pLISA: Parallel Library for the Identification and Study of Astroparticle

Cristiano Bozza[1 2 *] and Giulia De Bonis [3 @]

[1]Dipartimento di Fisica “E. R. Caianiello”, Università di Salerno
[2]Istituto Nazionale di Fisica Nucleare (INFN), Gruppo Collegato di Salerno
[3]Istituto Nazionale di Fisica Nucleare (INFN), Sezione di Roma

Abstract

pLISA is a software tool under development in the framework of the OBELICS Working Package of the ASTERICS Research Infrastructure. The goal is to have a library with a collection of parameters, classes and methods to be linked and used in reconstruction programs. The library aims at offering the software scaffolding for applications developed inside the different experiments. The request for flexibility and interoperability is the leading idea of pLISA, that has been designed with an effort to overcome the specificity of any experimental facility. Another key point of the pLISA project is to be oriented towards parallel computing architectures, in particular at the opportunity of using GPUs as accelerators to enhance the performance of the algorithms, to reduce the processing time requested for the reconstruction tasks. Indeed, a fast (ideally, real-time) reconstruction can open the way for the developments of alert systems and for efficient multi-messenger programmes among the different experimental facilities involved in ASTERICS. Interoperability demands a detailed documentation, for promoting the usability of the software in different experiments. This report is a first step in this direction, since it presents an outline of the software with a focus on the key elements and on future developments.

CONTENTS

* cbozza@unisa.it
@giulia.debonis@roma1.infn.it

I. Introduction

ASTERICS (ASTronomy ESFRI and Research Infrastructure CluSter) [1] is a Research Infrastructure funded by the European Commission's Horizon 2020 framework. It collects experiences from astronomy, astrophysics and particle physics and aims at producing a common set of tools and strategies to be applied in the Astronomy ESFRI[1] facilities, pushing for carrying out a multi-messenger approach in data analysis and for creating synergies among different experiments and scientific communities.

Activities in ASTERICS are organised in five Working Packages (WP). WP3 is named **OBELICS** (OBservatory E-environments Linked by common ChallengeS) and focuses on **interoperability** and **software re-use** for data generation, integration and analysis. Specific tasks of action, aimed at promoting multi-wavelength/multi-messenger data analyses, are the establishment of open standards and software libraries, the development of common solutions for data processing and extremely large databases, the study of advanced analysis algorithms and strategies. All these elements of WP3 have been arranged in three sub-tasks:

- D-GEX, Data GEnereation and Information eXtraction (task 3.2)
- D-INT, Data systems INTegration (task 3.3)
- D-ANA, Data ANAlysis/intepretation (task 3.4)

The pLISA project is inserted in task 3.4 (**D-ANA**) and fulfils the mission and the objectives of WP3 OBELICS, in particular for what concerns interoperability. In addition, the OBELICS plans include a centre of attention towards the development of effective reconstruction methods and new graphical processing approaches, and the optimisation for **new computing technologies** and maximum efficiency. Following this line of research, the studies for the design of pLISA have been directed towards **Multi-Variate Analysis** (MVA) (as neural networks and boosted decision trees) and the implementations of the code have been devised as explicitly parallel, for running on GPUs since the very beginning. In this sense has to be intended the term "parallel" in the name of the package, while "Identification and Study" is the task to be accomplished using machine-learning techniques (MVA).

The people currently involved in pLISA are INFN-affiliated and have and a research experience within the ANTARES[4] and KM3NeT [5] Collaborations[2]. The study for the pLISA project has been initiated in the framework of KM3NeT and the peculiarity of ANTARES and KM3NeT data taking, trigger systems and reconstruction algorithms have been taken into account and have been the obvious starting point for the definition of the main features of pLISA. Nevertheless, an effort has been made to implement these features in a general way, in order to drop out any specific reference to the KM3NeT data format and to adapt the code structure for the data analysis of a generic event-based or signal-based experiment.

II. Overview of Existing Solutions

The first part of the study for pLISA has been dedicated to a preliminary overview of the current state of the art concerning MVA software, considering existing solutions available both in the scientific and in the industrial communities. Examples of existing tool-kits are TMVA (a ROOT-integrated project that provides a machine learning environment, developed in particular for the needs of high-energy physics experiments)[6] and SciKit-Learn (machine-learning in Python)[7]. Some attention has been paid on specific software implementations of machine-learning techniques in data analysis, considering in this case only activities carried out within the ANTARES and KM3NeT Collaborations. In addition, because of the focus on parallel architectures, a bird's eye survey on CUDA[8] and on the tools specifically designed for deep learning[9] has been accomplished.

