

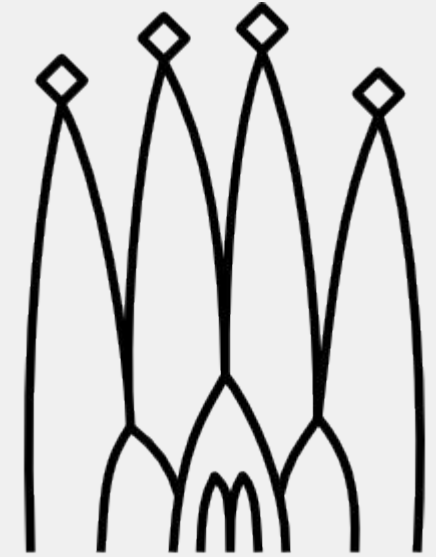
Evolving the Gaudi Framework

Charles Leggett LBNL

Fourth US FCC Annual Meeting

Sept 9 2026

- Gaudi is a "mature" framework: over 28 years old!
 - work started in 1998
 - first publication at CHEP 2000
- The framework has evolved multiple times over the decades
 - C++98 -> C++20
 - ASCII text configuration steering -> python modules
 - single threaded -> multi threaded
 - explicit Algorithm ordering -> data flow
 - functional programming
 - lots of internal reorganization
 - support for multiple experiments
- Where do we go from here?



- Heart of Gaudi event processing is the Scheduler
- Currently have multiple Schedulers for different scenarios
 - Low latency w/ flattened data dependencies, flexible navigation of complex data graphs, etc
 - Designed for synchronous execution of Algorithms - before the era of GPU Algorithms
- Offloading Algorithms to other devices requires asynchronicity
 - waiting for task to complete results in an idle CPU
- Gaudi has some support for asynchronous scheduling
 - currently it's handled within an "AsynchronousAlgorithm" class which are scheduled in a separate pool that uses Boost Fibers to offload to GPUs and yield the CPU
 - a little clunky, but works, assuming we have only a small fraction of async tasks
- Will redesign the Scheduler from the ground up to treat synchronous and asynchronous tasks equally, or really make all tasks asynchronous

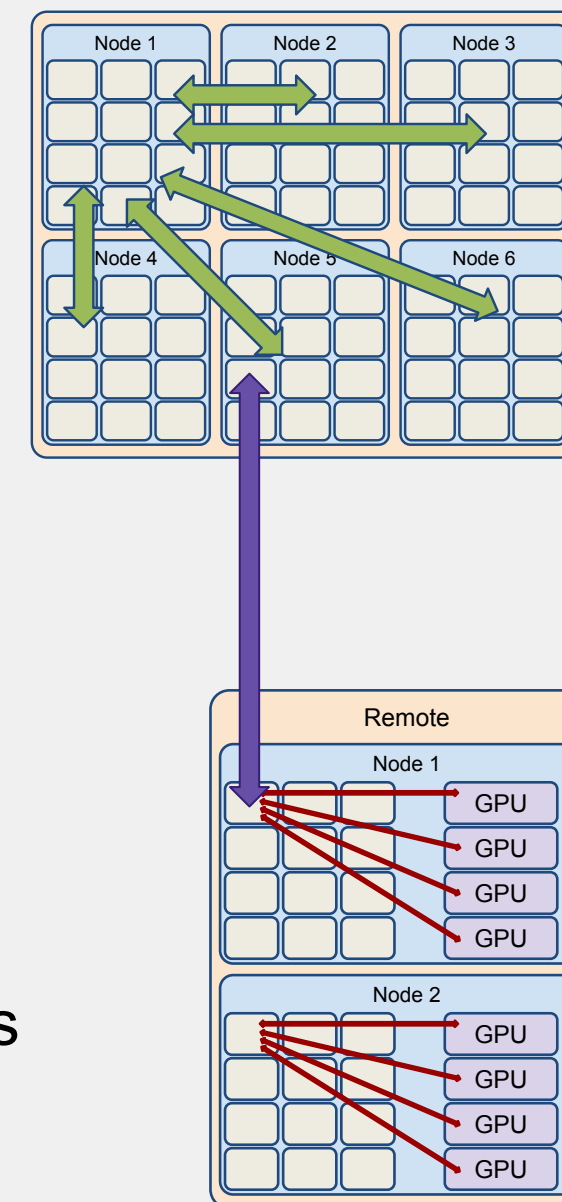
- Gaudi works great on one node - single threaded, multi-threaded, multi-process

