Gaudi components for concurrency:
Concurrency for existing and future experiments

M Clemencic, D Funke, B Hegner, P Mato, D Piparo and I Shapoval
CERN, CH-1211 Geneva 23, Switzerland
E-mail: {marco.clemencic, daniel.funke, benedikt.hegner, pere.mato, danilo.piparo, illya.shapoval}@cern.ch

Abstract.

HEP experiments produce enormous data sets at an ever-growing rate. To cope with the challenge posed by these data sets, experiments' software needs to embrace all capabilities modern CPUs offer. With decreasing memory/core ratio, the one-process-per-core approach of recent years becomes less feasible. Instead, multi-threading with fine-grained parallelism needs to be exploited to benefit from memory sharing among threads.

Gaudi is an experiment-independent data processing framework, used for instance by the ATLAS and LHCb experiments at CERN's Large Hadron Collider. It has originally been designed with only sequential processing in mind. In a recent effort, the framework has been extended to allow for multi-threaded processing. This includes components for concurrent scheduling of several algorithms – either processing the same or multiple events, thread-safe data store access and resource management.

In the sequential case, the relationships between algorithms are encoded implicitly in their pre-determined execution order. For parallel processing, these relationships need to be expressed explicitly, in order for the scheduler to be able to exploit maximum parallelism while respecting dependencies between algorithms. Therefore, means to express and automatically track these dependencies need to be provided by the framework.

In this paper, we present components introduced to express and track dependencies of algorithms to deduce a precedence-constrained directed acyclic graph, which serves as basis for our algorithmically sophisticated scheduling approach for tasks with dynamic priorities. We introduce an incremental migration path for existing experiments towards parallel processing and highlight the benefits of explicit dependencies even in the sequential case, such as sanity checks and sequence optimization by graph analysis.

1. Introduction

Data analysis in HEP is extremely compute intense task. Experiments have build elaborate software packages to reconstruct physical objects from the raw detector readout. These packages feature a wealth of algorithms for hit and track reconstruction, particle flow, analysis of physical objects, etc. A supporting framework is responsible for facilitating the execution of these algorithms, together with providing access to required data – such as detector geometry or magnetic field description – and services, e.g. histogramming and messaging. Conceived in the mid-1990s, these frameworks were designed to sequentially process one event, leveraging the embarrassingly parallel nature of HEP data through multiple jobs.

With higher collision energies, the complexity of the recorded events – and thus the time required for reconstruction – increases super-linearly. At the same time, CPU clock speeds have
reached the power wall, putting an end to the “free-lunch” era [1]. Instead, multi-core processors need to be actively exploited, requiring a redesign of the executed software for parallel processing. Moreover, the memory/core ratio is decreasing – particularly for many-core architectures, thus rendering the one-process-per-core approach unfeasible [2]. This gives rise to the need for more fine grained parallelization. This entails handling several events concurrently within one process – inter-event parallelism – as well as simultaneously executing independent algorithms for one event – intra-event parallelism. The treatment of intra-algorithm parallelism, i.e. concurrent processing of several physical objects, is beyond the scope of this work.

In this paper, we present concepts and components developed to allow inter- and intra-event level parallelism in the Gaudi framework [3]. Gaudi is an experiment independent framework offered by CERN, used for instance by the LHCb and ATLAS collaboration. Section 2 introduces the basic concepts of the framework. An overview of the developed components for concurrency is given in Section 3. Sections 4 and 5 give details on the implementation of data handles for automatic in- and output tracking and the scheduling of concurrent algorithms, respectively. The paper is concluded with a summary and an outlook on future work in Section 6.

2. The Gaudi Framework
Gaudi was originally developed as framework for the LHCb collaboration, but flexibility and adaptability to other experiments were among the primary drivers of all the design decisions involved [3]. The framework defines interfaces for algorithms, tools and services, that are loosely coupled encapsulations of work, communicating via data store(s), refer to Figure 1a. Algorithms constitute the main processing entity, transforming input data objects read from the data store to output objects placed in the store. Furthermore, they produce a binary decision, whether the data processed data is accepted or rejected. Tools encapsulate functionality reusable by many algorithms. A tool instance can either exclusively belong to one algorithm – private tool – or be owned by the tool service and shared among several algorithms – a public tool. Tools may retrieve or store objects in the data store. Services are managed by the framework and provide functionality to all algorithms and tools.