[1]See [2] for the description of the mission and the scopes of the European Strategy Forum on Research Infrastructures (ESFRI) and see [3] for a detailed report of the current roadmap.

[2]KM3NeT is one of the facilities in the ESFRI roadmap.

III. IMPLEMENTATION OF pLISA

The programming language used for the implementation is C++. The code has been written as an **header file**, the current implementation is included In Appendix A. It has been compiled with *Visual C++* and *g++*[3] to check the correctness of the syntax, no errors or warnings have been produced.

The current implementation of pLISA is the "skeleton" of the code. Classes and constants are very strongly typed, and extensive use of namespaces has been made with two purposes:

- minimising the chances of name clashes when pLISA is used alongside other libraries;
- help developers and users in producing bug-free code by enforcing type compliance.

All the classes in the main header file of pLISA are *interfaces*, i.e. purely abstract classes containing only pure virtual methods. Adoption of explicit transient or persistent storage models has been carefully avoided for three reasons:

- every scientific community will have its own data model, and choosing one might lead to incompatibilities with others (e.g. useless or missing fields);
- GPU-based implementations will require data structures that will be different from those used in CPU-based implementations. As an example, in normal object-oriented coding, one would use AOS (*arrays-of-structures*) whereas for optimised performances and efficient memory access in GPU the SOA paradigm (*structure-of-arrays*) is almost unavoidable;
- a library that needs a specific data model also needs data converters and intermediate conversion steps that will use CPU, memory and even storage, thus largely spoiling the benefits of optimised parallel computing.

pLISA only puts requirements on the information that is to be provided to algorithms, not on the way they are stored. Among the advantages of this approach, it is worth recalling:

- the transient/persistent data model of user code need not be changed, provided "reader" classes are produced by users, with the sole purpose of exposing the properties required by pLISA extracting them from the user data;
- memory access is optimised: data that are in GPU can be read only on-demand, and proper seamless caching mechanisms can be implemented, e.g. including "lazy retrieval", i.e. retrieval of a full memory block only after a certain number of accesses are performed;
- by adding flags that describe the internal encoding, pLISA can optimise memory transfers and decide to skip them: in case of two chained algorithms working on GPU, if pLISA can determine that the output of the first algorithm is already in GPU, the second algorithm can start from there without having to move data back and forth to CPU;
- all non-implemented properties for data that are non-existing or meaningless do not take memory/disk space.

Shortcuts for memory access are foreseen but not yet defined in this document and the current version of pLISA. Their implementation is largely dependent on the best data model to be used for each specific algorithm, and is postponed to the final deliverable when a default set of functionalities is available. In case pLISA cannot determine that memory access shortcuts can be applied, fallback solutions are automatically used by accessing data properties, item-by-item. By this policy pLISA supports at the same time full generality and optimised performances.

The organisation of pLISA in namespaces is also convenient to follow for the purpose of giving an overview of the code. The outermost "container" is the namespace *pLISA*. It contains:

- the **FeatureSetId** type, used to identify pLISA-known types without using RTTI (*Run-Time-Type-Identification*).
- the **FeatureSet** class, whose basic method is to identify itself.

Because properties usually come in atomic bundles (e.g. it makes no sense to separate the three components of a vector), a **FeatureSet** provides a convenient way to dynamically query their availability as a group without sticking

[3] g++ (GCC) 4.8.5

to a storage data model. A specific implementation may or may not support a group of properties, but when it does all of its properties must be implemented.

Inside *pLISA*, the information is then further arranged according to a hierarchy, as presented in the following list:

- namespace ***Devices***
- namespace ***Hits***
- namespace ***Events***
- namespace ***Algorithms***

The next subsections give additional details on each namespace, in order to point out the main features and possible developments.

A. Devices

This section contains definitions used to describe the device on which data and software are supposed to work.

- **DeviceTypeId** currently defines *Host*, *GPU* and *MIC*. More types may be supported in the future.
- **InternalEncodingFlag** defines a flag mask that describes the internal encoding of a data structure. This is the foundation of pLISA memory access shortcut technology. By checking this flag, pLISA algorithms can determine what data formatting steps are needed to handle data.
- **DeviceMemoryBlock** is a generic pointer to a device-specific memory block.
- **DeviceMemoryAccessor** provides pLISA with direct access to memory blocks. The exact meaning of the block totally depends on the underlying data format, and must be known to both the class providing the access and the code that requests usage. The identifier that is passed is used to select the memory block. For example, in a structure that contains hits in a detector, one may have separate memory blocks for the X, Y, Z and T coordinates. The inner format is described by the *InternalEncoding*.

