Making distributed ALICE analysis simple using the GRID plug-in

A Gheata[1], M Gheata[1,2]
[1] Europeean Organization for Nuclear Research (CERN) - Geneva, Switzerland
[2] Institute for Space Sciences – Bucharest, Romania

E-mail: andrei.gheata@cern.ch

Abstract. We have developed an interface within the ALICE analysis framework that allows transparent usage of the experiment's distributed resources. This analysis plug-in makes it possible to configure back-end specific parameters from a single interface and to run with no change the same custom user analysis in many computing environments, from local workstations to PROOF clusters or GRID resources. The tool is used now extensively in the ALICE collaboration for both end-user analysis and large scale productions.

1. The ALICE computing model and data analysis

At a rate of up to *1.25 GBytes/second* data storage, the *ALICE* experiment has the biggest event size (tens of MBytes for Pb-Pb events) among the *LHC* experiments. This justifies the specific choices made for the *ALICE* computing model [1]. For scalability reasons, the reconstructed event summary data (*ESD*) is "democratically" replicated on the available distributed disk storages. The size of the data encouraged using a "pull" model in the *ALICE GRID* middleware, running jobs rather close to the data than on fixed sites and making *TIER 1* and *TIER 2* centers equivalent as role played in the data analysis chain. Moreover, the same resources are used for reconstruction, calibration and analysis tasks. All these facts, combined with a very ambitious physics program, are creating quite challenging scalability problems that had to be addressed by the analysis framework.

2. The analysis framework

The *ALICE* analysis framework was described in detail in a different paper [2]. In this section we just give a short overview of the components that are relevant for the way distributed analysis is steered. An *ALICE* analysis module is seen as an independent task that shares with others the main event loop steered by the framework. In fact, the event loop mechanism is provided by the *ROOT* framework [3] and the *ALICE analysis manager* is just a wrapper around the *ROOT TSelector* class. The advantage of this choice is that our framework can use directly the *ROOT* built-in tools for looping tree entries, including the formalism defined by the *TSelector* class [4] for the phases of data processing:

- Local initialization phase (corresponding to *TSelector::Init*) - allows initialization of the custom data structures for the analysis task. This phase is typically used to configure the analysis code.

- Remote initialization phase (corresponding to *TSelector::SlaveBegin*) – allows initializing the custom output objects produced by the analysis task (like booking histograms)
- Event processing phase (corresponding to *TSelector::Process*) - allows executing the custom task algorithm per event. This can be used as entry point for a specific user analysis framework
- Merging phase – This applies for all systems that allow parallel computing, where the workers produce only partial results that have to be merged. This phase is not directly defined by the *TSelector* interface but has to be handled by the framework.
- Post-processing phase (corresponding to *TSelector::Terminate* - allows any post-processing operation (like normalization, fitting) performed on the merged results

The analysis manager adds a specific model on top of the event loop, requiring that the user analysis follows the interface defined by a base analysis class (*AliAnalysisTask*). Using this interface, all analysis tasks registered to the manager are sequentially called in the different stages of processing. Analysis tasks are only allowed to communicate via output data, so that direct dependencies are avoided. This allows running together practically any combination of user analysis algorithms, taking profit from having the large-sized input data in memory. The analysis manager class acts here like a locomotive for a train having many analysis task wagons, while the deployment of the code, the data and merging services are provided.

