Evolving LHC Data Processing Frameworks for Efficient Exploitation of New CPU Architectures

IEEE Nuclear Science Symposium (NSS) 2012, Anaheim, USA

B. Hegner, P. Mato, D. Piparo

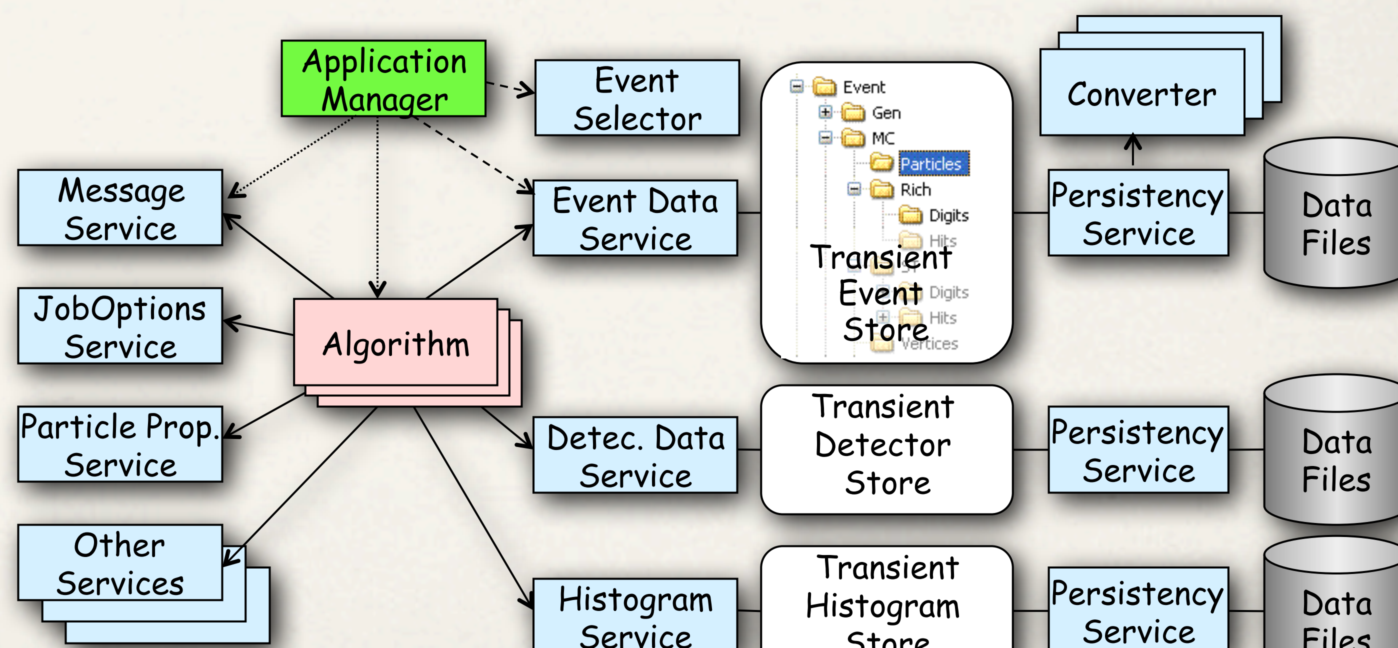
1/11/2012

Contents

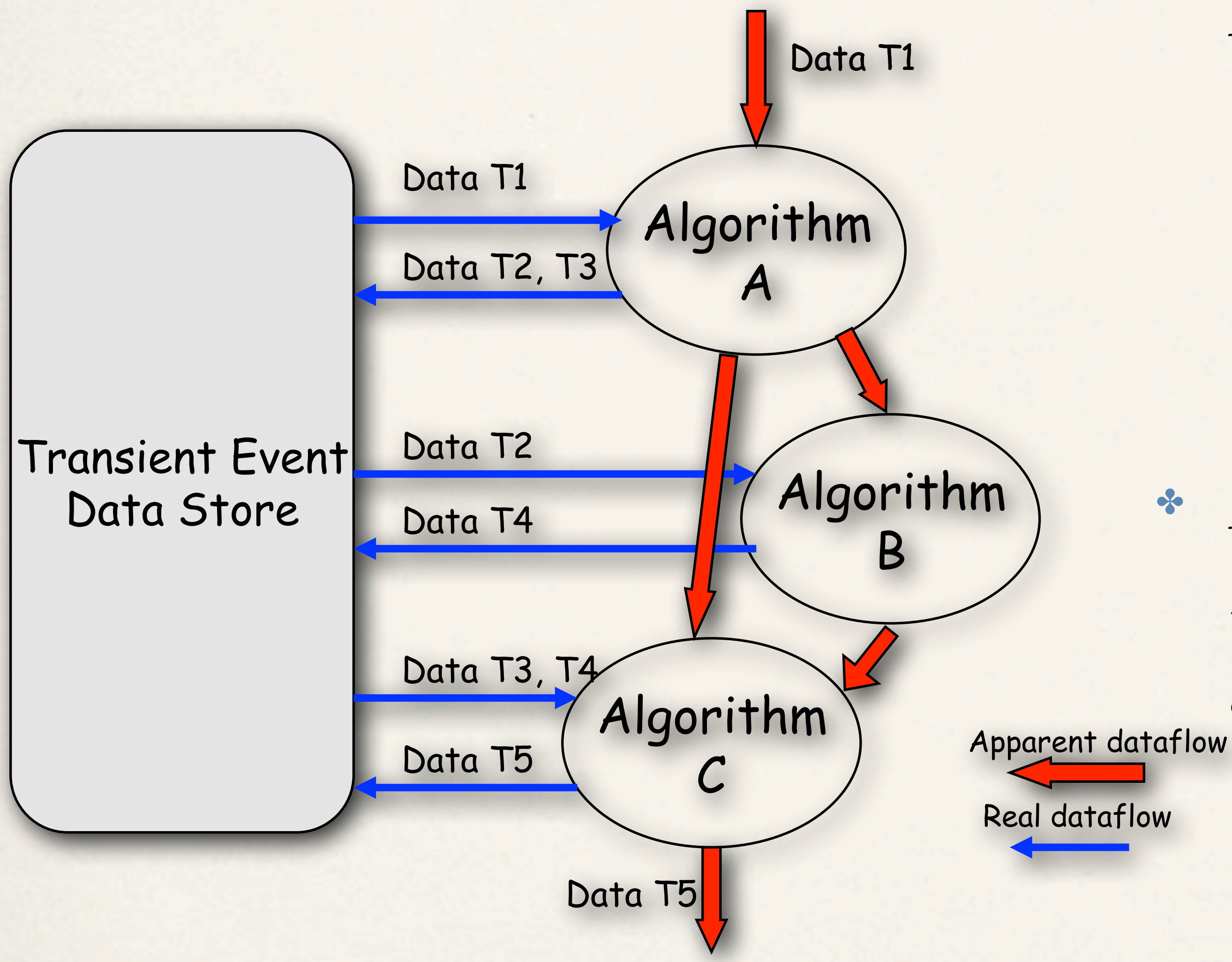
HEP Software Frameworks

- 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



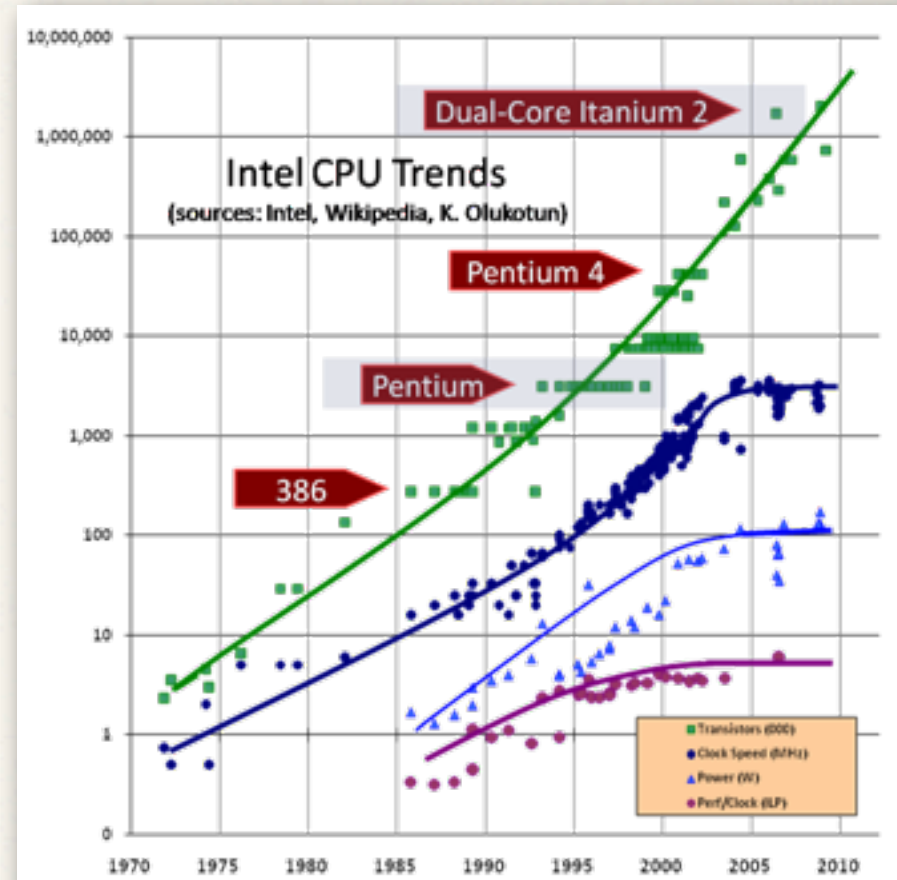
Algorithms and Data Flows



- 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



© 2009 Herb Sutter

- 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

Time for a New Framework

- For the last 40 years HEP event processing frameworks have had the same structure
  - initialize; loop over events {loop over modules {…} }; finalize
  - O-O has not added anything substantial
  - It is simple, intuitive, easy to manage, scalable
- Current frameworks designed late 1990’s
  - We know now better what is really needed
  - Unnecessary complexity impacts on performance
- Clear consensus that we need to adapt HEP applications to new generation CPUs
  - Multi-process, multi-threads, GPUs, vectorization, etc.
  - The one job-per-core approach will fail soon due to demanding too much memory and sequential file merging

Why Concurrency?