As mentioned above, algorithms are loosely coupled via the data store. They are arranged in sequences to ensure that (i) every algorithm finds its required input in the data store and (ii) only required algorithms are executed for an event. The latter is determined by the binary

1 http://proj-gaudi.web.cern.ch
decision produced by the algorithms, which can lead to the premature termination of a sequence. Complex execution flows can be modeled by composing sequences with and or or behavior, with or without early termination. The configuration of an execution workflow is prescribed by a Python file, which defines the sequences and sets properties of the executed algorithms.

Gaudi was designed to leverage parallelism through multi-process execution. With more CPU cores available but decreasing memory (bandwidth) per core, the need arises for fine-grained inter- and intra-event level parallelism within one process. Thus, the Concurrent Gaudi project was started [5]. The following section gives an overview of the concepts and components introduced for parallel processing.

3. Gaudi Components for Concurrency

Goal of the Concurrent Gaudi project is the support of inter- and intra-event level parallelism. Different concepts need to be introduced to support these two levels of concurrency, refer to Figure 1b for an overview of the components described in the following:

**Inter-event level parallelism** requires a multi-slot data store to hold data products for simultaneously processed events – called “Whiteboard” in Concurrent Gaudi. To provide a minimally intrusive solution, algorithms interact with this data store using the same interface as for sequential processing, with the appropriate slot transparently selected by the framework prior to algorithm execution. Events are read by the event loop manager from the input file and handed over to the scheduler. For inter-event level parallelism, the scheduler can follow the sequential logic with the addition that executing an algorithm of a sequence requires the acquisition of an algorithm instance from the algorithm pool. The pool delivers and collects algorithm instances prior and after execution, respectively. For non-clonable algorithms, i.e. algorithms that use a unique resource, only one instance is present in the pool, thus creating a serialization point. Clonable algorithms have a configurable number of instances and therefore can be executed simultaneously for many events. Intel Threading Building Blocks (TBB) was chosen as multi-threading library for Concurrent Gaudi [6]. Therefore, the scheduler wraps the obtained algorithm instance in a TBB task and submits it to the TBB runtime. [4, 7] give a more detailed account of the presented components.

**Intra-event level parallelism** requires a more sophisticated scheduling strategy than inter-event level parallelism but promises increased speedups, as independent sub-workflows can be executed concurrently. The scheduler therefore needs to be aware of the relationships between algorithms. These manifest themselves in data dependencies and control flow restrictions, due to the sequence composition. In the sequential Gaudi framework, data dependencies were implicitly modeled by sequences. However, for parallel execution they need to be expressed explicitly. Therefore, data handles are introduced to the framework, described in detail in the following section. Section 5 discusses the scheduling strategies used for intra-event level parallelism.

4. Data and Tool Handles

The required inputs of an algorithm must be known to the scheduler after initialization. To facilitate an automatic tracking of read and written data objects, data handles are introduced to the Gaudi framework. These smart pointers contain the name and location of the required/supplied data product, read/write designator and optional flag. Furthermore, for read handles, alternate locations can be supplied. During the processing, one data object can be copied from one location to another, e.g. the raw detector readout might be copied from its original location in the data acquisition sub-tree of the data store to the reconstruction sub-tree, in order to have it available in files only containing the latter tree. With alternate locations,
Listing 1: Algorithm definition using data and tool handles.

```cpp
class MyAlgorithm : public GaudiAlgorithm{

private:
  DataObjectHandle<LHCb::Tracks> m_tracks;
  DataObjectHandle<LHCb::Tracks> m_filteredTracks;
  ToolHandle<ITrackExtrapolator> m_extrapolator;
  ToolHandle<IMaterialLocator> m_materialLocator;

public:
  MyAlgorithm( ... ) : GaudiAlgorithm( ... ) {
    declareInput("Tracks", m_tracks,
                LHCb::TrackLocation::Default);
    declareOutput("FilteredTracks", m_filteredTracks,
                  "Analysis/FilteredTracks");
    declarePrivateTool(m_extrapolator, "TrackLinearExtrapolator");
    declareProperty("TrackExtrapolator", m_extrapolator); // optionally make it a property
    declarePublicTool(m_materialLocator, "DetailedMaterialLocator");
  }

  void execute() {
    LHCb::Tracks * tracks = m_tracks.get();
  }
};
```