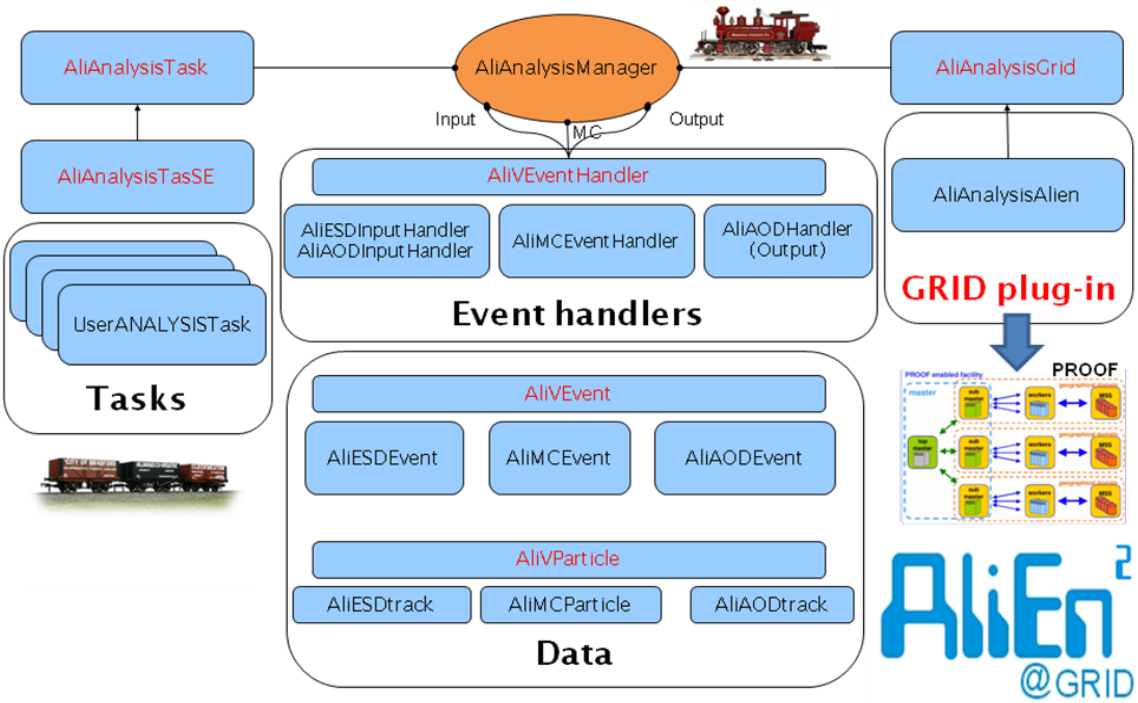

Figure 1. The *ALICE* analysis framework. The central role is taken by the analysis manager that steers the main event loop for a train of analysis tasks. The data services are provided via a base event handler class and specific data handlers. The distributed analysis is implemented as a plug-in that hides the complexity of the *PROOF* and *GRID* back-ends.

The reconstructed events are made available for analysis as event summary data, which in *ALICE* are rather large data structures containing information which is in many cases detector-specific. The *ESD* files can be skimmed or specialized to more compact formats (*AOD*, mini-*AOD*) for more specific analysis usage. The *ALICE* event model [5] derives from a base virtual class that makes the analysis

transparent to the existing specific event formats. The same applies at the level of tracks, vertices or other structures. Similar approaches have been used by few other experiments (like STAR and RHIC) in order to make it possible to use generic data handlers to dispatch data to many analysis clients. The ALICE analysis framework [2] extended this concept in the sense that any analysis task can become at its turn a data producer and dispatch this newly computed data to an arbitrary number of clients. The train of tasks presented in Figure 1 is just the trivial use case of the framework, which can propagate not only the input event, but also analysis results in a this data-driven tree of tasks.

One can include analysis wagons to an existing train (analysis manager) using custom ROOT macros. This is done via a simple standard procedure which is now a mandatory condition to be included in any train. The analysis trains were usually deployed using specific *GRID* or *PROOF* [6] setups. To make this operation simple and fully transparent for the user code we developed the so called **GRID plug-in**, described in detail in the following sections.

3. Distributed analysis in ALICE

There are two main use cases for distributed data analysis in *ALICE*. One can use dedicated *PROOF* clusters [7] to process a reasonable amount of data (like few millions events) with a fast turn-around cycle. This is quite useful to get to look into fresh new data, perform reconstruction and calibration tests or develop preliminary analysis cuts. A *PROOF* analysis session looks very much like a local one and uses the same analysis code. There are however specific settings that have to be made using the *TProof* interface in *ROOT* for: connecting to the *PROOF* cluster, staging/using datasets and configuring cluster parameters.