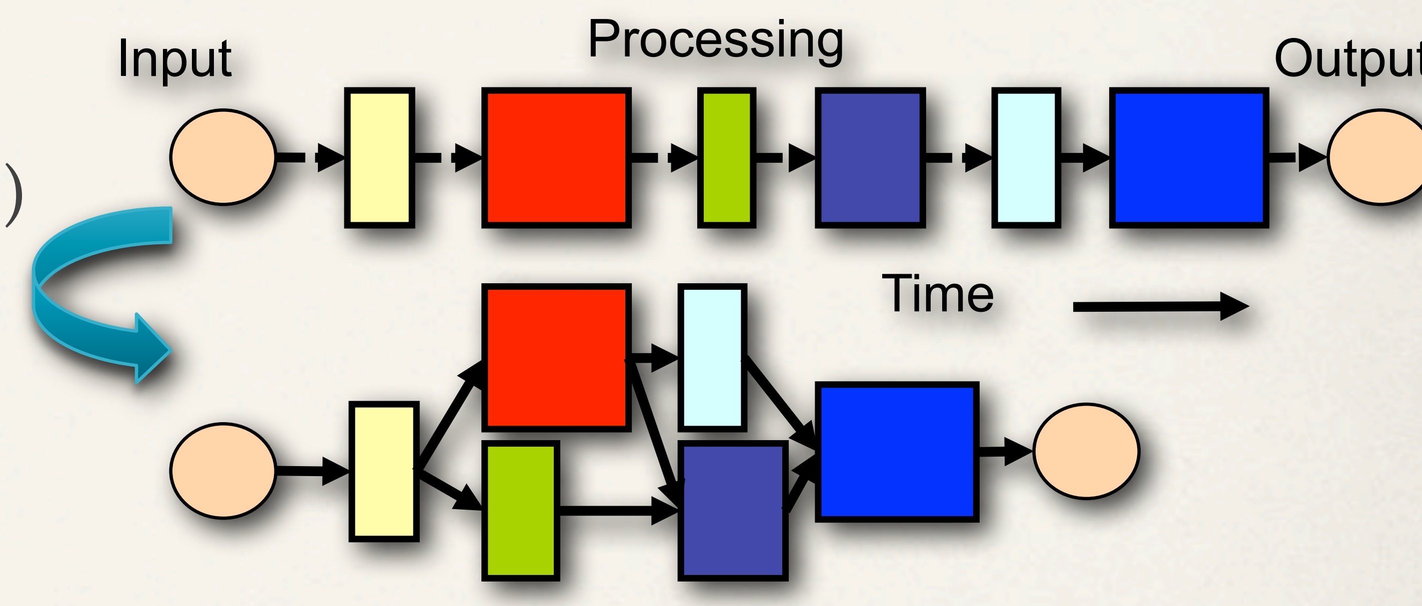
- We need to adapt current data processing applications to the new many-core architectures (~100 cores)
  - No major change is expected in the overall throughput with respect to trivial one-job-per-core parallelism with today core counts
- We must reduce the required resources per core to avoid real 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

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 to code the algorithms
- At the **Framework level** you have the full overview and control of the application
  - Controlling the access to critical shared state
  - The framework may decide to run some parts of the code sequentially

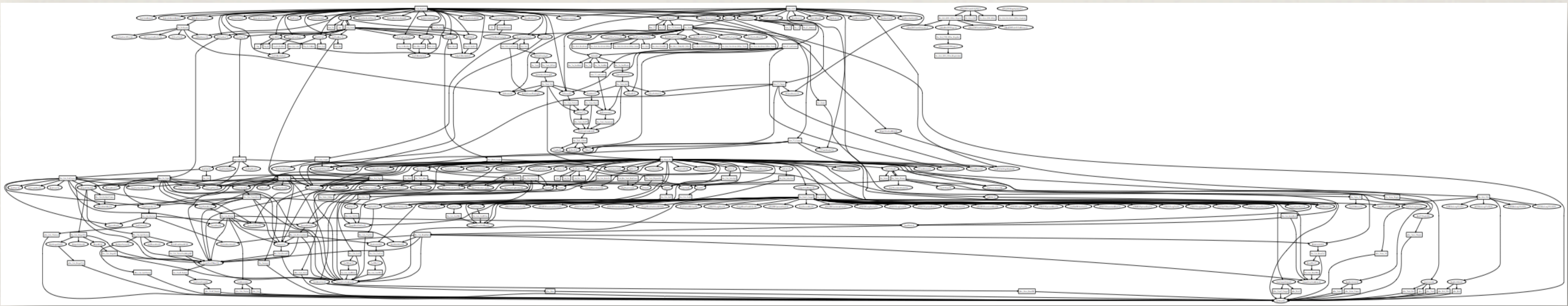
Concurrent 'Algorithm' processing

- Ability to schedule modules/algorithms concurrently
  - Full data dependency analysis would be required (no global data or hidden dependencies)
  - Need to resolve the DAGs (Directed Acyclic Graphs) statically and/or dynamically

