SNiPER: an offline software framework for non-collider physics experiments

J. H. Zou[1], X. T. Huang[2], W. D. Li[1], T. Lin[1], T. Li[2], K. Zhang[1], Z. Y. Deng[1], G. F. Cao[1]

[1] Institute of High Energy Physics, Chinese Academy of Sciences, Beijing, China
[2] Shandong University, Jinan, China

E-mail: `zoujh@ihep.ac.cn`

Abstract. SNiPER (Software for Non-collider Physics ExpeRiments) has been developed based on common requirements from both nuclear reactor neutrino and cosmic ray experiments. The design and implementation of SNiPER is described in this proceeding. Compared to the existing offline software frameworks in the high energy physics domain, the design of SNiPER is more focused on execution efficiency and flexibility. SNiPER has an open structure. User applications are executed as plug-ins based on it. The framework contains a compact kernel for software components management, event execution control, job configuration, common services, etc. Some specific features are attractive to non-collider physics experiments.

1. Introduction

For most modern high energy physics experiments, the offline software plays an important role in improving physics analysis quality and efficiency. A unified software platform is necessary to provide a common working environment. So that physicists are able to share ideas and results conveniently following some conventions, without suffering from technical programming details. This has advantages for resource optimization and manpower integration, which can improve the software development, usage and maintenance.

Generally the data processing procedure obeys a few fixed patterns in a specific domain. These patterns can be implemented as software framework independently. A software framework is reusable and extensible. It is the skeleton of a software platform. In such a framework, programming experts have taken account of various requirements for data processing. Many generic functionalities and attractive features are integrated. Users can selectively replace a module or add new modules to it. A distinctive framework determines the vitality of the whole software platform. There have already been several very successful and widely used frameworks, such as Gaudi [1] and basf2 [2].

However, there are always new challenges to software in recent experiments. A new framework SNiPER (Software for Non-collider Physics ExpeRiments) is implemented based on the requirements of nuclear reactor neutrino and cosmic ray experiments, especially JUNO (Jiangmen Underground Neutrino Observatory) [3] and LHAASO (Large High Altitude Air Shower Observatory) [4] in China. As a general purpose framework, it is customizable, extensible and maintainable. We try to keep it concise and lightweight. Uncertain requirements in the far future are not concerned.

SNiPER is designed and managed modularly. Every functional element is implemented as a module, and can be dynamically loaded and configured. Modules are designed as high cohesion units with low couplings between each other. They communicate only through interfaces. We can replace or modify one module without affecting any of the others.

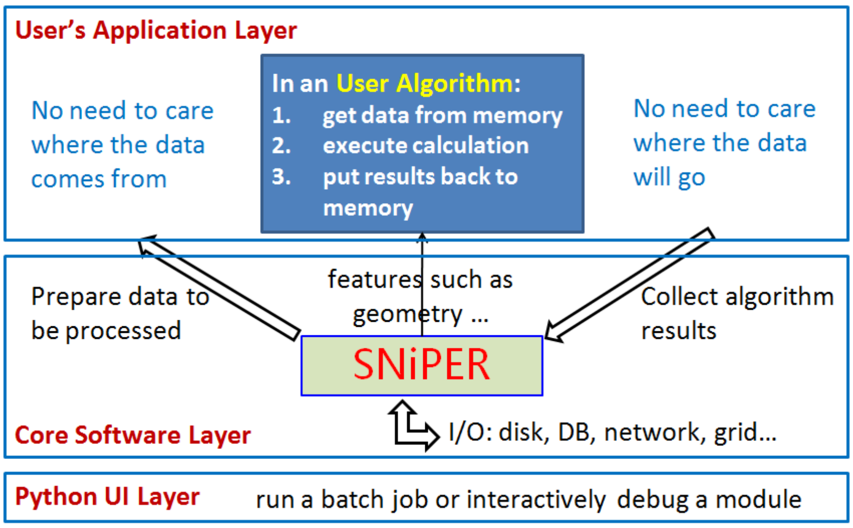
User's Application Layer
No need to care where the data comes from
In an User Algorithm:
1. get data from memory
2. execute calculation
3. put results back to memory
No need to care where the data will go
Prepare data to be processed
features such as geometry ...
Collect algorithm results
SNiPER
Core Software Layer
I/O: disk, DB, network, grid...
Python UI Layer
run a batch job or interactively debug a module

Figure 1. An overview of SNiPER

As shown in Fig.1, a SNiPER based system can be divided into 3 layers. In the core software layer, a compact SNiPER kernel provides many common interfaces, such as data I/O and memory management. All features in the kernel can be reused directly by any components in the system. In the application layer, algorithm developers are free from trivial missions, such as data preparing and result collecting. In the User Interface layer, the dynamic scripting language Python is used to provide additional flexibility. A SNiPER job can be executed in batch mode with script files or interactively in command line.