Once a certain analysis needs to run over a big part or the totality of the *ALICE* data in a period, it has to be deployed in the *ALICE GRID*. As described in the first section, distributed disk storage and computing resources are used here according to a "democratic" model. The system that allows running batch jobs on these resources is called **AliEn** (***Ali****ce* ***En****vironment*) [8]. It uses a specific job description language (*JDL*) to define: the software libraries to be used, the user executable code, the input data, the job splitting and other job parameters, together with the job output.

In practice, a user had to go through several steps before being able to run his analysis on the *GRID*: he/she needed to write the analysis code as an analysis task (as mentioned in the previous section), test and debug the code on local data, write the *JDL*, create a data set within the *AliEn* shell and copy all files required in the job input box. All this had to be done by keeping in sync the local and *GRID* software versions, monitoring the jobs and resubmitting in case of errors. The whole procedure (now obsolete) required a special training of rather qualified users. Gaining experience with the system was critical for the success rate.

A first step in making the system more user-friendly was the introduction of a web monitoring interface based on *Monalisa* [9]. This provides web interfaces for browsing and uploading files in the *AliEn* file catalog, monitoring the job status and user quotas, checking the job efficiency or inspecting central productions.

Efficiency is an important aspect for getting results out of a distributed system. The size of *ALICE* events encourages the use of CPU intensive analysis, which can be done by using big trains of analysis tasks, at the cost of extra (and sometimes limiting) memory requirements. Large productions have to be prioritized with respect to single user analysis. The life cycle of a given analysis passes through local and *PROOF* tests, fine tuning of cuts on the *GRID*, then entering when mature enough in a central production train conducted by one of the *ALICE* physics working groups. The whole procedure

was not straightforward and required a certain effort and time to conclude, so development of central services and tools to shorten it became a priority.

4. The AliEn plug-in

To hide the complexity of the *AliEn* system from the user we have developed a utility class to describe in a general way the requirements of a given analysis job. The idea was to provide an *API* to allow defining in a simple key-value manner all the settings that are specific to the *GRID* job like: time to live (*TTL*), software versions, dataset splitting or run numbers to be processed. Based on these settings we can automatically generate and upload the needed datasets, *JDL* and analysis files. The system supports loading existing *AliRoot* [10] libraries or compiling the analysis code on the fly. Using the *GRID* plug-in makes the analysis session look exactly like a local one, the only addition coming from the few extra plug-in configuration settings.

The implementation of this tool is based on the *TGrid* interface available in *ROOT*. This is specialized for the *AliEn GRID* via the *TAlien* class which allows executing shell commands in a programmatic way. This makes it possible to copy files directly to and from the *AliEn* file catalog, create datasets and most importantly, submit jobs and inspect their status. In practice, one has to connect in a plug-in fashion the utility ***AliAnalysisAlien*** to the analysis manager (as in figure 1). The way to configure it is straightforward and will be described in the next section. We wanted to provide a minimum amount of mandatory configuration settings (like software version and run numbers to be processed), while all the expert settings have default values.
While in a local session, the *AliEn* plug-in inserts into the analysis manager initialization stage without any change in the user calling sequence. The manager just detects its presence and delegates all *GRID*-specific tasks to it. To make the local analysis run on the *GRID*, the plug-in performs the following tasks:

- Connect to the *GRID* via the *TGrid::Connect("alien:")*. This uses the existing user token and is a mandatory step for being able to communicate with the *AliEn* system;
- Create the remote user working and output directories to hold the initial files and the future job output. A check for the possibility to copy files is performed by default at this stage;
- Check the availability of the requested input data and produce a dataset via the *AliEn* "*find"* command. The created dataset is checked to contain at least one entry and then copied to the *GRID* workspace.
- Write the initialized analysis manager from memory to a file, then copy it on grid storage
- Generate a steering analysis macro that is able to extract the analysis manager from file and launch a local analysis on a subset of the input data provided by the splitting procedure in *AliEn*. This macro is also copied in the input box and will be used to run the analysis on the remote batch nodes.
- Generate a *ROOT* macro to merge the outputs of the analysis jobs. This is an important step in distributed data analysis that otherwise has to be done separately.
- Generate validation scripts for both the analysis and merging jobs. These check after the job completion the presence of all the output files declared by the tasks connected to the manager. They also check the presence of a special end-of-job file marker named *outputs_valid* produced by the analysis manager in case of successful processing.
- Finally generate and upload the *JDL's* corresponding to the analysis and merging jobs. These keep the whole setup coherent and incorporate all the user-defined or default *GRID* configuration settings