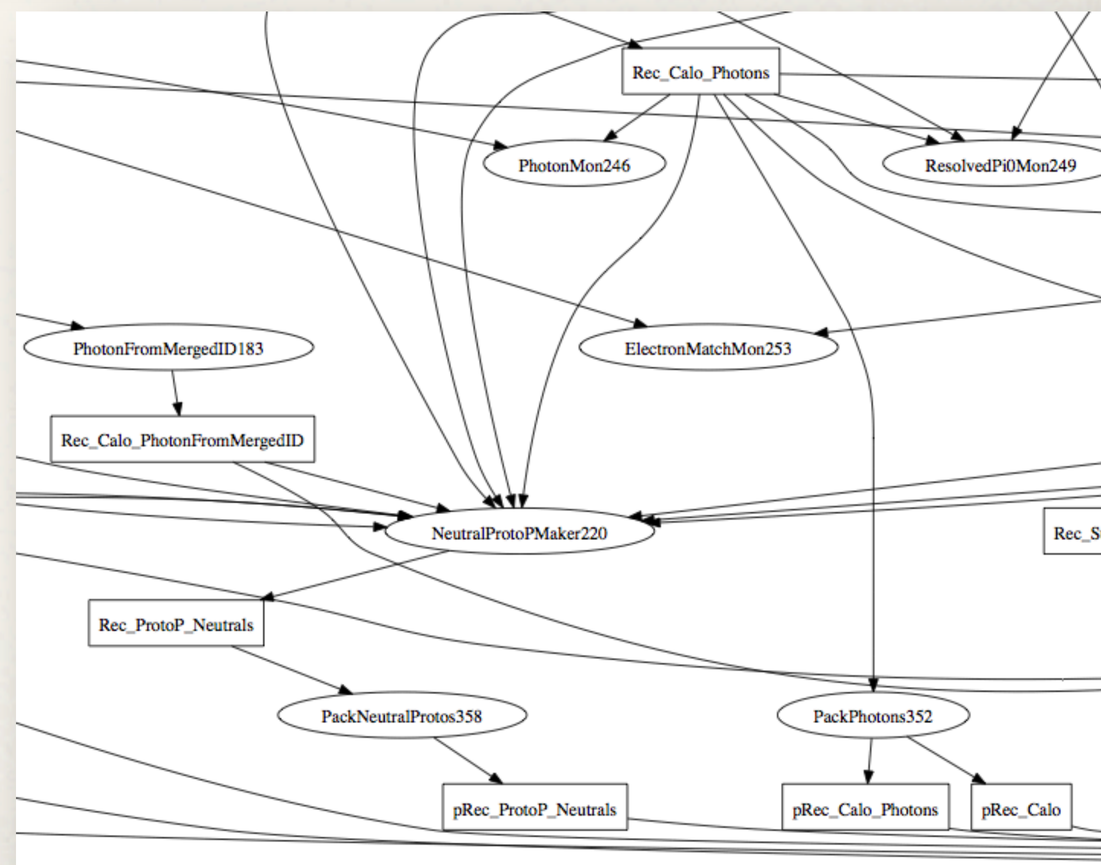


- Unfortunately with today's existing *Algorithms* we cannot use efficiently ~100 cores
  - Estimated concurrency factor rather low for CMS and LHCb (between 3 and 6)