B. Hits

While higher-level algorithms for particle identification and energy estimate may use only the output of a fit, it is very common to face the need to go to the level of detector hits to extract relevant parameters for an estimator or classifier algorithm. The feature sets supported in the **Hit** class are the following:

- **Identification** contains only an integer number, which is the way pLISA uniquely identifies hits (specific detectors might have more sophisticated indexing mechanisms);
- **CustomIdentification** contains alternative hit indexing methods and must be provided by user code, since it depends explicitly on detector technology and segmentation;
- **Basic** provides basic space and time information on the hit;
- **Direction** is added in the case of directionally selective detector segments.
- **Weight** must be provided in all cases in which hits are weighted (e.g. according to energy deposition).
- **MonteCarlo** distinguishes real data from simulations;
- **CustomMonteCarlo** contains specific information for a set of simulated data. This must be provided by derived implementations.
- **Background** tells whether a hit has been added from the background.
- **CustomBackground** is used for custom background information.
- **Trigger** tells whether a hit triggered an event (alone or in combination).
- **CustomTrigger** is used for custom triggering information.
- **Reconstruction** tells whether a hit is part of an event reconstruction.
- **CustomReconstruction** is used for custom reconstruction information.

Finally, the **EventHitSet** roughly corresponds to sets of hits before and after a *possible* trigger. It has a timestamp and allows access to all the contained hits.

C. Events

This namespace represents events at several stages of reconstruction, hence with different feature sets that are filled as one goes through the steps in the analysis chain. Here below the various features supported by an **Event** are shown.

- **ParticleIdentification** provides the PDG identifier of the particle identified;
- **Hits** provides the set of hits from which the event was derived;
- **Track** contains geometrical track information without fitting errors, for the cases in which a track has been reconstructed;
- **TrackErrors** contains statistical information on the track fit, completely dependent on the algorithm and detector structure;
- **Shower** contains geometrical shower information without fitting errors, for the cases in which a shower has been reconstructed;
- **ShowerErrors** contains statistical information on the shower fit, completely dependent on the algorithm and detector structure;
- **Energy** contains the reconstructed energy;
- **EnergyErrors** contains statistical information on the energy fit, completely dependent on the algorithm and detector structure;
- **Momentum** contains the reconstructed momentum;
- **MomentumErrors** contains statistical information on the momentum fit, completely dependent on the algorithm and detector structure;

D. Algorithms

Every algorithm is fed with data from a previous step and produces new data accumulating information on the same event. The available steps are:

- **Trigger** is the basic analysis function;
- **Track** will represent a track reconstruction algorithm;
- **Shower** will represent a shower reconstruction algorithm;
- **ParticleIdentification** is exposed by particle identification algorithms;
- **Energy** is exposed by energy estimation algorithms;
- **Momentum** is exposed by momentum measurement algorithms.

A **Logger** class is also provided to be called by algorithm classes to monitor the various steps of the processing.

