CERN Concurrency Framework Project (CF4Hep)

Danilo Piparo, Pere Mato Vila, Benedikt Hegner
CERN

Outline

- Our Vision
- Current Activities
- Components

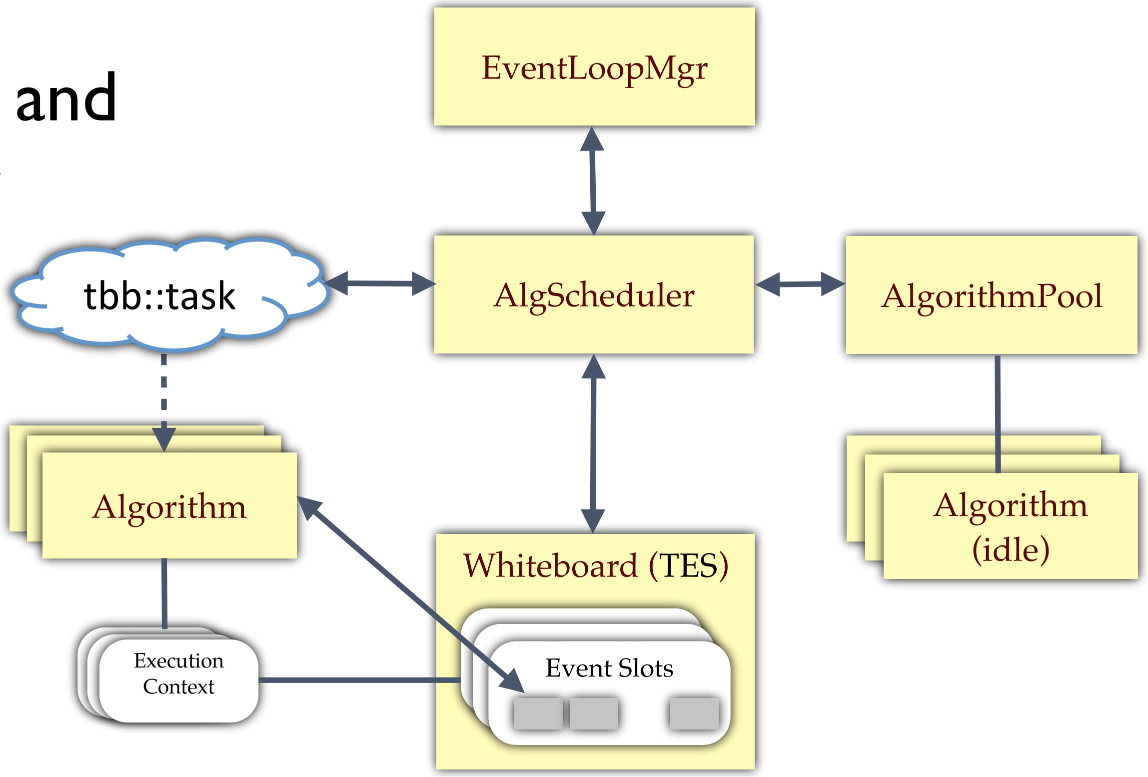
- Summary

Our Vision

- **Develop a full parallel framework for future experiments**
  - Supporting concurrency at multi-event level, among and inside algorithms
    - we think **all** three levels are necessary
  - Robustness over speed (some coarse-grain rather than lots of fine-grain locking)
  - Design based on loosely coupled re-usable components
- **Provide the re-usable components to the LHC experiments**
  - Components are designed as experiment agnostic
  - Only constrains are choice of C++11 and TBB as assisting library
- **Assist physicists in writing proper algorithms**
  - Support them with static code checking and good design patterns
  - Community training to reach required knowledge (C++11, tools, ...)