Example: LHCb Reconstruction



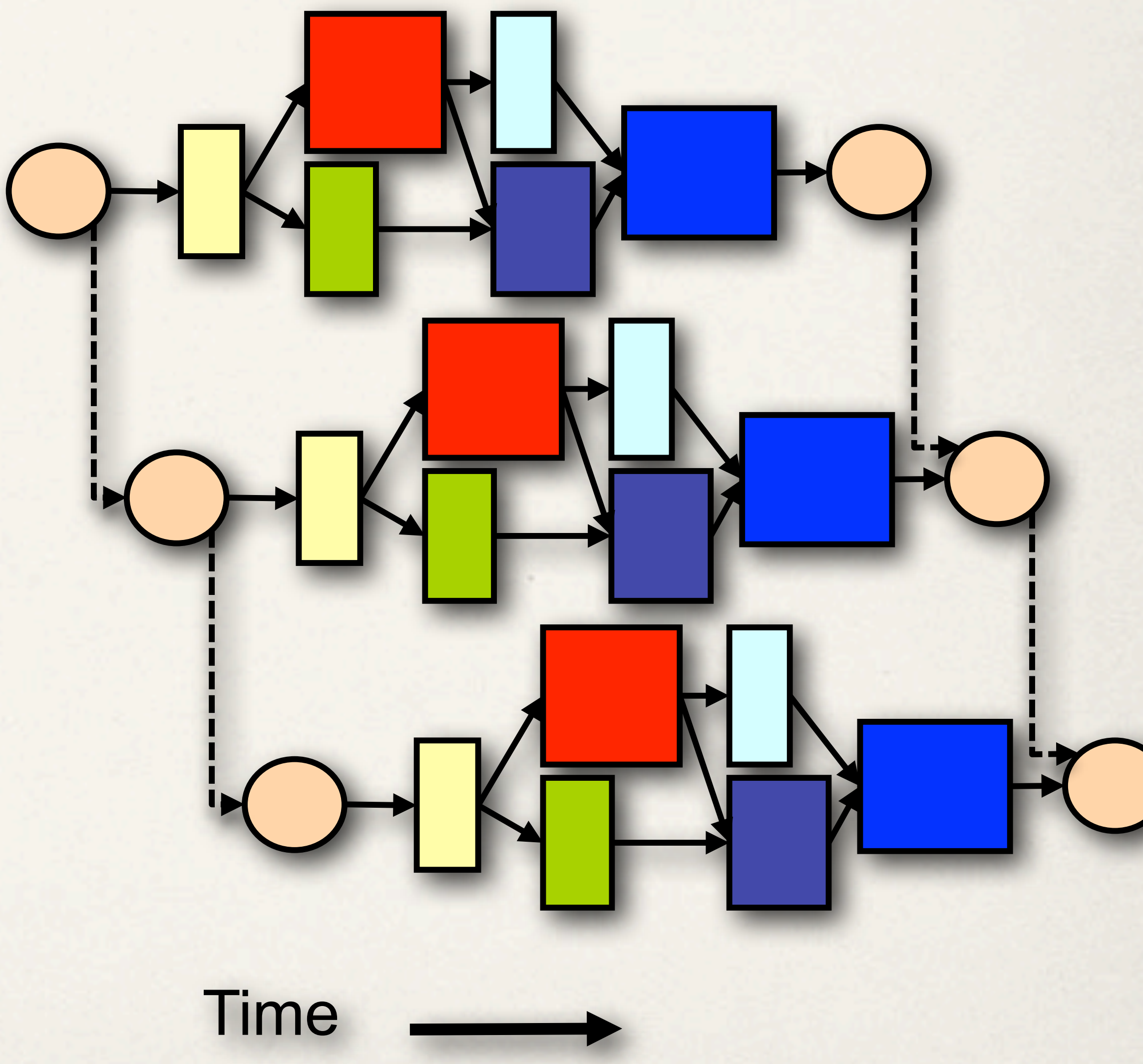
- DAG of Brunel (214 Algorithms)
  - Obtained by instrumenting the existing sequential code
  - Probably still missing 'hidden or indirect' dependencies



- 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 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*