APPENDIX

```
#ifndef _PLISA_H_
#define _PLISA_H_

namespace pLISA
{
        namespace Devices
        {
                typedef unsigned DeviceTypeId;

                DeviceTypeId const Host = 0;
                DeviceTypeId const GPU = 1;
                DeviceTypeId const MIC = 2;

                typedef unsigned long long InternalEncodingFlag;

                namespace InternalEncodingFlags
                {

                        /* Reserved to pLISA-provided algorithms. All user code should avoid this flag. */
                        const InternalEncodingFlag pLISAReserved = (1ull << 63);
                        const InternalEncodingFlag InHostMemory = 1;
                        const InternalEncodingFlag InGPUMemory = 2;

                        /* All user code use this flag. */
                        const InternalEncodingFlag Custom = 0;
                }

                typedef void *DeviceMemoryBlock;

                class DeviceMemoryAccessor
                {
                public:
                        virtual InternalEncodingFlag InternalEncoding() = 0;
                        virtual DeviceMemoryBlock DirectAccess(unsigned blockid) = 0;
                        virtual DeviceTypeId Container() = 0;
                };
        }

        typedef unsigned FeatureSetId;

        class FeatureSet
        {
        public:
                virtual FeatureSetId GetFeatureSetId() = 0;
        };

        namespace Hits
        {
                namespace Features
                {
                        FeatureSetId const Null = 0x0;
                        FeatureSetId const Identification = 0x1;
                        FeatureSetId const CustomIdentification = 0x2;
                        FeatureSetId const Basic = 0x30;
                        FeatureSetId const Direction = 0x40;
                        FeatureSetId const Weight = 0x50;
                        FeatureSetId const MonteCarlo = 0x10;
                        FeatureSetId const CustomMonteCarlo = 0x20;
                        FeatureSetId const Background = 0x100;
                        FeatureSetId const CustomBackground = 0x200;
                        FeatureSetId const Trigger = 0x1000;
                        FeatureSetId const CustomTrigger = 0x2000;
                        FeatureSetId const Reconstruction = 0x10000;
                        FeatureSetId const CustomReconstruction = 0x20000;
                }

                class IdentificationSet : public FeatureSet
                {
                public:
                        virtual FeatureSetId GetFeatureSetId() { return Features::Identification; }
                        virtual unsigned int Id() = 0;
                };

                /* The custom identification feature set must be provided by the experiment code. */

                class BasicSet : public FeatureSet
                {
                public:
                        virtual FeatureSetId GetFeatureSetId() { return Features::Basic; }
                        virtual double X() = 0;
                        virtual double Y() = 0;
                        virtual double Z() = 0;
                        virtual double T() = 0;
                };

                class DirectionSet : public FeatureSet
                {
                public:
                        virtual FeatureSetId GetFeatureSetId() { return Features::Direction; }
                        virtual double DirX() = 0;
                        virtual double DirY() = 0;
                        virtual double DirZ() = 0;
                };

                class WeightSet : public FeatureSet
                {
                public:
                        virtual FeatureSetId GetFeatureSetId() { return Features::Weight; }
                        virtual double Weight() = 0;
                };

                class MonteCarloSet : public FeatureSet
                {
                public:
                        virtual FeatureSetId GetFeatureSetId() { return Features::MonteCarlo; }
                        virtual bool IsSimulated() = 0;
                };

                /* The custom montecarlo feature set must be provided by the experiment code. */


                class BackgroundSet : public FeatureSet
                {
                public:
                        virtual FeatureSetId GetFeatureSetId() { return Features::Background; }
                        virtual bool IsBackground() = 0;
                };

                /* The custom background feature set must be provided by the experiment code. */

                class TriggerSet : public FeatureSet
                {
                public:
                        virtual FeatureSetId GetFeatureSetId() { return Features::Trigger; }
                        virtual bool IsTriggered() = 0;
                        };

                        /* The custom trigger feature set must be provided by the experiment code. */

                        class ReconstructionSet : public FeatureSet
                        {
                        public:
                                virtual FeatureSetId GetFeatureSetId() { return Features::Reconstruction; }
                                virtual bool IsInReconstruction() = 0;
                        };

                        /* The custom reconstruction feature set must be provided by the experiment code. */

                        class Hit
                        {
                        public:
                                virtual FeatureSetId FeatureSets() = 0;
                                virtual bool HasFeatureSet(FeatureSetId mask) = 0;
                                /* Exceptions are thrown if attempting to get unsupported feature sets */
                                virtual IdentificationSet &Id() = 0;
                                virtual BasicSet &Basic() = 0;
                                virtual DirectionSet &Direction() = 0;
                                virtual WeightSet &Weight() = 0;
                                virtual TriggerSet &Trigger() = 0;
                                virtual FeatureSet &CustomTrigger() = 0;
                                virtual BackgroundSet &Background() = 0;
                                virtual ReconstructionSet &Reconstruction() = 0;
                        };

                        class EventHitSet : public Devices::DeviceMemoryAccessor
                        {