Current Activities

- **Event-loop component (working title *GaudiHive*)**
  - Forward-scheduling via dependency analysis (i.e. start an algorithm once data there)
  - Rather clear idea about the general design and behaviour after Whiteboard demonstrator
  - Started to work on Gaudi+LHCb reconstruction (Brunel) as test case
    - Concrete migration problems popping up at interesting places



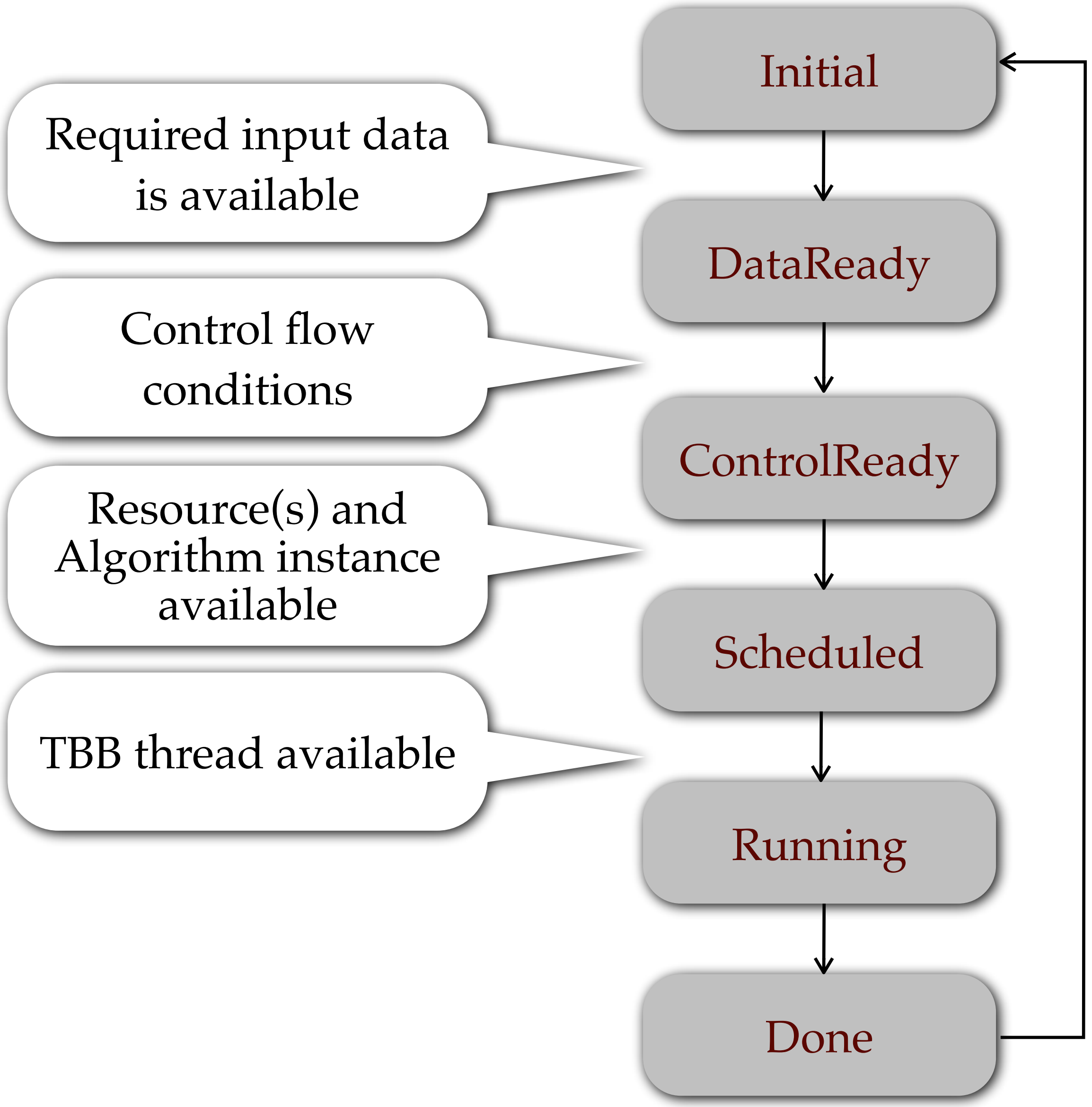
- **(Near) Future**
  - Successfully run a slice of the full reco (*MiniBrunel*)
  - Develop other component prototypes along the way
  - Only after the full exercise we will decide on concrete implementation (we dare throwing away prototypes!)

