jacal Documentation

YANDA Team

Feb 22, 2022

CONTENTS:

Joint **A**stronomy **CAL**ibration and imaging software

CHAPTER

ONE

INTRODUCTION

JACAL integrates Yandasoft (previously known as *ASKAPSoft*) and the execution framework DALiuGE. A shared library offers a calling convention supported by DALiuGE and internally links and reuses Yandasoft code. JACAL is freely available in GitLab under a variation of the open source BSD 3-Clause [License](LICENSE). The repository contains the following:

- The C/C++ code of the shared library `libjacal.so` described above.
- A number of tests running the different components inside DALiuGE graphs.
- A standalone utility for library testing independent of DALiuGE.

The repository is an offshoot from the original located in GitHub. The latter should be considered deprecated, and has only been left available for reference.

1.1 DALiuGE apps

The way jacal integrates Yandasoft into DALiuGE is by wrapping individual pieces of functionality into DALiuGE-compatible applications that can then be deployed on a DALiuGE graph.

DALiuGE is an execution framework where programs are expressed as directed acyclic graphs, with nodes representing not only the different computations performed on the data as it flows through the graph, but also the data itself. Both types of nodes are termed *drops*. Computation drops (in DALiuGE, *application drops*) read or receive data from their input data drops, and write the results into their output data drops. Data drops on the other hand are storage-agnostic and host-agnostic, meaning that regardless of underlying storage and location application drops can work with their inputs and outputs in the same way.

Although application drops can be implemented in many ways, DALiuGE offers out-of-the-box support for certain type of applications. Among those, *shared libraries* can be written by users to implement application drops. This capability allows reusing code written in C, C++ or other low-level languages to work as application drops in a DALiuGE graph.

1.2 Using Yandasoft in DALiuGE

Before JACAL, the only way to use the Yandasoft functionality was to invoke the binaries it generates (e.g., `cimager`, `cbpcalibrator`, etc.); composition was only possible by arranging pipelines using shell scripts and similar techniques, and with data having to touch disk between each invocation of the binaries.

JACAL on the other hand implements a shared library (i.e., `libjacal.so`) wrapping different parts of Yandasoft as DALiuGE-ready application drops. This makes it possible to reuse finer-grained pieces of functionality from the Yandasoft code base, and with data not having to be necessarily written to disk between these steps.

CHAPTER

TWO

INSTALLATION

2.1 Dependencies

Jacal has two main dependencies (which in turn might require a lot more):

- The DALiuGE execution framework, and
- The Yandasoft libraries

Installation for both dependencies is covered below:

2.1.1 DALiuGE

DALiuGE is written in python and has publicly available releases in PyPI:

```
pip install daliuge
```

Alternatively one can install it directly from its GitHub repository:

```
pip install git+https://github.com/ICRAR/daliuge
```

In both cases all dependencies will automatically be built and installed. Most are offered as binary wheels and require no compilation, but some do; hence a compiler will be needed.

2.1.2 Yandasoft

Yandasoft is written in C++ and uses the CMake build system for its installation. The source code that makes up Yandasoft is not contained in a single repository but in a few, which can be found here. However, a separate "integrated" repo brings them all together into a single build pass, with options to skip building some of the repositories if one doesn't need them.

Note: At the moment of writing, jacal builds against the `develop` branch of Yandasoft.

Not all Yandasoft components are required by jacal. Therefore to build Yandasoft in preparation for jacal the following instructions would be required:

```
git clone https://github.com/rtobar/all_yandasoft
cd all_yandasoft
./git-do clone
mkdir build
```

(continues on next page)

(continued from previous page)

```
cd build
cmake .. -DBUILD_ANALYSIS=OFF -DBUILD_PIPELINE=OFF
make
make install
```

Note that Yandasoft has a list of dependencies on its own. These include:

- casacore and casarest
- wcslib
- cfitsio
- fftw
- boost
- log4cxx
- gsl

These need to be installed before attempting to install Yandasoft, but instructions to do so are outside the scope of this document.

2.2 jacal

Once all dependencies are installed, jacal itself can be built. jacal uses the CMake build system, hence the build instructions are those one would expect:

```
git clone https://gitlab.com/ska-telescope/jacal
cd jacal
mkdir build
cd build
cmake ..
make
```

This process should generate a `libjacal.so` shared library which one can use within DALiuGE's `DynlibApp` components. Two stand-alone executables are also produced under `test` which are used for testing the code outside the context of DALiuGE.

