JOINT INSTITUTE FOR NUCLEAR RESEARCH
Meshcheryakov Laboratory of Information Technologies

FINAL REPORT ON THE START PROGRAMME

Gaudi Hive for SPD

Supervisor:
Dr. Danila Oleynik

Student:
Alexey Yastrebov, Belarus
Sukhoi State Technical
University of Gomel

Participation period:
July 08 – August 16,
Summer Session 2024

Dubna, 2024

TABLE OF CONTENTS

ABSTRACT

Like any large experiment, SDP involves the development of its own applied software for data processing and analysis. To settle common practices and methodology of development a special framework should be used. One such framework is Gaudi. It is a framework software package that is used to build data processing applications for High-Energy Physics experiments. Gaudi contains all of the components and interfaces which allows build event data processing applications for your experiment. Of particular interest is Gaudi Hive – extension to Gaudi with multithreading support.

This work describes about:

– Gaudi and Gaudi Hive architecture and their main components;

– Gaudi Hive features like parallel event processing and algorithms executing, automatic composing correct sequence of algorithms execution;

– writing Job Options via Python script;

– building algorithms pipelines and branches in Job Options Python script.

INTRODUCTION

The Spin Physics Detector, a universal facility for studying the nucleon spin structure and other spin-related phenomena with polarized proton and deuteron beams, is proposed to be placed in one of the two interaction points of the NICA collider that is under construction at the Joint Institute for Nuclear Research (Dubna, Russia) [1]. At the heart of the project is extensive experience with polarized beams at JINR. The main objective of the proposed experiment is the comprehensive study of the unpolarized and polarized gluon content of the nucleon. Spin measurements at the Spin Physics Detector at the NICA collider will make a unique contribution and will challenge our understanding of the spin structure of the nucleon.

Due to the difficulties encountered in constructing a hardware trigger, a triggerless data acquisition system is assumed for SPD [2]. Together with the high collision frequency (up to 12 MHz) and hundreds of thousands of detector channels, this presents a challenge to design an efficient data acquisition and processing system.

There are two systems for data acquisition and processing are developing today for SPD: online and offline filter. The online filter will be a high-throughput system which will include heterogeneous computing platforms similar to many high performance computing clusters. The offline software is designed to solve such tasks as reconstruction of events, their modelling, as well as physical analysis of the data obtained as a result of experimentation.

To take advantage of modern computing hardware, the offline software for the processing of experimental data should be capable of taking advantage of concurrent programming techniques, such as vectorization and thread-based programming. This is the reason why Gaudi framework is chosen for SPD. Of particular interest is Gaudi Hive – extension to Gaudi with multithreading support.

ABOUT GAUDI

Gaudi is a software package containing all the necessary interfaces and components for writing frameworks for experiments in high-energy physics [3]. Initially Gaudi was developed for the internal needs of the LHCb collaboration, but soon after the ATLAS collaboration joined the development it became clear that the package can be easily transformed for any other experiment. The robustness of the package is confirmed by its use in numerous collaborations around the world.

Gaudi makes clear distinction between data and procedures. This isolation is achieved by building an architecture, which is a set of components and rules of their interaction. Each component has its own interface and functionality. The user's task is reduced to defining the functionality of a particular component while preserving its interface. Programmatically, this is done by inheriting from one of the base classes. The architecture of the framework is shown on figure 1.