An important part of this automatic procedure is the fact that all errors cause a well-documented execution abortion to prevent starting invalid code while on the GRID, which was very far from being the case before putting in place the plug-in. Besides that, handling all the information from a single

point allows for a very powerful testing phase that was implemented in this tool from the very beginning.

Once the job input box and JDL are assembled as described above, the *AliEn* plug-in submits a configurable number of master jobs and starts automatically an *AliEn* shell. Note that all this happens from within the starting phase of a **local** analysis macro that hides completely the *GRID* API. The job submission is done gradually in case several runs are to be processed, to avoid overloading the job queue. Once the analysis jobs are submitted, the user has complete freedom to inspect the jobs status, resubmit the ones failing due to recoverable grid errors or leave the session.

An important feature of the plug-in is the ability to resume an interrupted session once a given phase of processing is completed. This first happens when the user finds out that all sub-jobs are in a final state and need to be merged. At this point the user needs to simply re-run his plug-in enabled local analysis macro, marking the start of the termination phase. The different analysis phases are described in the next section.

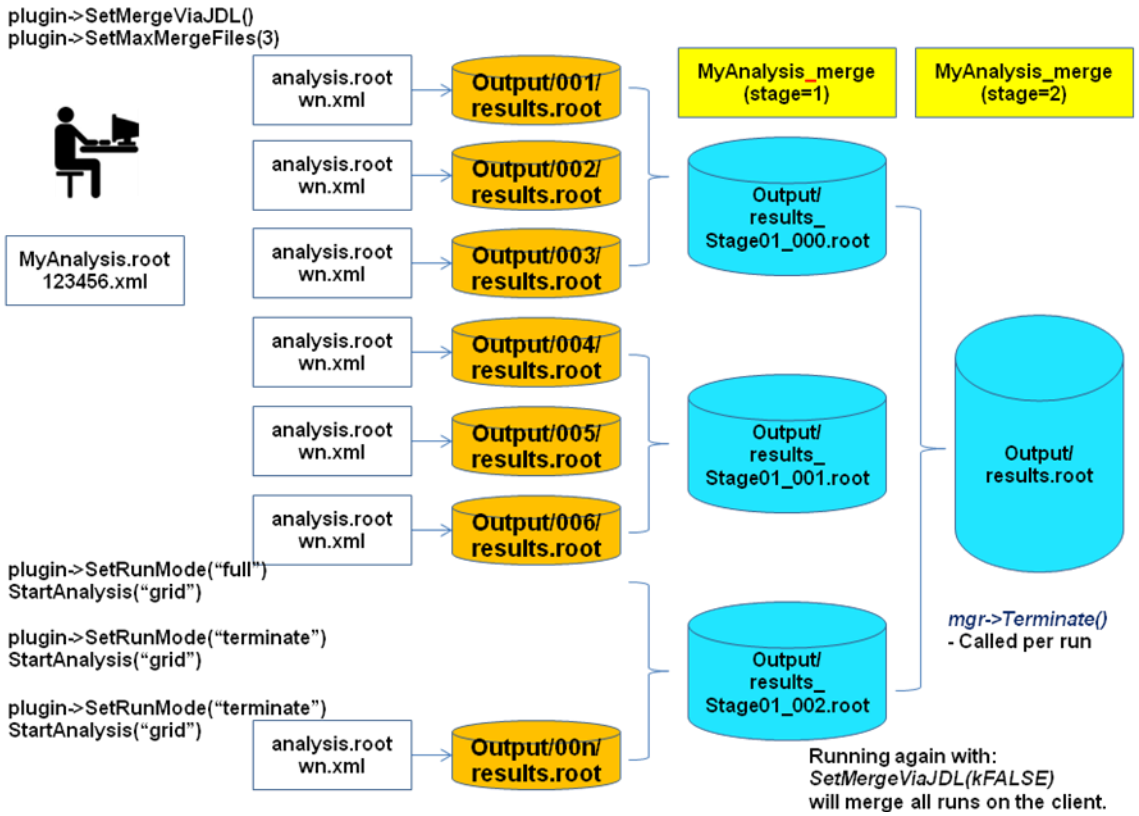

Figure 2. Merging scheme using the AliEn plug-in. The number of file per stage and the maximum number of stages are configurable.

After resuming the session, the tool first makes a new collection based on the output files available in the job output directories. This collection is used as input for the merging jobs that are subsequently submitted. We just mention here that merging is a specific phase in any distributed analysis, being a very complex problem in itself and subject to scalability issues. Merging is very sensitive to grid problems like load on storage and temporary network glitches, but also to memory limitations due to the type of the objects to be merged.

The *AliEn* plug-in alleviates this issue by providing a multi-stage iterative merging procedure that ends-up with a single file per master job (Figure 2). The number of merging stages is configurable and the user has full control on (re)starting the next merging stage. The intermediate merging stages are run as independent grid jobs, taking as input a configurable fraction of the output files produced by the previous stage (or the analysis jobs in case of the first merging stage). The final merging stage is somewhat special since it runs as a single job, taking all previous outputs in one go. After performing the merging, the *Terminate()* phase of the analysis manager is automatically executed (executing post-processing analysis on the merged data) and the final merged files are copied to the grid output directory.

5. Run modes and configuration