2. Implementation

Mixed programming with multiple languages is attractive and practical. It is possible to choose different language for different aspects of our software. C++ is chosen for the main body of SNiPER. It determines the application execution efficiency. Meanwhile Python is used as user interface, which provides more runtime flexibilities.

Boost.Python [5] is the key of C++ and Python integration in SNiPER. We dealed with Boost.Python carefully in the core functionalities. Thus most users needn't to know the detailed skills of mixed programming.

2.1. Software Components Management

SNiPER is easy to be extended. New features can be implemented as SNiPER modules and embedded into the framework as plug-ins. The computing model is inspired by other pioneering software frameworks, especially the concept of algorithm and service from Gaudi. An algorithm provides a specific procedure for data processing. A complete data processing chain is composed of a sequence of algorithms. A service provides useful features that can be called by users anywhere when necessary.

There is a new concept that named task in SNiPER. Task performs like a lightweight application manager. But there can be more than one task instances in a single running job.

It's able to specify different algorithms and services in each task independently. We can achieve some complex goals in a simple way by this mechanism. For example, when there is only single-stream I/O service, we can hold a single-stream in each task and simply combine several tasks to implement a multi-stream I/O application.

Since a task acts like a components container, each task is embodied by some services, algorithms and sub-tasks. It is recommended to organize all task instances in a tree structure in a SNiPER job, as show in Fig.2. Each component (a task, an algorithm or a service instance) is assigned a unique path style string. Component instances can be created with the same name in different paths, and each one can be retrieved with its absolute or relative path. We can avoid the naming confusing or conflicts by this way compared to a plan namespace. This approach also provides us a clear architecture for components management. It groups related components together in a single task, and organizes tasks via their affiliation.

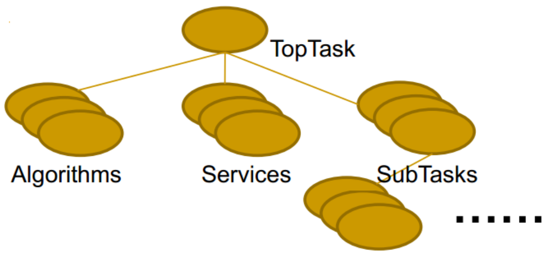

Figure 2. Components tree in a SNiPER job

2.2. Event Execution Control

As mentioned above, algorithms and services are plugged and executed dynamically. They can be selected and combined flexibly for different requirements. Algorithms in a single task are executed sequentially. Obviously this is not enough. In SNiPER we implement an incident mechanism to enhance the communication between tasks.

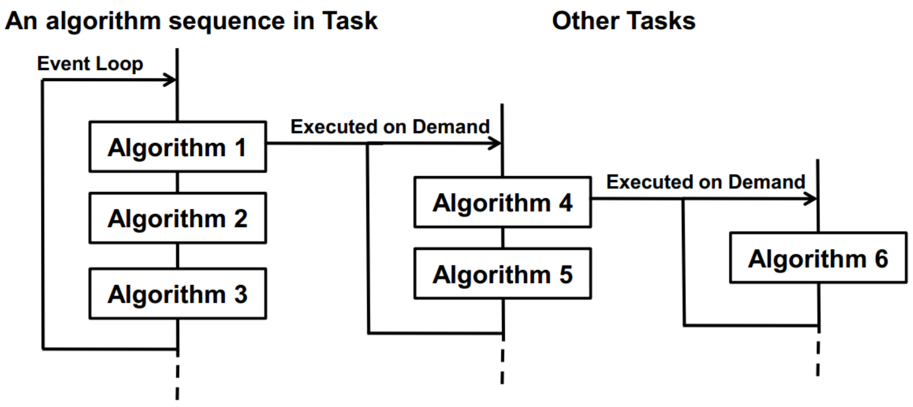

Figure 3. Conditional execution of algorithm subsets

As show in Fig.3, a task can be triggered by an incident on demand. So that users can selectively excute algorithm subsets during event loop. In this way, SNiPER gains more flexibility to fit non-collider physics experiments.