CHAPTER

THREE

EXAMPLE USAGE

In this page we briefly describe how to use jacal in a DALiuGE graph. This assumes you already *built jacal*.

3.1 Outline

In this example we will replicate one of the unit tests run in the GitLab CI pipeline, namely `test_basic_imaging`. This test performs basic imaging on an input MeasurementSet using the *CalcNE* and *SolveNE* jacal components. The other unit tests work similarly, exercising different jacal components in different modes of operation.

In DALiuGE a program is expressed as a *graph*, with nodes listing applications, and the data flowing through them. Graphs come in two flavours: *logical*, expressing the logical constructs used in the program (including loops, gather and scatter components), and *physical*, which is the fully-fledged version of a logical graph after expanding all the logical constructs.

This test is expressed as a *logical graph*. After translation into a *physical graph* it is submitted for execution to the DALiuGE *managers*, which need to be started beforehand. During execution one can monitor the progress of the program via a browser.

3.2 Preparing the graph

This test needs a few inputs:

- The logical graph.
- A parset (parsets are text files containing configuration options, and are the configuration mechanism used throughout yandasoft).
- Some input data.

Put all three files above in a new directory, and then decompress the input data:

```
$> mkdir tmp
$> cd tmp
$> export TEST_WORKING_DIR=$PWD
$> wget https://gitlab.com/ska-telescope/jacal/-/raw/master/jacal/test/daliuge/test_
↪basic_imaging.json?inline=false
$> wget https://gitlab.com/ska-telescope/jacal/-/raw/master/jacal/test/daliuge/test_
↪basic_imaging.in?inline=false
$> wget https://gitlab.com/ska-telescope/jacal/-/raw/master/data/chan_1.ms.tar.gz?
↪inline=false
```

(continues on next page)

(continued from previous page)

```
$> tar xf chan_1.ms.tar.gz
$> PARSET=$PWD/test_basic_imaging.in
```

Next, some adjustments will need to be made to the graph so that the jacal shared library can be found, and the parset is correctly read at runtime:

```
$> sed -i "s|%JACAL_SO%|$PATH_TO_JACAL_SO|g; s|%PARSET%|$PARSET|g" test_basic_imaging.
↪json
```

3.3 Starting DALiuGE

Firstly, one needs to start the DALiuGE *managers*, the runtime entities in charge of executing graphs. We will start two: the *Node Manager* (NM), in charge of executing the graph, and a *Data Island Manager* (DIM), in charge of managing one or more NMs. Note that starting the DIM is not strictly required, but is done for completeness.

Start the managers each on a different terminal so you can see their outputs independently. Also, to make the test simpler, start both in the same directory where the downloaded files are placed:

```
$> cd $TEST_WORKING_DIR
$> dlg nm -v
$> dlg dim -N 127.0.0.1 -v
```

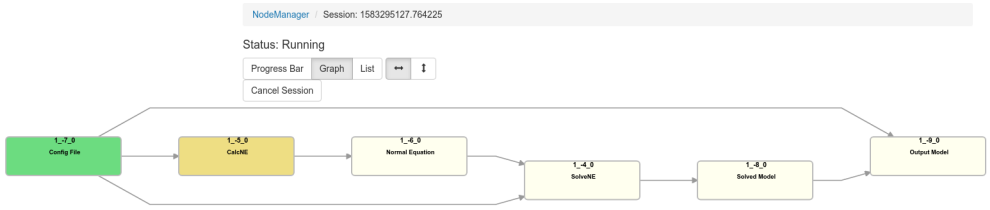
3.4 Running

To execute a graph we submit it to one of the DALiuGE managers (in our case, the DIM). Also, because we are starting from a logical graph, we need to transform it into a physical graph that can be run on the deployed managers.

This can be done as follows:

```
$> cd $TEST_WORKING_DIR
$> cat test_basic_imaging.in \
   | dlg unroll-and-partition `# Logical -> Physical translation` \
   | dlg map                  `# Assign nodes to drops (i.e., schedule the graph)` \
   | dlg submit -w            `# Submit and wait until execution finishes`
```

Finally, connect to `127.0.0.1:8000` to see the graph running:



CHAPTER

FOUR

API

4.1 Available applications

class **CalcNE** : public askap::*DaliugeApplication*
CalcNE.