As described in the previous section, the AliEn plug-in defines default values for most JDL parameters. These are however not always appropriate for a given analysis and need to be customized. The most important ones are described in this section.

The tool allows several different run modes to give a maximum of flexibility to the user analysis. The run mode has to be set as string via the method *SetRunMode()* of the plug-in. The supported run modes are described below:

- *"full"* – This mode is performing the full procedure described above, excluding the merging phase. If *GRID* merging is disabled or the number of stages is set to 0, merging is however executed locally. The plug-in makes a single collection over the job output files and starts a local file merger. This feature is however discouraged for more than 100 outputs. While doing local merging, the plug-in backs-up the last merged result so that in case of failure merging can be resumed. In case merging is done locally, the post-processing *Terminate()* phase is also executed locally.
- *"offline"* – This mode allows working with the plug-in without a *GRID* connection, skipping most tests and generating only the needed files. This allows modifying the intermediate analysis files or *JDL's*. This kind of configuration can be error prone and is discouraged, but in some cases it is very useful. Combined with the plug-in production mode it can be used for example to configure production jobs to be run by a different user.
- *"submit"* – This is used in correlation with the offline mode to submit the customized analysis files. It will first perform the remaining checks as in the full mode then it will submit the jobs
- *"terminate"* – This mode is used as trigger to pass to the next analysis phase. When the analysis jobs are done, one needs to put the plug-in in this mode and rerun the local analysis macro to start the merging phase. In case merging is done in stages, one needs to do the same to trigger the start of the next merging stage. As rule of thumb, it is important that all the previous jobs submitted by the plug-in are in a final state before running in this mode, otherwise the corresponding outputs will not be used.
- *"test"* – This mode performs a very powerful local test using the exact same files that are to be deployed and run on the *GRID*. The test first makes a collection over a configurable number of files from the requested input dataset. This is used to access the input data remotely via *xrootd* in the test session. The analysis is executed as on a *GRID* batch node and has to pass before running the full mode. The test mode is a unique feature of this tool and helps a lot to avoid most possible errors. Some of the features cannot be tested however, like the validity of the *JDL*, which in this case is not generated. Users have to make sure that the local software versions correspond to the ones that will be demanded on the *GRID*. To reproduce scalability issues it is recommended to set the number of test files the same as the maximum job split level. The test mode is available also in *LOCAL* or *PROOF* modes, as described in the next section.