Take the simulation of an IBD (Inverse Beta Decay) event as example,

$$\bar{\nu}_e + p \rightarrow e^+ + n$$

A neutrino event results two signal events (a positron and a neutron) in the detector. In this situation, we can handle the signal events in a sub-task. It can be triggered twice for each neutrino event. In other words, algorithm subsets are able to execute different times during event loop.

For the coupling between tasks is very weak, tasks can be combined almost with no limitation. We are able to construct more complicated applications with simple tasks.

2.3. Data Memory Management

Compared to collider experiments, neutrino experiments have two specific characteristics: 1) time correlation between events; 2) a very small fraction of signal events among a large number of backgrounds. The software framework should therefore provide a mechanism of flexible data I/O and event buffering to enable high efficiency data access and storage, as well as the capability to retrieve the events within a user-defined time window.

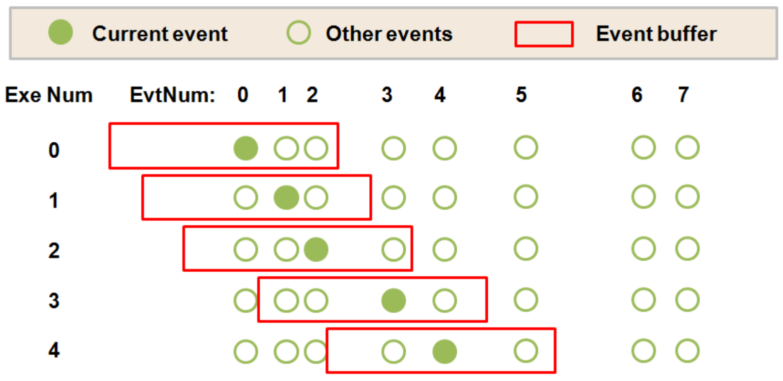

Figure 4. FIFO Event buffer with a time window

In order to facilitate event correlation analysis, a FIFO (First Input First Output) data buffer is used to store adjacent events and a sophisticated method of memory updating is designed, as show in Fig.4. In each execution moment, an event acts as the anchor of time window in the event flow. Adjacent events in the time window are cached in the buffer simultaneously. During the event loops, the anchor event moves forward one by one, and events in the time window are synchronized at the same time. So that access to successive events within a user-defined time window according to event timestamps becomes possible.

This FIFO event buffer is optional in SNiPER. Users are able to create a new memory management service by implementing the related interface of SNiPER. Both event data model and in-memory data management can be customized by different applications.

2.4. Common Services

The framework involves many frequently used functionalities. They are implemented as service modules, which can be selectively loaded.

Data to be processed may come from different places and be in different types and formats. The results may also be stored in different ways. In the framework we reserve interfaces for different I/O support, so that the I/O service can communicate conveniently with other modules via the interface. A simple logging mechanism is implemented to support formatted logs, with the level of priorities which can be configured at runtime. A system resource loading tool is developed for the performance monitoring. It's helpful for us to investigate performance bottlenecks, and potential bugs such as memory leak. Accessing to popular external libraries is also considered. For example, ROOT [6] histograms booking is wrapped in a service, so that it is configurable via SNiPER Python interface.

More common features, such as particle property lookup and database accessing, will be available in the future.

3. Conclusions

Currently a nascent product of SNiPER has been released and is being used by the JUNO and LHAASO experiments. The practices show that the software architecture is universal and expandable. As a general purpose framework, SNiPER can be used by other non-collider physics experiments to build their offline data analysis and processing systems. Now we are considering the parallel computing in SNiPER. This fantastic feature will be involved in the near future.

Acknowledgments

This work is supported by the Strategic Priority Research Program of the Chinese Academy of Sciences, Grant No. XDA10010900, and in part by the CAS Center for Excellence in Particle Physics (CCEPP).

References

[1] Barrand G et al 2001 GAUDI - A software architecture and framework for building HEP data processing applications *Comput. Phys. Commun.* **140** 45
[2] Moll A 2011 The Software Framework of the Belle II Experiment *J. Phys.: Conf. Ser.* **331** 032024
[3] The JUNO Project URL: http://english.ihep.cas.cn/rs/fs/juno0815/
[4] Cao Zhen et al 2010 A future project at tibet: the large high altitude air shower observatory (LHAASO) *Chinese Phys. C* **34** 249
[5] The Boost C++ Libraries URL: http://www.boost.org/
[6] Brun R and Rademakers F 1996 ROOT - An Object Oriented Data Analysis Framework *Proceedings AIHENP96 Workshop, Lausanne, Sep. 1996.* 1997 *Nucl. Inst. and Meth. in Phys. Res.* A **389** 81-6. See also http://root.cern.ch/.