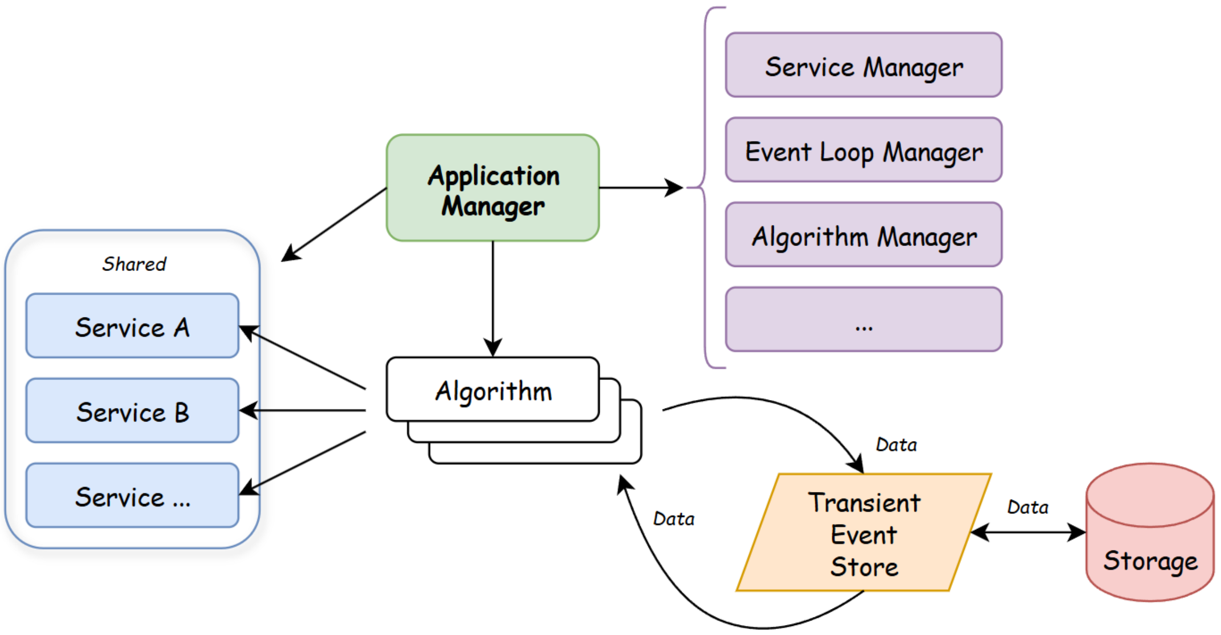

Figure 1 – Gaudi architecture

The central component in this framework is the *Algorithm*, which transforms raw event data into processed data e.g. from detector digitizations to hits, from hits to clusters or tracks, from tracks to jets, etc. The implementation of these *Algorithms* encapsulates the knowledge physicists have of detector and physics performance, and represents the real substance of these data processing applications. The software integrators then combine a fairly large number of these *Algorithms*, together with other components that provide core functionality, in order to assemble and configure a complete application.

Algorithms are initialized at the beginning of the job and run sequentially in a predefined order for each event in the main event loop, and finalized at the end to output any statistical quantities. The sequencing of algorithms such that some can run in parallel is central to enabling the concurrent execution of code when processing a single event.

The way an *Algorithm* interacts with the framework is kept very simple. It interacts solely with a special piece of code called the *Transient Event Data Store* in order to retrieve its input data and eventually also to store the results it produces, called the data products. The execution of each *Algorithm* is completely independent of those other algorithms (the producers) that provide its input data, as well as those algorithms (the consumers) that make use of its results.

The main disadvantage of Gaudi is the lack of multithreading. To take advantage of modern computing hardware, the offline software for the processing of experimental data should be capable of taking advantage of concurrent programming techniques, such as vectorization and thread-based programming. This is the reason why another framework based on Gaudi was developed – Gaudi Hive.

GAUDI HIVE

Gaudi Hive follows the concept of task parallelism [4]. Here a task executes a given *Algorithm* on a given event. The dependencies of these tasks on each other can be expressed in terms of a directed acyclic graph formed from the input-output relation of the *Algorithms* (figure 2).