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

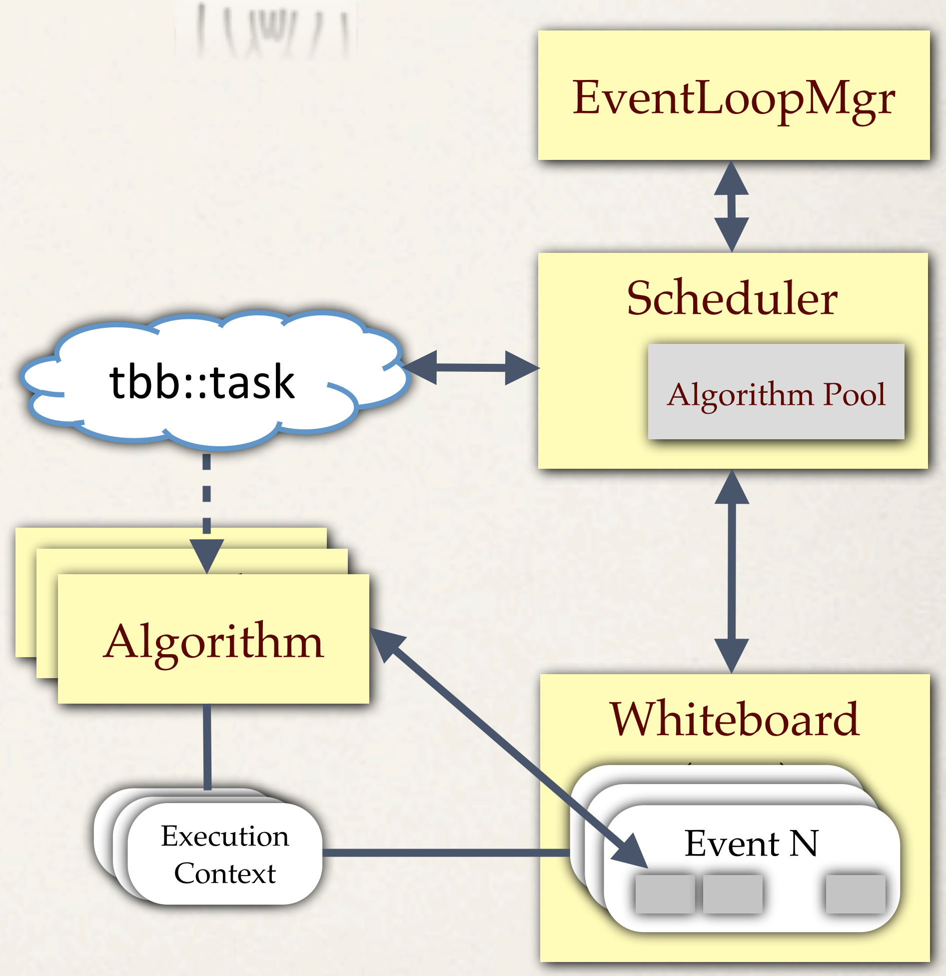
http://concurrency.web.cern.ch

TBB Technology

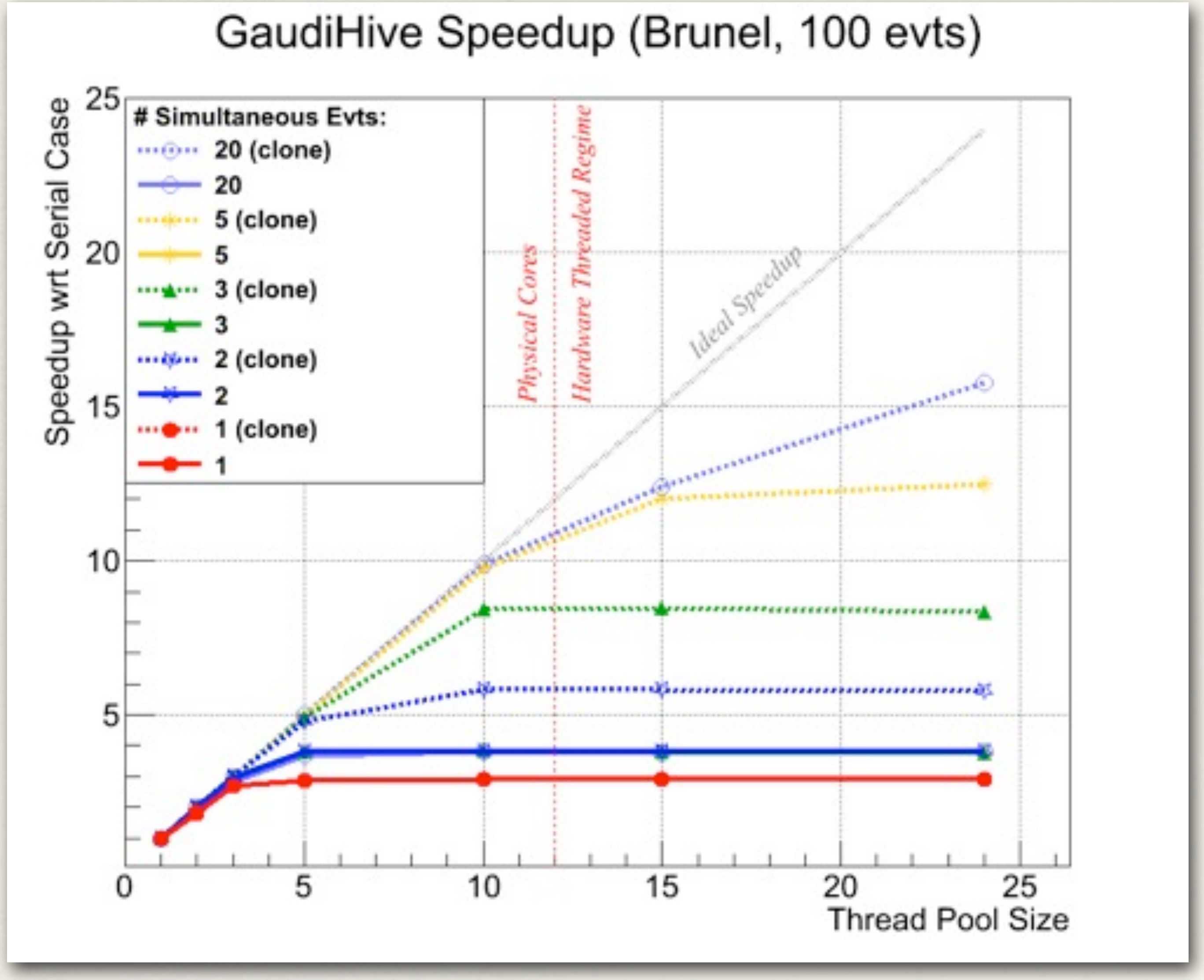
- Intel® Threading Building Blocks (TBB) has been identified as a good match for implementing concurrency at the Framework level
- C++ library with a rich and complete approach to express parallelism
  - Concurrent containers: concurrent_vector, concurrent_hash_map, ...
  - Algorithms: parallel_for, 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