Table 1. Most relevant plug-in configuration settings

AliEn plug-in method	Description
SetRunMode	Sets the plug-in run mode as described above
SetProductionMode	Produce files to be deployed and run by the production manager.
SetROOTVersion *SetAliROOTVersion* *SetAPIVersion*	Set requested package versions.
SetGridDataDir *SetGridWorkingDir* *SetGridOutputDir*	Define the grid directories
AddRunNumber *SetRunRange* *AddDataFile* *SetDataPattern*	Define run numbers to be processed or custom collection files. Set the pattern to be used for searching available input files (e.g. **AliESDs.root*)
SetAnalysisSource	Source code in case of dynamic compilation
SetExecutable *SetAnalysisMacro* *SetValidationScript* *SetJDLName*	Define custom names for the generated files
AddAdditionalLibrary *SetAdditionalLibs*	Instruct the plug-in to load these libraries (in the given order)
SetSplitMaxInputFileNumber	Maximum number of files to be allocated to a sub-job
SetDefaultOutputs	Output file names as defined by analysis tasks
SetOutputFiles	Customize the output files
SetOutputArchive	Define how the outputs should be archived
SetTerminateFiles	Extra files produced in the termination phase
SetMergeViaJDL	Enable merging as *GRID* job (default)
SetMergeExcludes	List of outputs to be ignored for merging
SetTTL	Time to live for sub-jobs
SetNtestFiles	Number of files used by the test mode
SetMaxInitFailed	Number of failed jobs to trigger abortion
SetMasterResubmitThreshold	Percentage threshold for automatic resubmission

6. Extension to local and PROOF modes

The plug-in has been recently upgraded to handle also the local and PROOF cases, so from the user perspective the computing infrastructure becomes completely transparent. The usage is straightforward, requiring the addition of few lines in the plug-in configuration. One just needs to have the plug-in connected to the analysis manager and start the analysis in the desired mode via: *AliAnalysisManager::StartAnalysis(const char *mode)*. The supported modes are: *"local"*, *"proof"* and of course *"grid"*. The table below presents the extra settings to be activated in case of PROOF.

Table 2. Plug-in configuration settings in PROOF mode

AliEn plug-in method	Description
SetProofCluster	Define the PROOF cluster to run the analysis
SetProofDataSet	Specify the dataset to be used. This has to be

	available via *gPROOF->ShowDatasets()*
SetProofReset	Reset the user session. Supported modes: *0-no reset, 1-soft, 2-hard*
SetNproofWorkers	Limit the number of workers (faster initialization)
SetNproofWorkersPerSlave	Define more than one worker per slave
SetRootVersionForProof	Request a specific *ROOT* version
SetAliRootMode	Load analysis libraries by default
SetClearPackages	Clear existing packages
SetProofConnectGrid	Connect to *GRID* on every worker
SetFileForTestMode	Define a file name containing pointers to data files stored locally, to be used for testing

A remarkable feature is that the test mode implemented initially for the *GRID* case was preserved also for the *PROOF* mode. This is activated in the same way, using the method *SetRunMode("test")* for the plug-in. In this case the file for the test mode must be defined. The test activates the *PROOF-lite* feature, which emulates a *PROOF* master on the local machine, using as many slaves as *CPU* cores. The analysis code is tested on the requested data without any connection to the *PROOF* cluster.

7. Future perspectives for organized analysis

The *AliEn* plug-in is just a simple utility that emerged from the need to make it easy for users to deploy analysis on large GRID data sets. Its usage increased very fast as well as the number of requests to support new features. This triggered several new developments and a tight integration with the GRID monitoring system, such that the tool became a key component of the analysis framework. A recent extension for *PROOF* support has made it even more useful, making all *ALICE* resources transparent and easy to use.