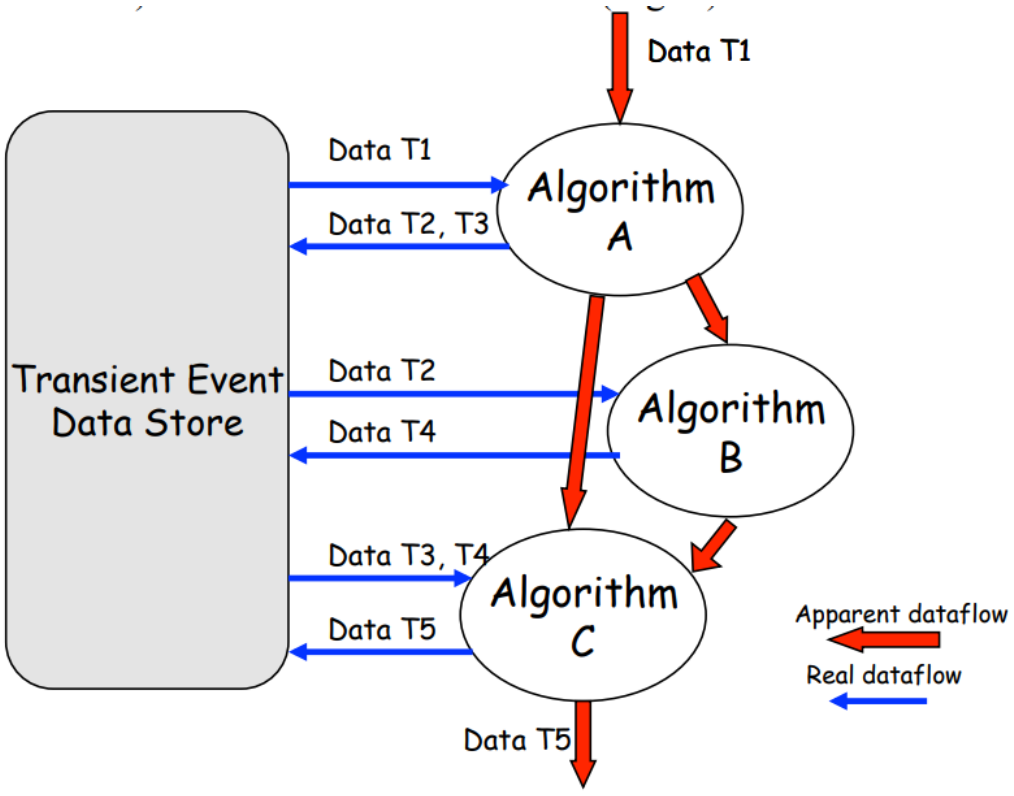

Figure 2 – The dataflow between *Algorithms*

Gaudi Hive is based on driving the execution according to the availability of data. In practice, this is achieved as follows. The central elements are depicted on figure 3 and include a special parallel *Scheduler* and the *Whiteboard* as thread-safe event store. All *Algorithms* are required to declare their required input data as part of the initialization step.

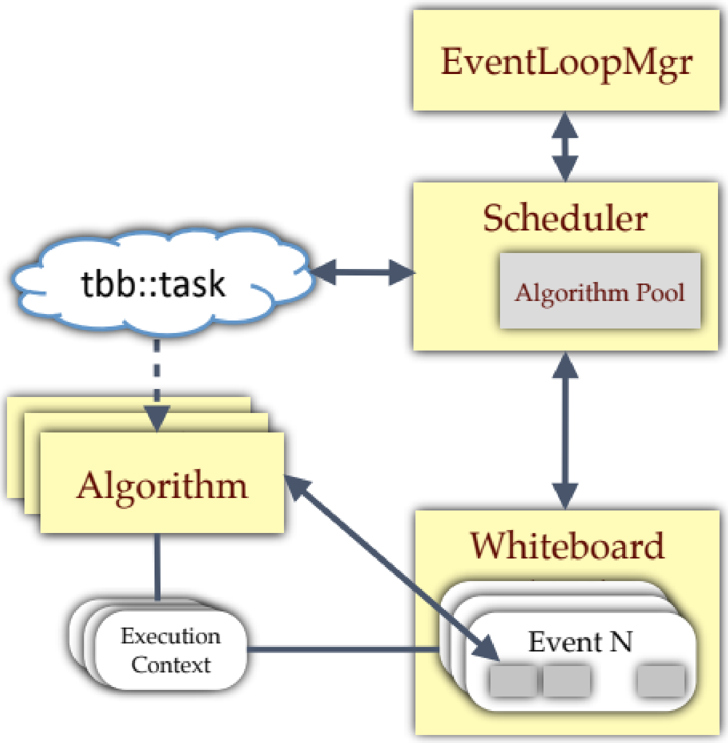

Figure 3 – Gaudi Hive architecture

The *Whiteboard* is a multi-event store, which can contain multiple *Event Stores*, implements the original event store's interface in a thread safe manner [5].

Algorithms are usually not thread-safe and so a complex *Algorithm*, such as found in track reconstruction, cannot be applied in two events at the same time and therefore it requires that the CPU has exclusive access to internal states of the track *Algorithm*. In Gaudi Hive, the management of exclusive *Algorithm* instances is done via a (thread-safe) *AlgorithmPool*. To reduce the blocking due to busy algorithms, the presented prototype allows the cloning of *Algorithms*, so that multiple instances of the same *Algorithm* are available in the *AlgorithmPool*.

As soon as new data become available in the *Whiteboard*, the *Scheduler* checks to see whether there are *Algorithms* whose input data dependencies are fulfilled. Concrete *Algorithm* instances are then requested from an *AlgorithmPool*. As soon as execution is finished, the instance is released again to the *AlgorithmPool*.