                        public:
                                virtual double UnixTime() = 0;
                                virtual unsigned Count() = 0;
                                virtual Hit &operator()(unsigned i) = 0;
                        };
                }

                namespace Events
                {
                        typedef long long PDGId;

                        namespace Features
                        {
                                FeatureSetId const Null = 0x0;
                                FeatureSetId const Hits = 0x1;
                                FeatureSetId const Track = 0x10;
                                FeatureSetId const TrackErrors = 0x20;
                                FeatureSetId const Shower = 0x40;
                                FeatureSetId const ShowerErrors = 0x80;
                                FeatureSetId const ParticleIdentification = 0x100;
                                FeatureSetId const Energy = 0x1000;
                                FeatureSetId const EnergyErrors = 0x2000;
                                FeatureSetId const Momentum = 0x10000;
                                FeatureSetId const MomentumErrors = 0x20000;
                        }

                        class TrackSet : public FeatureSet
                        {
                        public:
                                virtual FeatureSetId GetFeatureSetId() { return Features::Track; }
                                virtual double X1() = 0;
                                virtual double Y1() = 0;
                        virtual double Z1() = 0;
                        virtual double X2() = 0;
                        virtual double Y2() = 0;
                        virtual double Z2() = 0;
                        virtual double DirX() = 0;
                        virtual double DirY() = 0;
                        virtual double DirZ() = 0;
                };

                /* Track fitting errors depend on the algorithm. */

                class ShowerSet : public FeatureSet
                {
                public:
                        virtual FeatureSetId GetFeatureSetId() { return Features::Shower; }
                        virtual double X() = 0;
                        virtual double Y() = 0;
                        virtual double Z() = 0;
                        virtual double DirX() = 0;
                        virtual double DirY() = 0;
                        virtual double DirZ() = 0;
                        virtual double Radius() = 0;
                };

                /* Shower fitting errors depend on the algorithm. */

                class ParticleIdentificationSet : public FeatureSet
                {
                public:
                        virtual FeatureSetId GetFeatureSetId() { return Features::ParticleIdentification; }
                        virtual PDGId &Particle() = 0;
                };

                class EnergySet : public FeatureSet
                {
                public:
                        virtual FeatureSetId GetFeatureSetId() { return Features::Energy; }
                        virtual double Energy() = 0;
                };

                /* Energy fitting errors depend on the algorithm. */

                class MomentumSet : public FeatureSet
                {
                public:
                        virtual FeatureSetId GetFeatureSetId() { return Features::Momentum; }
                        virtual double Momentum() = 0;
                };

                /* Momentum fitting errors depend on the algorithm. */

                class Event : public Devices::DeviceMemoryAccessor
                {
                public:
                        virtual FeatureSetId FeatureSets() = 0;
                        virtual bool HasFeatureSet(FeatureSetId mask) = 0;
                        /* Exceptions are thrown if attempting to get unsupported feature sets */
                        virtual double UnixTime() = 0;
                        virtual long long Run() = 0;
                        virtual ParticleIdentificationSet &PID() = 0;
                        virtual Hits::EventHitSet &Hits() = 0;
                        virtual unsigned int GetTrackCount() = 0;
                        virtual TrackSet &Track() = 0;
```

```
                        virtual FeatureSet &TrackErrors() = 0;
                        virtual unsigned int GetShowerCount() = 0;
                        virtual ShowerSet &Shower() = 0;
                        virtual FeatureSet &ShowerErrors() = 0;
                        virtual EnergySet &Energy() = 0;
                        virtual FeatureSet &EnergyErrors() = 0;
                        virtual MomentumSet &Momentum() = 0;
                        virtual FeatureSet &MomentumErrors() = 0;
                };
        }

        namespace Algorithms
        {
                namespace Features
                {
                        FeatureSetId const Trigger = 0x04;
                        FeatureSetId const Track = 0x10;
                        FeatureSetId const Shower = 0x40;
                        FeatureSetId const ParticleIdentification = 0x100;
                        FeatureSetId const Energy = 0x1000;
                        FeatureSetId const Momentum = 0x10000;
                }

                enum ErrorSeverity
                {
                        Debug = -1,
                        Information = 0,
                        Warning = 1,
                        Error = 2,
                        CriticalError = 3,
                        FatalError = 4
                };

                class Logger
                {
                public:
                        virtual void Log(ErrorSeverity err_sev, unsigned err_code, const char *pText, unsigned st
                };

                class Algorithm
                {
                public:
                        virtual FeatureSetId GetFeatureSetId() = 0;
                        virtual bool HasFeatureSet(FeatureSetId mask) = 0;
                        virtual Events::Event &Process(Events::Event &ev, Logger &logger) = 0;
                };
        }
}

#endif
```

References

[1] ASTERICS web page.
[2] ESFRI web page.
[3] ESFRI Roadmap 2016 (*Strategy Report on Research Infrastructures*).
[4] ANTARES web page.
[5] KM3NeT web page.
[6] TMVA web page.
[7] Scikit-Learn web page.
[8] NVIDIA CUDA web page.
[9] NVIDIA Deep Learning web page.