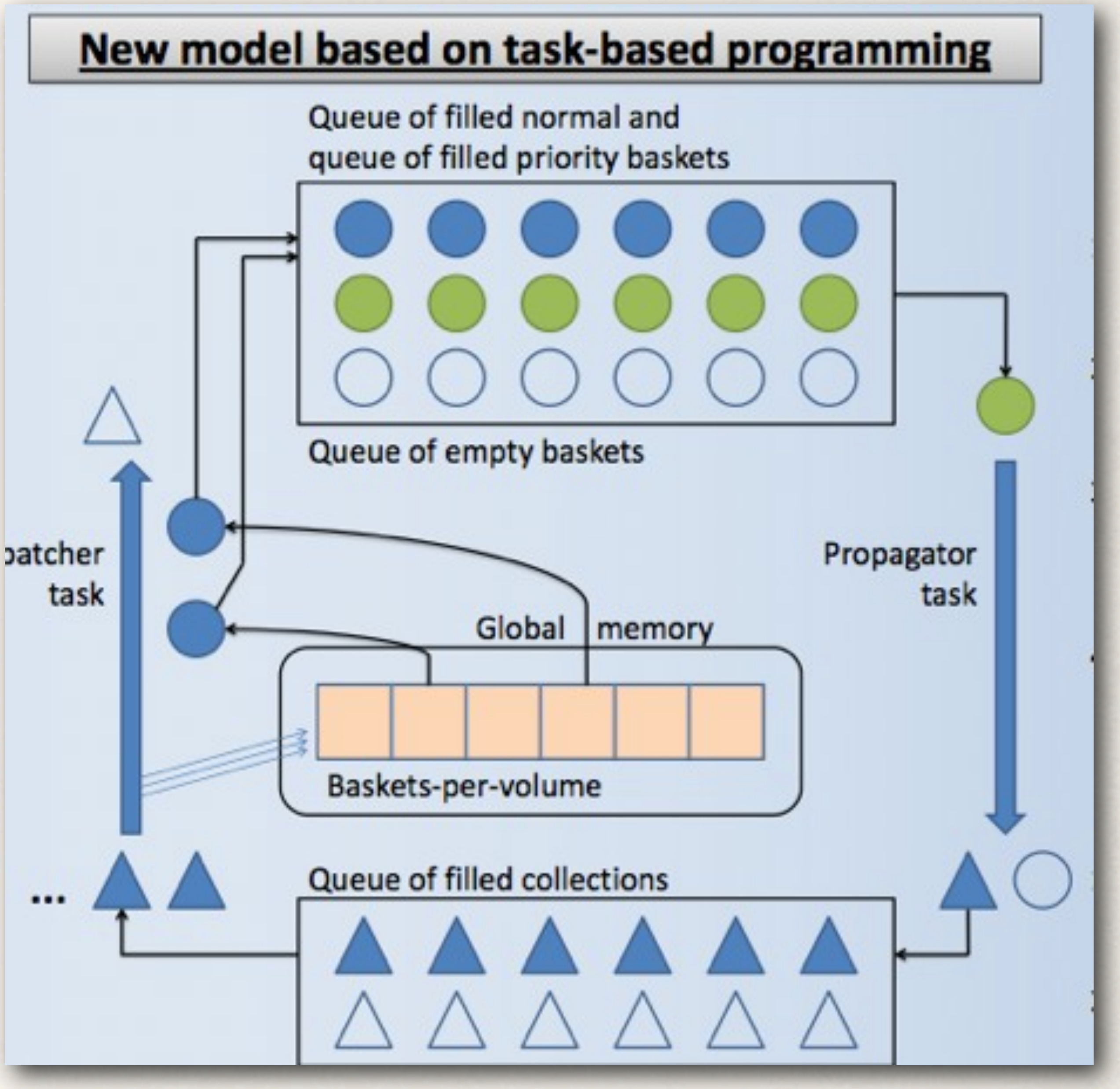




Scheduling Particle Transport with TBB

E. Ovcharenko et al.

- Replacing 'particle basket' scheduler in Geant V prototype with TBB
- Results
  - Performance and behavior of the new prototype is close to the old prototype scheduler
  - There are some features that need to be further understood
    - unexpected increase of cache misses
    - comparatively low scalability



Concluding Remarks

- Multi-job and multi-process solutions processing one event at the time give us good service and will continue for a long while
  - Hungry on resources (memory, open files, DB connections, etc.)
  - File merging problem
- We need to start embracing the next generation applications with finer-grain concurrency
  - Reduces memory and number of required resources
  - Pre-requisite for offloading to heterogenous resources
- In parallel we need to ‘vectorize’ our libraries and algorithms to make efficient use of SIMD instructions available in modern processors
- Most of the scientific software and algorithms was designed for sequential processor in use for many decades and **will require significant re-engineering**
- The community needs to **develop expertise in concurrent programming**
  - Sharing experiences, successes and failures is essential this early exploratory phase
  - The **Concurrency Forum** tries to address these needs