So, Gaudi Hive allows to parallelize event processing, as well as run several independent algorithms in parallel within the processing of a single event [6].

Figure 4 shows the scheme of parallel processing of three events using a single *Algorithm*. In this case, a clone of the *Algorithm* and *Event Store* is created for each event.

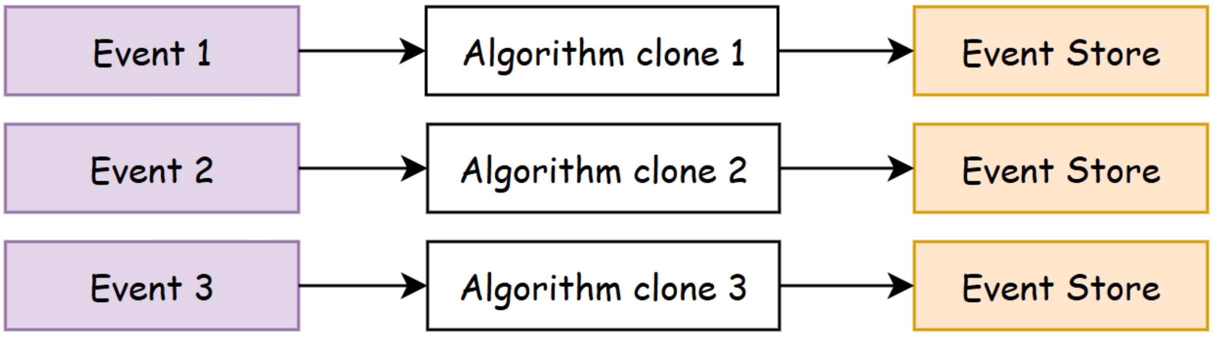

Figure 4 – Scheme of parallel processing of three events using a single *Algorithm* (clones)

Figure 5 shows an example of parallel execution of *Algorithms* with the processing of a single event.