Besides end-user analysis, most central analysis productions are submitted based on this tool. We are currently making an effort to extend the plug-in functionality aiming to ease-up automatic generation and testing of custom analysis trains created via web forms. The idea is to provide tools that will allow for train "conductors" from the different *ALICE* physics working groups to assemble and manage trains using existing analysis modules from their group. This is now possible after standardizing the procedure of inclusion of an analysis task, and the *ALICE* analysis libraries contain already the analysis code in the necessary form. The gain in developing this new feature (already in an advanced beta-testing phase) is not only from a management perspective but also from a job efficiency one.

The implementation of this new feature required defining a new object type which describes the configuration needed to run a given analysis task. This is composed of:

- Location of the *ROOT* macro that adds the given task to an existing analysis manager. These macros already exist in the *AliRoot* library, so their path is defined with respect to the base installation location of this software.
- Required libraries to be loaded, in the right order.
- Supported input data types (*ESD*, *AOD*, *MC*). This is used to exclude incompatible analysis modules from a given train.
- Dependency analysis modules. Some analysis tasks represent general-purpose utilities (like centrality selection or physics selection) that are requested by most analysis modules even though there is no direct dependency between them. The analysis plug-in makes sure that dependencies are always included.
- Configuration for event handlers. Any train needs event handlers that may be differently configured.

This configuration can be imported from text files generated via the web forms. The configuration objects are internally looped and the macros adding each wagon are being executed. This finally creates the analysis train in memory and at this point a test can be generated by the plug-in. The generated test can be run on a test machine by the train administrator, who gets a detailed report on the CPU and memory footprint for EACH composing wagon. The idea is to create reference data to follow the evolution of a given analysis task. All errors coming out of the tests are pointing directly to the code to blame. The plug-in is also able to generate the final code to be run on the GRID, if all tests succeed.

8. Acknowledgements

The tools described in this paper were mainly developed by the authors, but there are many other people that contributed with ideas, developments or fixes. We would like to thank A. Morsch, J.F.Grosse-Oetringhaus, M.Vala, C.Grigoras for their direct or indirect participation to the development, together with all *ALICE* analysis users that were actively using this tool and giving their feedback.

9. References

[1] Carminati F, Schutz Y for *ALICE* coll., *ALICE Computing Model*, CERN-LHCC-2004-038/G-086
[2] Gheata A, *ALICE Analysis Framework*, PoS (ACAT08) 028
[3] *http://root.cern.ch*
[4] *http://root.cern.ch/root/html/TSelector.html*
[5] *http://aliweb.cern.ch/secure/Offline/sites/aliweb.cern.ch.Offline/files/uploads/OfflineBible.pdf*
[6] Ballintijn M, Roland G, Brun R and Rademakers F, *The PROOF distributed parallel analysis framework based on ROOT*, Proc. Conf. for Computing in High-Energy and Nuclear Physics (*CHEP*), La Jolla, California, 24-28 Mar 2003
[7] Grosse-Oetringhaus J F, *The CERN Analysis Facility - A PROOF Cluster for Day-One Physics Analysis*, *JPCS* **119** (2008) 072017
[8] Bagnasco S, Betev L, Buncic P, Carminati F, Cirstoiu C, Grigoras C, Hayrapetyan A, A. Harutyunyan, A.J. Peters, P. Saiz, *AliEn: ALICE Environment on the GRID*, Proc. Conf. for Computing in High-Energy and Nuclear Physics (*CHEP*), Victoria, Canada, 2-9 Sep 2007
[9] Legrand I C, Newman H B, Voicu R, Carstoiu C, Grigoras C, Toarta M, Dobre C, MONALISA: *An agent based, dynamic service system to monitor, control and optimize GRID based applications*, Proc. CHEP 2004
[10] F. Carminati, P. Buncic, P. Hristov, A. Morsch, F. Rademakers, K. Safarik, *The AliRoot framework, status and perspectives*, Proc. Conf. for Computing in High-Energy and Nuclear Physics (*CHEP*), Interlaken, Switzerland 27 Sep - 1 Oct 2004