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



Test On Brunel Workflow

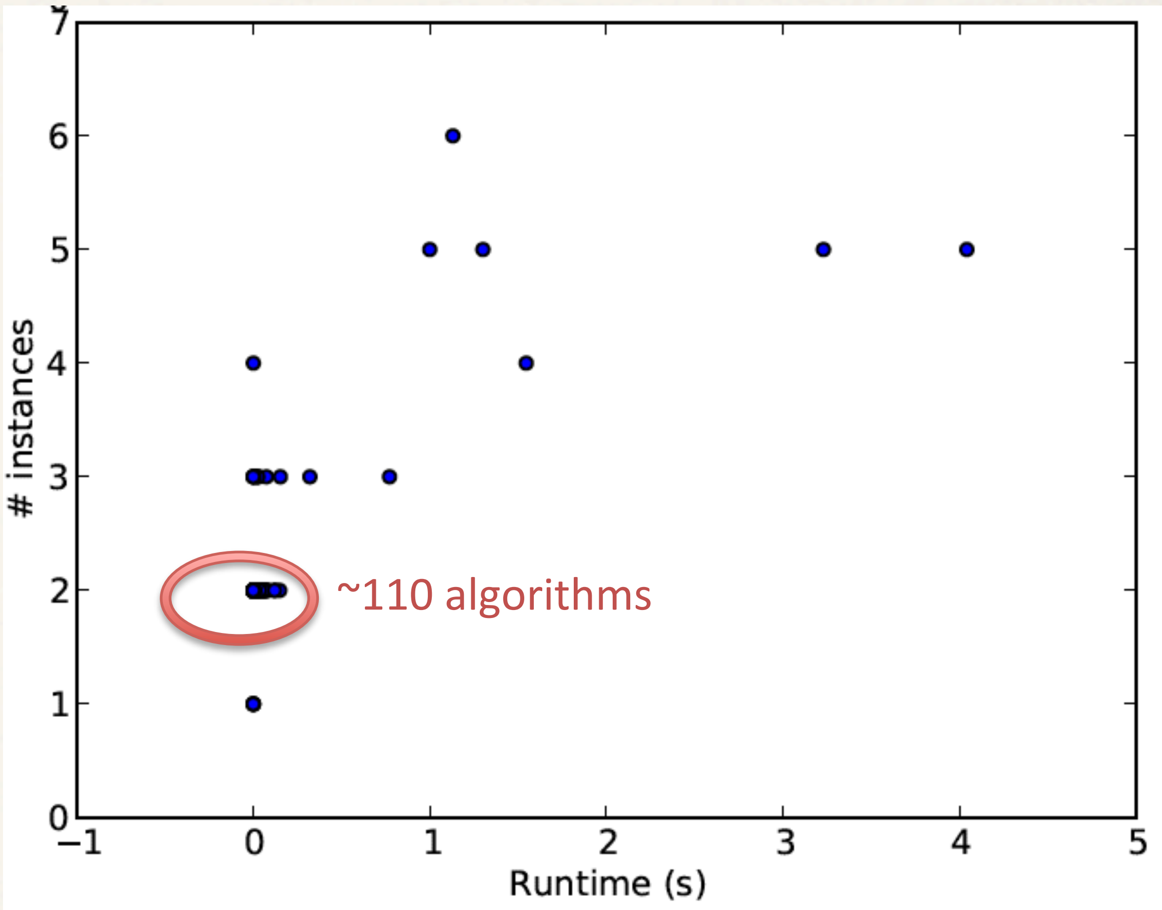


Test system with 12 physical cores x 2 hardware threads (HT)

- 214 Algorithms, real data dependencies, (average) real timing
  - Maximum speedup depends strongly on the workflow chosen
- Adding more simultaneous events moves the maximum concurrency from 3 to 4 with single *Algorithm* instances
- Increased parallelism when cloning algorithms
  - Even with a moderate number of events in flight