Calculates the Normal Equations

This class encorporates all the tasks to read from a measurement set; subtract a model; grid residual visibilities and FFT the grid

EAGLE_START

EAGLE_END

Param gitrepo

Param version

Param category DynlibApp

Param param/libpath/LibraryPath/%JACAL_SO%/String/readonly [in] The path to the JACAL library

Param param/Arg01/Arg01/name=CalcNE/String/readonly [in]

Param port/Config/LOFAR::ParameterSet [in] ParameterSet descriptor for the image solver

Param port/Model/scimath::Params [in] Params of solved normal equations

Param port/Normal/scimath::ImagingNormalEquations [out] ImagingNormalEquations to solve

class **InitSpectralCube** : public askap::*DaliugeApplication*
InitSpectralCube.

Build the output image cube

This class builds the output cube in the format specified by the parset.

EAGLE_START

EAGLE_END

Param gitrepo

Param version

Param category DynlibApp

Param param/libpath/LibraryPath/%JACAL_SO%/String/readonly [in] The path to the JACAL library

Param param/Arg01/Arg01/name=InitSpectralCube/String/readonly [in]

Param port/Config/LOFAR::ParameterSet [in] The Config file

Param port/Cube [out]

class **LoadNE** : public askap::*DaliugeApplication*
LoadNE.

Example class that simply loads Normal Equations from a drop

Implements a test method that uses the contents of the the parset to load in a measurement set and print a summary of its contents. We will simply load in a NormalEquation from a daliuge drop and output the image. This simply tests the NE interface to the daliuge memory drop.

EAGLE_START

EAGLE_END

Param gitrepo

Param version

Param category DynlibApp

Param param/libpath/LibraryPath/%JACAL_SO%/String/readonly [in] The path to the JACAL library

Param param/Arg01/Arg01/name=LoadNE/String/readonly [in]

Param port/Normal/scimath::ImagingNormalEquations [in] ImagingNormalEquations to solve

class **LoadParset** : public askap::*DaliugeApplication*
LoadParset.

Load a LOFAR Parameter Set in the *DaliugeApplication* Framework

Loads a configuration from a file drop and generates a LOFAR::ParameterSet The first ASKAP example in the Daliuge framework that actually performs an ASKAP related task. We load a parset into memory from either a file or another Daliuge drop_status

EAGLE_START

EAGLE_END

Param gitrepo

Param version

Param category DynlibApp

Param param/libpath/LibraryPath/%JACAL_SO%/String/readonly [in] The path to the JACAL library

Param param/Arg01/Arg01/name=LoadParset/String/readonly [in]

Param port/Config/LOFAR::ParameterSet [in] ParameterSet descriptor for the image solver

Param port/Config/LOFAR::ParameterSet [out]

Param port/Config/LOFAR::ParameterSet [out]

class **LoadVis** : public askap::*DaliugeApplication*
LoadVis.

Loads a visibility set, grids it onto the UV plane and FFTs the grid

Loads a configuration from a file drop and a visibility set from a casacore::Measurement Set

EAGLE_START

EAGLE_END

Param gitrepo

Param version

Param category DynlibApp

Param param/libpath/LibraryPath/%JACAL_SO%/String/readonly [in] The path to the JACAL library

Param param/Arg01/Arg01/name=LoadVis/String/readonly [in]

Param port/Config/LOFAR::ParameterSet [in] The Config file

Param port/Model/scimath::Params [in] Params of solved normal equations

Param port/Normal/scimath::ImagingNormalEquations [out] ImagingNormalEquations to solve

class **MajorCycle** : public askap::*DaliugeApplication*
MajorCycle.

Loads a visibility set, grids it onto the UV plane and FFTs the grid

Loads a configuration from a file drop and a visibility set from a casacore::Measurement Set

EAGLE_START

EAGLE_END

Param gitrepo

Param version

Param category DynlibApp

Param param/libpath/LibraryPath/%JACAL_SO%/String/readonly [in] The path to the JACAL library

Param param/Arg01/Arg01/name=MajorCycle/String/readonly [in]

Param port/Config/LOFAR::ParameterSet [in] The Config file

Param port/Solved Model/scimath::Params [in]

Param port/Cube [in]

Param port/Normal/scimath::ImagingNormalEquations [out] ImagingNormalEquations to solve

class **NESpectralCube** : public askap::*DaliugeApplication*
NESpectralCube.