Listing 2: Corresponding Python configuration for algorithm defined in Listing 1.

```python
myAlg = MyAlgorithm( 'AnalysisFilter' )
myAlg.Inputs.Tracks.Path = 'Skim/Tracks' # use pre-filtered tracks
myAlg = MyAlgorithm( 'AnalysisFilter' )
myAlg.TrackExtrapolator.Iterations = 1 # rough estimate
```

A data object is first tried be found within the reconstruction sub-tree and – if unsuccessful – then in the data acquisition one, possibly requiring the opening of another file. All properties of a data handle can be customized in the Python configuration file of the workflow, therefore the final data flow can only be determined after the configuration has been processed.

When an algorithm is initialized, i.e. its configuration is processed, the data handle registers itself with the framework. The scheduler – being initialized after all algorithms – queries all algorithms for their in- and output handles and computes the data flow graph. Section 5 elaborates on the use of this graph for scheduling.

Besides algorithms, tools may also retrieve objects from the data store. Thus, an algorithm using a tool inherits its data dependencies. Therefore, we introduced tool handles to the Gaudi framework, to track the usage of tools by algorithms and to properly propagate the in- and outputs of tools to their callers. Accordingly, tools must also use data handles to interact with the data store to benefit from the automatic input/output tracking. Tool handles are smart pointers, containing name and type of the tool as well as a public/private designator. For private tools, properties of the tool can be configured through the parent algorithm in the Python configuration of the workflow. During the initialization of an algorithm, its tools – private and, if required, public – are initialized as well, therefore all input requirements are available at the time the scheduler computes the data flow graph.

Listing 1 exhibits a simple class, making use of the novel data and tool handles. The accompanying Listing 2 illustrates, how properties of data handles and private tools can be
Figure 2: Sequence composition of the MiniBrunel workflow. Sequences are denoted by triangular shapes, the labels correspond to the appropriate C language operator.

Figure 3: Data flow of the MiniBrunel workflow [7]. Algorithms are denoted by rectangular nodes, data objects by ellipses.

...manipulated in the Python configuration of the workflow. The syntax for declaring data and tool handles resembles the syntax for declaring properties of algorithms and tools and is therefore familiar to Gaudi users. To further ease migration, tools retrieved via the tool<T>() convenience method of GaudiCommon, which is widely used to access tools in the users’ code base, also benefit from the automatic data dependencies propagation to the caller. For in/output tracking however, manual labor is required to introduce data handles to the user code.

The following section details the role of the data flow graph for intra-event parallel scheduling.

5. Scheduling
Scheduling of work is a crucial ingredient for multi-threaded applications. Gaudi's scheduling is twofold: (i) selection and order of algorithms to submit as task to the work queue and (ii) scheduling of work queue items on available processor cores. In this section, we detail the former, while the latter is addressed by the TBB runtime [6].

As previously stated, a scheduler for parallel environments requires explicit knowledge of the workflow to be scheduled. The control flow stems from the composition of sequences of algorithms, as explained in Section 2. Figure 2 depicts the control flow for a subset of the LHCb Velo reconstruction, referred to as “MiniBrunel”.\(^2\) As seen in the figure, many sequences

\(^2\) The full LHCb reconstruction is called “Brunel”.

are configured with early-return, thus the algorithms of a sequence are marked as executable one after the other. This allows for the first algorithm to act as a guard – or filter – for the subsequent ones; however, it severely limits the potential for parallel execution.

In addition to being marked as executable by the control-flow, an algorithm can only be scheduled if all its required input data objects are present. The data flow graph for the MiniBrunel sequence is shown in Figure 3. Both flows determine when a particular algorithm is executable, therefore we combine them into one unified execution flow graph, a precedence-constrained directed acyclic graph. Figure 4 shows the unified graph for the Brunel reconstruction sequence, with nodes for algorithms, data objects and decision hubs (stemming from sequences) as well as edges for control and data flow.