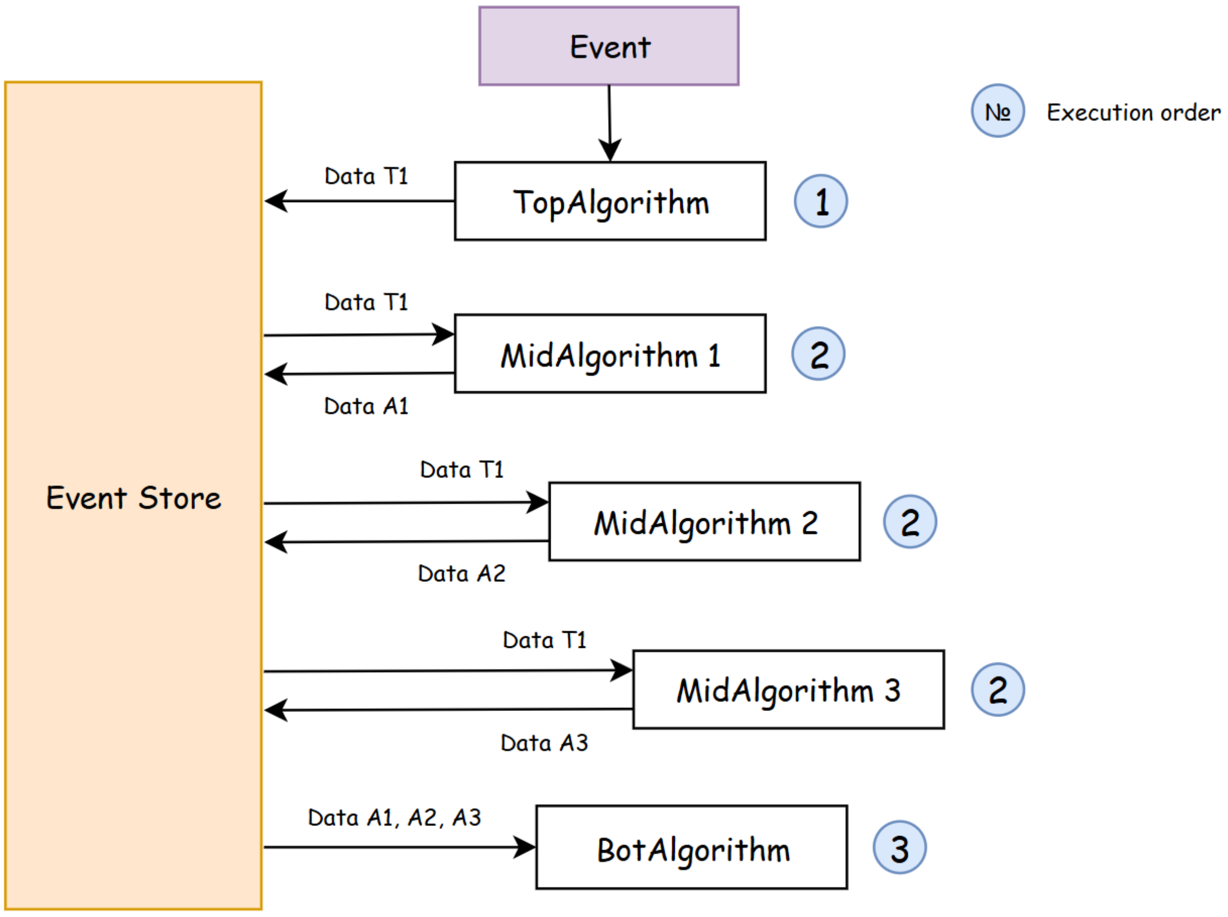

Figure 5 – Example of parallel *Algorithms* execution

The *Algorithms* are executed in the following order:

– *TopAlgorithm* is runs first, do something and put *DataObject T1* into the *Event Store*;

– three different *MidAlgorithm* whose inputs is *DataObject T1* are runs in parallel and produce *DataObject A1*, *A2* and *A3*;

– *BotAlgorithm* whose input is *DataObject A1*, *A2* and *A3* runs after completion of all *MidAlgorithms*.

JOB OPTIONS PYTHON SCRIPT FOR USING GAUDI HIVE

Job Options files are used to run Gaudi. The term Job refers to running a programme in a certain configuration on certain input data. In order to configure a Job, Gaudi provides a mechanism for Job Options files, which are a set of commands interpreted by Gaudi. The language of these commands describes the sequence of algorithms, their parameters, the services used, the input data, and more. However, most modern experiments use a different approach. It involves configuring tasks using Python scripts.

In Python script first of all, we need to import all needed python classes: *Algorithms* and *Services* from *Configurabels* module and Gaudi Hive classes. Gaudi Hive's main classes are as follows:

– *HiveWhiteBoard* – event data service;

– *HiveSlimEventLoopMgr* – event loop manager;

– *AvalancheSchedulerSvc* – scheduler service;

– *AlgResourcePool* – used for enable algorithm cloning.

Also we need to import *ApplicationMgr* class from *Gaudi.Configuration* module, that is responsible for launching Gaudi app.

During import, *Gaudi* scan special *.confdb* and *.confdb2* files, which contains names of Python modules and files. These modules and files are creating during building project with CMake. Python files contains classes of algorithms and services with their properties.

On the next step we can create a global variables with number of *Event Stores*, events and thread pool size. For example, in case according to the scheme on figure 4, we need to run processing of three events in parallel. Each event will be processed in a separate thread (i.e. the thread pool size is equal to three), and a separate *Event Store* for each event will be used (i.e. the number of *Event Stores* is also equal to three).

Next step is creating instances of *HiveWhiteBoard*, *HiveSlimEventLoopMgr* and *AvalancheSchedulerSvc* classes and pass configuration via their constructor or properties:

– for *HiveWhiteBoard*: proprety *EventSlots* – a number of *Event Stores* for each event processing;

– for *AvalancheSchedulerSvc*: property *ThreadPoolSize* – number of threads in thread pool.

In the same way we need to create instances of *Algorithm*. Since the events will be processed in parallel, it's necessary to specify the number of algorithm clones that will be created by the Gaudi to process each event. The number of clones is specified in the property named *Cardinality* [7].

Also, we need to «say» Gaudi that it can clone algorithms. This can be done by creating an instance of *AlgResourcePool* class and specifying property *OverrideUnClonable* with *True* value.

Finaly, we need to create instance of *ApplicationMgr* class and pass next properties:

– *EvtMax* – number of events to processing;

– *EvtSel* – event selection;

– *ExtSvc* – list with external services;

– *EventLoop* – event loop manager (to use Hive it should be instance of *HiveEventLoopMgr*);

– *TopAlg* – list of top level algorithms;

– *MessageSvcType* – message service for streaming info, warning, debug and error messages to the standard output stream (for Hive it’s the *InertMessageSvc*).