Highlights

- **Forward scheduling**
  - Forward-scheduling works just perfectly
- **Concurrent access to unique resources**
  - So far we didn't need any special dead-lock risky coding
  - Resource management is currently done at two levels:
    - framework internals via thread-safe data structures and queues
    - User code via a resource pool
      (if an algorithm declares it requires shaky libA, then
      no other algorithm needing shaky libA can be scheduled)
- **Synchronization within the Framework**
  - Message queues with a listener thread waiting behind

Scheduler

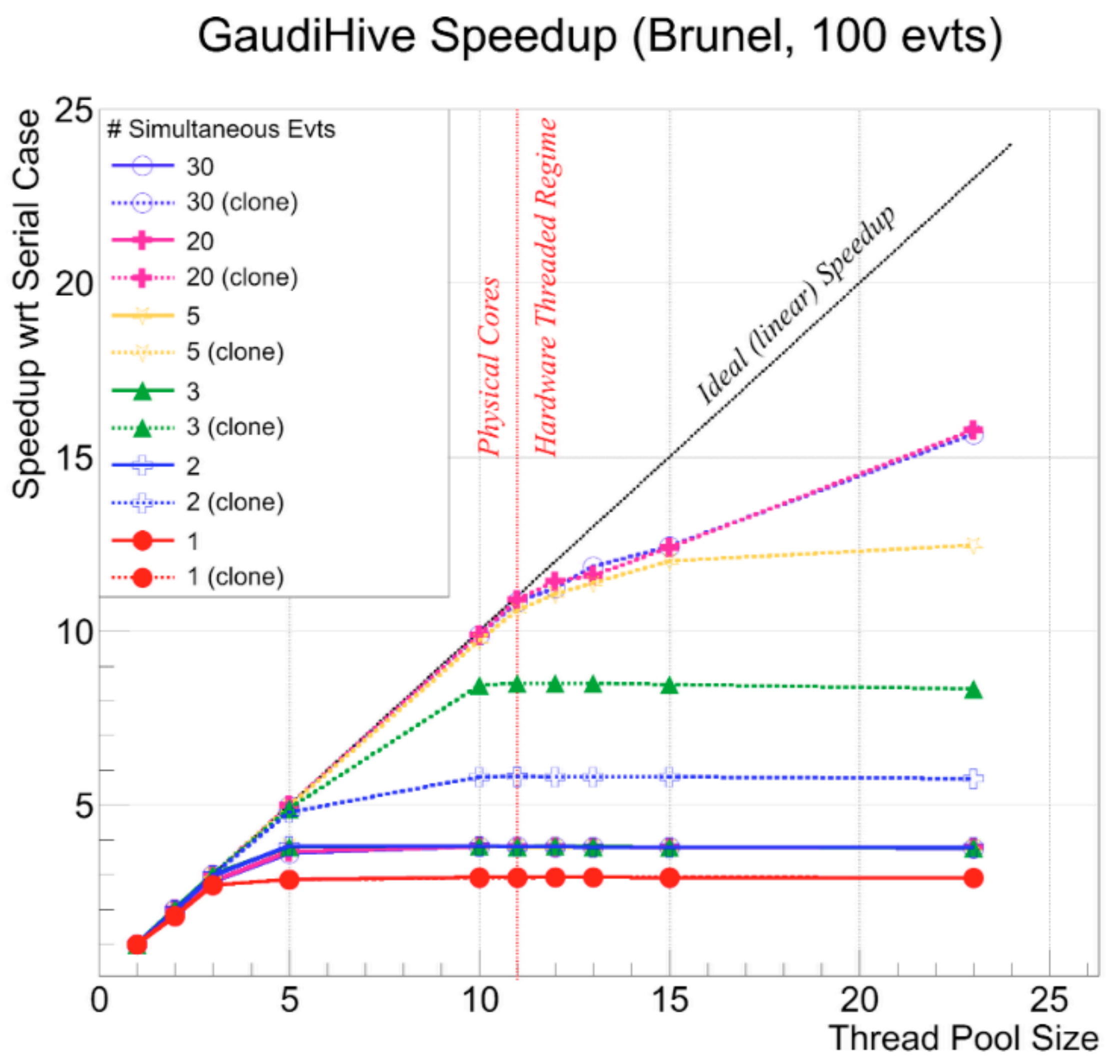
- **Scheduler keeps a state for each algorithm in each event**
  - Simple Finite State Machine
  - Checks for state transitions can be delegated to other classes
  - Allows for rather simple scheduler code