The scheduler maintains a list of all executable algorithms. After the execution of an algorithm, its binary decision is propagated along the control flow edges of the graph, updating decision and algorithm nodes accordingly. Additionally, its output data objects are marked as available, with the corresponding data nodes notifying their consumers along the outgoing data edges. If all inputs of an algorithm are available and its control flow requirements are fulfilled it is added to he list of executable algorithms.

As mentioned in the previous section, data handles can be marked as optional. For optional output data handles, the scheduler checks the presence of the object in the data store after the algorithm has finished.\(^3\) For optional input objects, the algorithm execution is deferred as long as the optional object is still producible, i.e. its producing algorithms have not been executed yet, but are still reachable from an undecided decision hub.

The unified graph provides comprehensive insights into the workflow at hand, information that can be exploited for several purposes. The out-degrees of an algorithm’s output data nodes determine its scheduling priority, as a high degree enables many further algorithms to run. If an algorithm is no longer required to be executed due to the control flow, its priority becomes zero. The critical path – i.e. the longest dependency chain – can be identified in the graph and extra priority given to the algorithms belonging to it, thus reducing the makespan. The graph also reveals the maximum concurrency in the workflow, i.e. the maximum number of algorithms executable at any given time. This bounds the number of occupiable threads, thus aiding in the resource planning in computing centers. However, pure graph analysis can not reveal the

\(^3\) The check is omitted for non-optional output handles for performance reasons.

\(^4\) brunel2012magdown workflow
obtainable speedup for a workflow, as the speedup depends on the relative time spent in each algorithm. Therefore, the number of threads that should sensibly be allocated to a process, can only be determined using runtime information. Further static analysis can be applied to identify unfulfillable data dependencies and superfluous control flow constructs.

6. Summary
The concurrent Gaudi project provides components and concepts to the experiments to introduce inter- and intra-event level parallelism in their software workflows. As non-intrusiveness is one of the primary goals of the project, all introduced components present an option to the user, while leaving their sequential production code intact. This allows for an incremental revision of the existing implementation and adoption of the components offered. In order to process data concurrently the user must (i) explicitly declare data in- and outputs for each algorithm, (ii) declare which tools are used by an algorithm, and (iii) revise thread-unsafe code, such as use of caches, back-channel communication, race conditions in the update of shared data, . . .

The first two points can be achieved by using the newly introduced data and tool handles, respectively. To ease their use, the interfaces of the handles were designed according to components familiar to Gaudi developers. The third point requires a case by case analysis of the operations performed by the algorithms.

Having explicitly stated the algorithms’ in- and outputs enables the automated deduction of the data flow among them. This allows for the computation of a unified execution flow graph, combining information from the algorithms’ data dependencies and the control flow between them, stemming from their sequence composition. The graph is used to identify algorithms with both their data and control flow prerequisites fulfilled. These are submitted for execution by the scheduler, ordered by their priorities. The priority of an algorithm is determined by the connectedness of its output data objects in the execution flow graph, thus enabling maximal parallelism of the execution.

The graph structure also serves a secondary purpose, allowing the analysis of a given configuration statically for unfulfillable data dependencies and superfluous control flow constructs. Experiments can benefit from this analysis capabilities even without revising all their code for parallel execution by just introducing data and tool handles.

The gradual migration strategy and the benefits of static configuration analysis present a strong argument for the adoption of the components and concepts developed in the concurrent Gaudi project. Consequently, the LHCb collaboration has chosen to adapt these features into their production framework.

References
[1] Sutter H 2005 Dr. Dobb’s Journal 30 202–210
[2] Borkar S 2007 Proceedings of the 44th Annual Design Automation Conference (New York, NY, USA: ACM) pp 746–749
[3] Barrand G et al. 2001 Computer Physics Communications 140 45 – 55
[4] Clemencic M, Hegner B, Mato P and Piparo D 2014 Journal of Physics: Conference Series 513 022013
[5] Hegner B, Mato P and Piparo D 2012 2012 IEEE Nuclear Science Symposium and Medical Imaging Conference (NSS/MIC) pp 2003–2007
[6] Willhalm T and Popovici N 2008 Proceedings of the 1st International Workshop on Multicore Software Engineering (New York, NY, USA: ACM) pp 3–4
[7] Clemencic M, Hegner B, Mato P and Piparo D 2014 Journal of Physics: Conference Series 513 052028