Gaudi app will start automaticly during creating instance of *ApplicationMgr*.

To run Gaudi app with Job Options Python script, the *gaudirun.py* script is used with the Job Options script passed as its parameter.

ALGORITHMS PIPELINES AND BRANCHES IN JOB OPTIONS PYTHON SCRIPT

There is a *GaudiSequencer* class designed to manage the execution order of *Algorithms*. It can encapsulate a series of *Algorithms* and control their execution as a single unit. Its ensures that *Algorithms* are executed in a specific order.

In *GaudiSequencer* can specify how the algorithms should be executed: sequentially or in parallel. The member of *GaudiSequencer* can be not only *Algorithm* but also *GaudiSequencer*, because it’s inherit *Algorithm* class. In this way, it is possible to create an algorithms pipeline. Figure 6 shows example of algorithms pipeline with parallel *GaudiSequencer* in sequential. *GaudiSequencer A* is in sequential mode, *GaudiSequencer B* is in parallel mode.

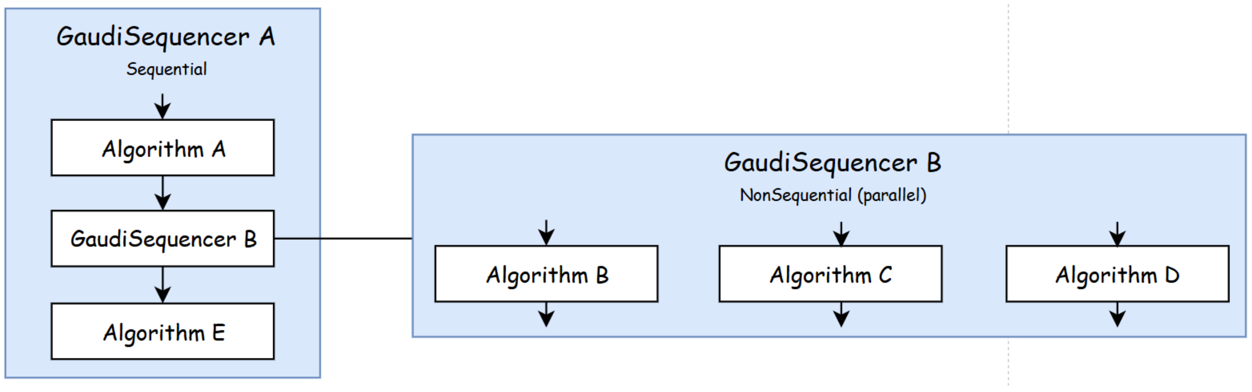

Figure 6 – Example of algorithms pipeline using *GaudiSequencer*

In Job Options Python script *GaudiSequencer* class is located in the *Configurables* module.

Also, *GaudiSequencer* class has a *ModeOR* property boolean type for creating algorithm's branches. In *OR* mode execution of sequence is terminated as soon as one of the *Algorithms* is completed. In *AND* mode execution of sequence is terminated after all *Algorithms* are completed. Schematically, it's shows on figure 7.