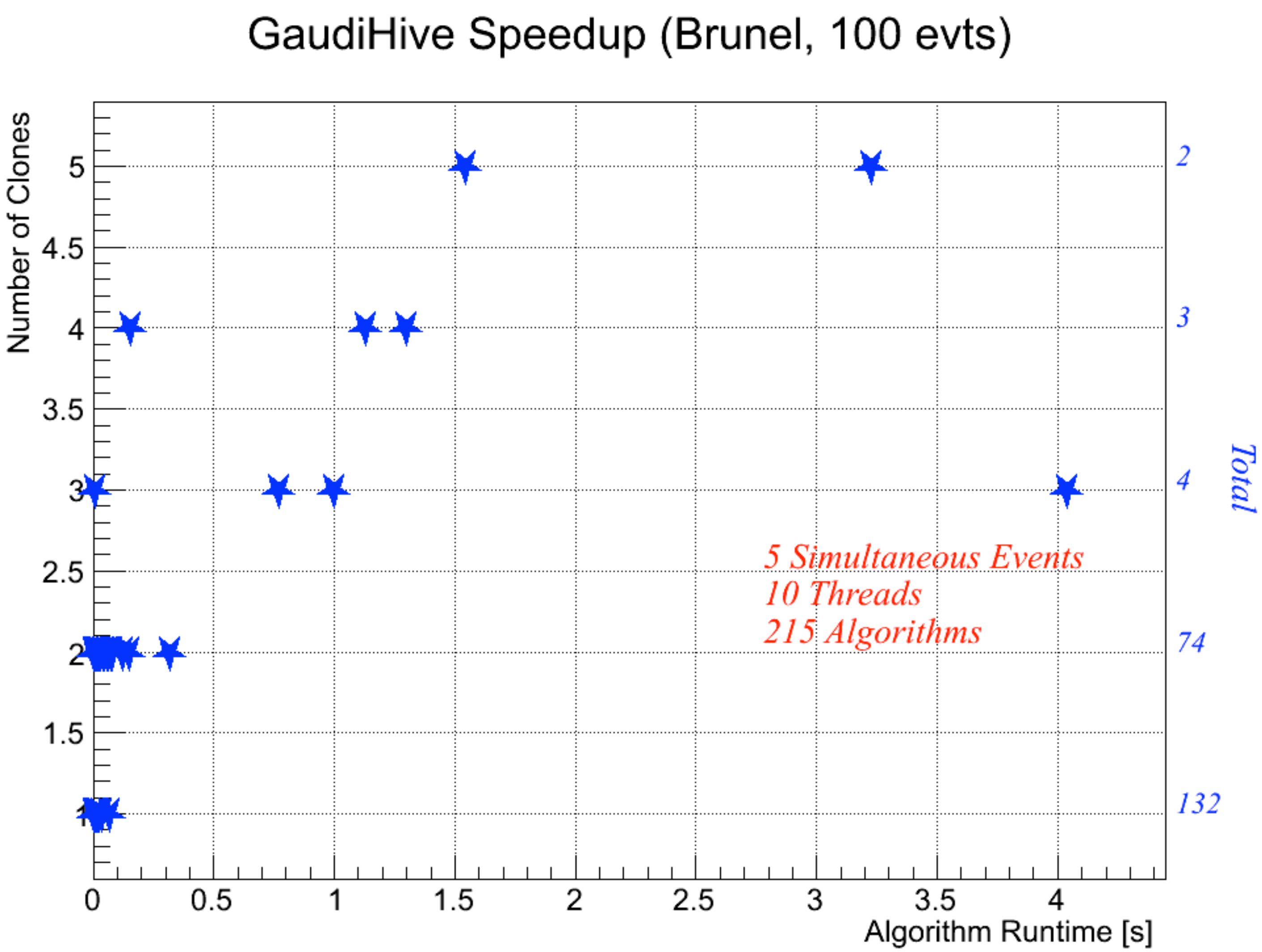


AlgoPool

- Algorithm instances are kept in an **AlgoPool**
- Instances are acquired when creating tbb::tasks and released once task finished
- Number of algorithm instances depend on reentrancy of code:
  1 : non re-entrant;
  n : non re-entrant; use n clones
  -1 : perfectly re-entrant; same instance re-used
- The interface allows more complicated resource checking
  - e.g. two algorithms using the same non re-entrant external library

Scaling Behaviour





see IEEE-NSS 2012 proceedings:
https://concurrency.web.cern.ch/sites/concurrency.web.cern.ch/files/NSS2012-N43-1.pdf

Demands to I/O system

- As in Amdahl's law the slowest serial component limits the maximal achievable speedup
- Slow algorithms can be 'by-passed' by processing more events in parallel
- Serial I/O cannot as it is a shared resource across events
  - application-side resource control/locking to avoid thread-safety issues decrease performance
  - nevertheless multi-event processing has the potential of hiding I/O latencies
- We anticipate the I/O to be a limiting factor rather sooner than later
  - Both for thread-safety and performance

Algorithm requirements

- **Framework orchestrates work using a task-based approach**
- **Other scheduling might negatively interfere with that**
  - Intra-algorithm parallelism has to be limited to using TBB tools
  - No explicit thread handling
  - If chunks of work are big enough -> split algorithm in multiple ones

- **Algorithm interface**
  - Need to know required input; output not strictly needed but useful for sanity checks