- Going beyond a single node

- what if we want to run on a cluster or an HPC?
- what if we want to send tasks to another cluster or facility?
 - remote Inference
 - memory hog: need nodes with lots of RAM
 - remote GPUs
 - flexible cluster design and deployment

- **As-a-Service** model

- makes use of a client/server model
- communications between client and server using various protocols
MPI / gRPC / RDMA / Triton
- server can be nearby or 1000km away



- Job configuration files have already evolved over the decades
 - ASCII text -> python "configurables"
- But steering files are still large, cumbersome, lots of conflicting global settings
- Evolve towards a configuration model that is:
 - more explicit and easier to compose
 - separating configuration construction from execution
 - more Python-native
 - representing configuration as Python data structures
 - easier to validate and serialize
 - fast error detection
 - less dependent on implicit global mutable configuration state
 - reducing surprising side effects from configuration import order or global state
 - easier to use in tests and notebooks
 - suitable for producing reproducible configurations
- Big question: how to handle and migrate from legacy code?

- Original Gaudi was not a Data Flow framework
 - evolution to support multi-threading greatly simplified by using a data flow design
- Legacy Algorithms and Gaudi::Functional
 - ATLAS: turning Athena/Gaudi into a data-flow driven framework was a massive undertaking
 - converting old Algorithms to use DataHandles for legacy code took a long time and was fraught with problems
 - LHCb converted completely to Gaudi::Functional for a pure data flow driven design with no explicit data retrieval
 - reduces boilerplate code and extensive scaffolding
- There is no reason for new experiments to not use the Functional approach
- We need cleaner separation of the Functional scaffolding and the Event Store implementation, and better native support for time-varying data
 - one of the reasons that ATLAS decided against Functional

- Continue to update C++ standard
 - C++20 -> C++23 -> C++26 and beyond
- Use of modern standard-library features
- Safer ownership and lifetime patterns
- Reduced raw pointer and manual resource management
- Improved component/plugin registration
- Easier external consumption of Gaudi libraries
 - and consumption of external libraries
- Asynchronicity: make use of senders/receivers, co-routines, `std::exec` for native C++ asynchronous Algorithms scheduling
- Code cleanup
 - remove deprecated APIs, legacy code, workarounds for old versions of gcc
 - improve maintainability

- Support for and use of Key4Hep is growing
 - CEPC, CLIC, FCC, ILC, SCT, EIC, some Muon Collider studies, LUXE and more
- Integration of Gaudi into the Key4Hep distribution
 - Gaudi is already "in" Key4Hep, but that's not the same as "integrated" into the ecosystem
 - should Gaudi be split up into more modular components?
 - how much of Gaudi belongs in Gaudi, and how much can be moved explicitly into other parts of Key4Hep?
 - does Gaudi need a full-fledged I/O system, or should it just define APIs that users can tie into with pre-existing solutions?
 - better support for EDM4Hep and DD4Hep and other Key4Hep components

- Gaudi may be mature, but it's not geriatric!
- Many enormous changes over the decades
 - more to come
 - (but we're not going to rewrite it in Rust or Go)
- Important to keep up with the times
 - technologies change rapidly - very important to look ahead and anticipate trends
 - changing framework designs is like steering a battleship that's towing an aircraft carrier
- Very important to maintain input from all stakeholders
 - this means you!
- Strive to make Gaudi future proof
 - or at least future tolerant!

Fin