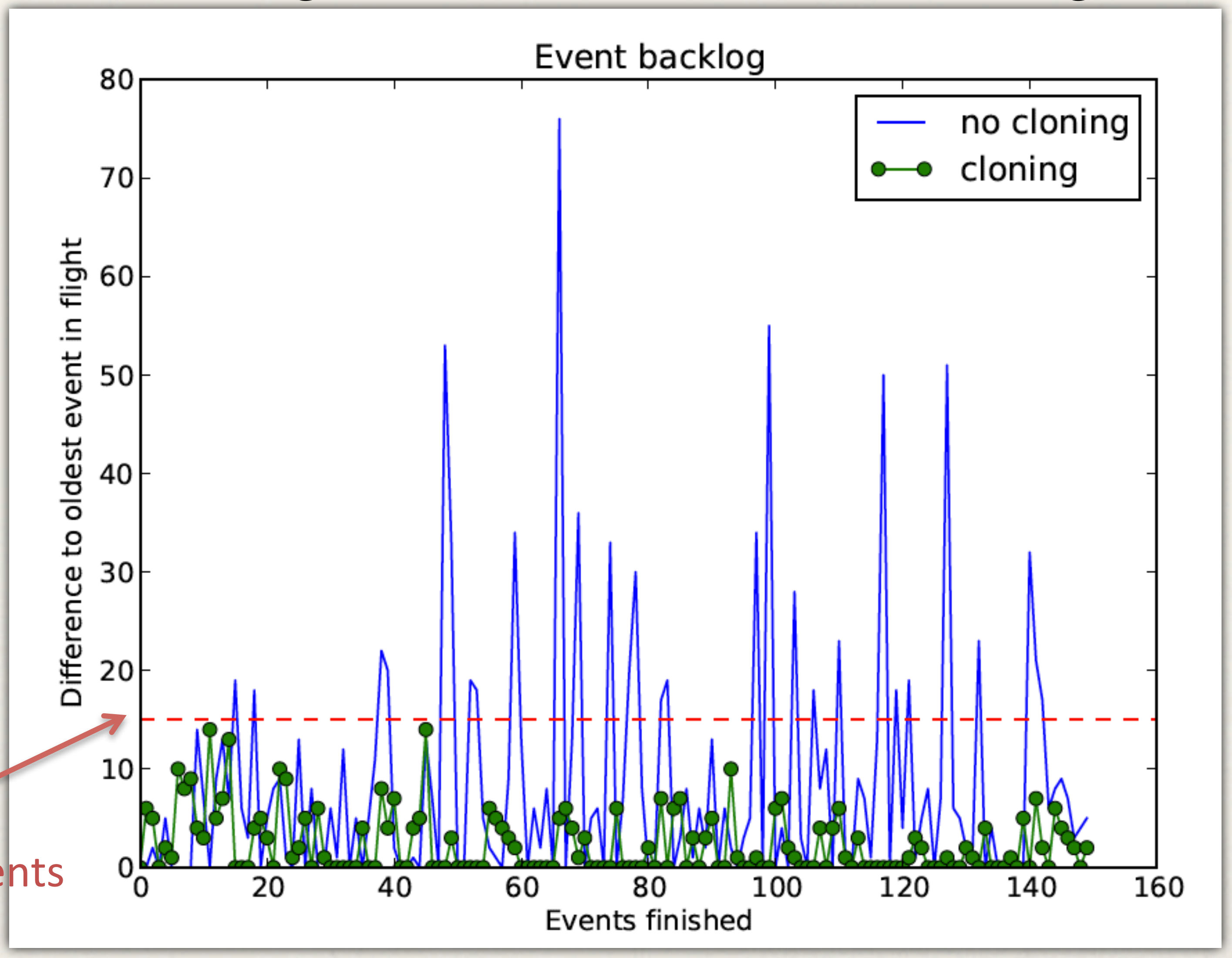
Clones vs. Runtime

- Tested strategy
  - *Algorithm* cloned if it can be scheduled and all its existent instances busy on other events
- Long running algorithms end up having multiple clones
  - Easy solution but we need to worry about statistical outputs (counters, histograms, etc.)
  - Alternatively, these are candidate algorithms to be parallelized
- A high number of short algorithms have 2 copies
  - We may forbid multiple copies for those without probably reducing achievable parallelism



Event Backlog

- Event backlog: difference between latest event put in flight and oldest event being processed
- Cloning helps maintaining a little event backlog
- Cloning increases throughout, but as well results in guaranteed latencies



Number of events in flight

Concurrent Gaudi: Status

- A prototype of a concurrent Gaudi (**GaudiHive**) has been developed as an evolution (new branch in the Gaudi git repository)
  - Able to schedule and run **algorithms concurrently**
  - Able to run **multiple events simultaneously**
  - Friendly with **sub-event parallelism** if using TBB (not tested yet)
- So far has been tested with "fake" BRUNEL reconstruction workflow:
  - Important speedup already been obtained, but no "perfect" scaling achieved yet
  - *Algorithm* cloning increase parallelism, keeps "latency" under control
- Test bench to exercise timings and dependencies for other applications:
  - CMSSW reconstruction workflow (already there)
  - ATLAS (got preliminary input)

Concurrent Gaudi: Plans

- Continue the investigation about thread unsafe Gaudi elements
  - For example *Services*, public *Tools*, *Incidents*, etc.
- Provide options for their upgrade
  - Multiple copies+merge?
  - Locked-gateway?
- Finding reusable patterns for thread-safe access to shared resources
- Strategy: start running real algorithms
  - Start with subset of LHCb reconstruction (~30 algorithms) including I/O
  - Extend to full workflow later

Conclusions

- Applications will increasingly need to be concurrent if we want to fully exploit the continuing exponential CPU throughput gains
  - Parallelizing the framework spares physicists from developing parallel code and is the natural place to have the full overview and control of the application
- The Concurrency Forum: important results achieved
  - Evaluation of possible common technologies (e.g. TBB)
- Prototype of Gaudi Framework with concurrency has been developed
  - Ideal test-bench for validating scheduling strategies
  - Initial results has been presented
- A clear trend emerged for the future of HEP data processing
  - Parallelism within the algorithms
  - Parallelism among algorithms
  - Parallelism among events