  - Stateless algorithms are a nice-to-have but we think that will never happen in real life
    - Algorithm needs to declare its behaviour under cloning
  - Are any external libraries used that are not thread safe?
    - Defining libraries and thus their clients as 'unsafe' could be integrated into the build process

Other components

- **Conditions system**
  - Access to correct conditions for a given event can be handled like event data
    - Request for data is forwarded to the proper conditions slot
  - Problem to solve is how much and which condition data to keep in the cache
    - The actual logic to decide can be hidden from other components easily
- **Statistical and Bookkeeping Data**
  - DQM, Histogram handling, counters are all of the same kind
  - Various approaches possible (locks, thread-safe build-ins, transactional memory)

Conclusions and Outlook

- **The prototype is very encouraging to provide concurrency at all levels**
  - Good scalability potential, although actual implementations are still very primitive
- **We just started scratching the surface and the work in front of us is very large**
  - Concurrency-adaption of services already started, some will need proper re-engineering
  - A lot of room for contributions!
- **Re-usable patterns start to emerge**
  - Opportunity to share knowledge (if not implementations' skeletons) with other prototypes
- **Started effort towards concurrent-development tools**
  - Static code analysis
  - (Semi-)Automated output validation
  - Workflow debugging (not only post-mortem)
- **We are looking forward to see a realistic application running with the newly developed components**