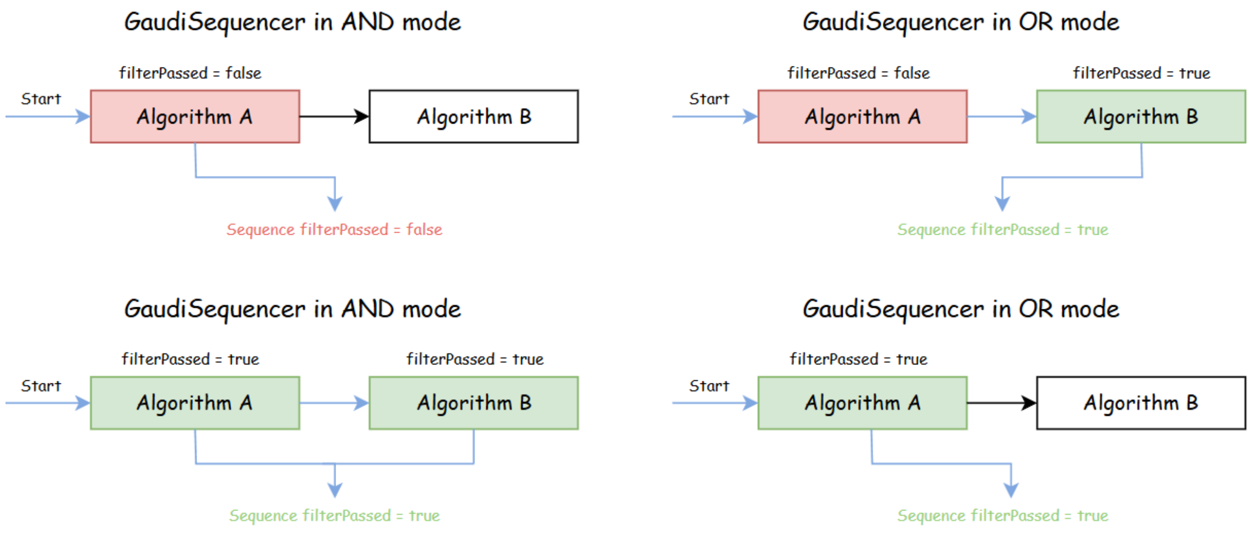

Figure 7 – *Algorithms* execution using *GaudiSequencer* in OR and AND mode

FilterPassed is *Algorithm* property, which means is algorithm task was completed successfully or not (no critical error has occurred, just task of the algorithm is not completed and need to execute another algorithm's branch for such situation).

SETTING INPUTS AND OUTPUTS OF GAUDI ALGORITHM

Gaudi Hive can automatically compose the correct sequence of *Algorithms* execution using information about *Algorithm's* inputs and outputs. To set inputs and outputs we need to declare *DataObject* handlers, which provide opportunity to read and write data to *Event Store*.

In algorithm include file:

– create property with vector of strings for inputs and outputs paths;

– create map with Event store path as key and unique pointer for *DataObjectHandle* as value.

In algorithm cpp file in initialize method we need to create instances of the *DataObjectHandle* class for each input and output. This class contains methods for interaction with *Event store*. As constructor parameters we need to pass a path to *DataObject* in the *Event store*, mode (*Reader* or *Writer*) and link to the owner class instance. Mode is enumeration in Gaudi *DataHandle* class. Mode *Reader* means that *Algorithm* read data from the *Event store* (i.e. it's *Algorithm's* input), and mode *Writer* means that algorithm write data into the *Event store* (i.e. it's *Algorithm's* output).

To put and get *DataObject* from the *Event store* it's used *put* and *get* method accordingly from *DataObjectHandle* class.

CONCLUSION

As a result of this work, the architecture, components and main features of Gaudi framework and its extension – Gaudi Hive – were studied. Also learnt how to write Job Options and run Gaudi via Python script.

Gaudi Hive allows to process several events in parallel and execute algorithms in parallel within one event processing, which allows to increase the speed of data processing on multi-core CPUs. At the same time, it is not necessary to take care that algorithms were thread-safe, because a clone of the algorithm is automatically created for each thread.

Gaudi also provides a possibility to create algorithms pipelines and branches. And Gaudi Hive implements a mechanism of automatic composing the correct sequence of algorithms execution using information about algorithm's inputs and outputs.

Configuring algorithms and services via Python script allows to automate the setup and startup of Gaudi, which is extremely useful in distributed computing systems.

REFERENCES

1. Conceptual design of the Spin Physics Detector / The SPD collaboration // JINR, 2022. – 162 p.

2. Technical Design Report of the Spin Physics Detector at NICA / The SPD collaboration // JINR, 2024. – 349 p.

3. Gaudi Project documentation, https://gaudi-framework.readthedocs.io

4. Hegner, B. Evolving LHC Data Processing Frameworks for Efficient Exploitation of New CPU Architectures / B. Hegner, P. Mato, D. Piparo // CERN, 2012.

5. Clemencic, M. Introducing concurrency in the Gaudi data processing framework / M. Clemencic // Journal of Physics: Conference Series, 2014.

6. Gaudi/Athena Multithreaded Scheduling, https://indico.cern.ch /event/931842/contributions/3916050/attachments/2067127/3469328/AthenaGaudiScheduling.pdf.

7. Leggett, C. AthenaMT: upgrading the ATLAS software framework for the many-core world with multithreading / C. Leggett // Journal of Physics: Conference Series, 2017.

ACKNOWLEDGMENTS

The author expresses gratitude to Dr. Danila Oleynik for providing comprehensive guidance, technical support and valuable recommendations. I am also grateful to the organizers of START for the opportunity to complete the program.