Build an output image cube from input NormalEquations

This class builds the output cube is whatever format specified by the parset. Generates a cube of NormalEquation slices.

EAGLE_START

EAGLE_END

Param gitrepo

Param version

Param category DynlibApp

Param param/libpath/LibraryPath/%JACAL_SO%/String/readonly [in] The path to the JACAL library

Param param/Arg01/Arg01/name=NESpectralCube/String/readonly [in]

Param port/Config/LOFAR::ParameterSet [in] ParameterSet descriptor for the image solver

Param port/Normal/scimath::ImagingNormalEquations [in] ImagingNormalEquations to solve

class **OutputParams** : public askap::*DaliugeApplication*
OutputParams.

Solves an Normal Equation provided by a Daliuge Drop. Outputs the Params class as images.

Implements an ASKAPSoft solver. This essentially takes a NormalEquation and generates a a set of "params" usually via a minor cycle deconvolution. We will simply load in a NormalEquation from a daliuge drop and solve it via a minor cycle deconvolution. This drop actually generates the output images based upon the contents of the Params object.

EAGLE_START

EAGLE_END

Param gitrepo

Param version

Param category DynlibApp

Param param/libpath/LibraryPath/%JACAL_SO%/String/readonly [in] The path to the JACAL library

Param param/Arg01/Arg01/name=OutputParams/String/readonly [in]

Param port/Config/LOFAR::ParameterSet [in] ParameterSet descriptor for the image solver

Param port/Model/scimath::Params [out]

class **RestoreSolver** : public askap::*DaliugeApplication*
RestoreSolver.

Implements an ASKAPSoft Restore solver. This essentially takes a NormalEquation and a set of "params" and creates a restored image.

This takes a configuration and a set of normal equations and uses the Solver requested in in the ParameterSet to produce an ouput model.

EAGLE_START

EAGLE_END

Param gitrepo

Param version

Param category DynlibApp

Param param/libpath/LibraryPath/%JACAL_SO%/String/readonly [in] The path to the JACAL library

Param param/Arg01/Arg01/name=RestoreSolver/String/readonly [in]

Param port/Config/LOFAR::ParameterSet [in] The Config file

Param port/Model/scimath::Params [in] Params of solved normal equations

Param port/Normal/scimath::ImagingNormalEquations [in] ImagingNormalEquations to solve

Param port/Restored Model/scimath::Params [out]

class **SolveNE** : public askap::*DaliugeApplication*
SolveNE.

Implements an ASKAPSoft solver. This essentially takes a NormalEquation and generates a set of params usually via a minor cycle deconvolution.

This takes a configuration and a set of normal equations and uses the Solver requested in in the ParameterSet to produce an ouput model.

EAGLE_START

EAGLE_END

Param gitrepo

Param version

Param category DynlibApp

Param param/libpath/LibraryPath/%JACAL_SO%/String/readonly [in] The path to the JACAL library

Param param/Arg01/Arg01/name=SolveNE/String/readonly [in]

Param port/Config/LOFAR::ParameterSet [in] The Config file

Param port/Normal/scimath::ImagingNormalEquations [in] ImagingNormalEquations to solve

Param port/Model/scimath::Params [out] Params of solved normal equations

class **SpectralCube** : public askap::*DaliugeApplication*
SpectralCube.

Build the output image cube

This class builds the output cube is whatever format specified by the parset.

EAGLE_START

EAGLE_END

Param gitrepo

Param version

Param category DynlibApp

Param param/libpath/LibraryPath/%JACAL_SO%/String/readonly [in] The path to the JACAL library

Param param/Arg01/Arg01/name=SpectralCube/String/readonly [in]

Param port/Config/LOFAR::ParameterSet [in] The Config file

Param port/Model/scimath::Params [in] Params of solved normal equations

Param port/Cube [out]

4.2 Others

class **DaliugeApplication**
Daliuge application class.

This class encapsulates the functions required of a daliuge application as specified in dlg_app.h then exposes them as C functions

Subclassed by *askap::CalcNE*, *askap::InitSpectralCube*, askap::JacalBPCalibrator, *askap::LoadNE*, *askap::LoadParset*, *askap::LoadVis*, *askap::MajorCycle*, *askap::NESpectralCube*, *askap::OutputParams*, *askap::RestoreSolver*, *askap::SolveNE*, *askap::SpectralCube*

class **DaliugeApplicationFactory**
Factory class that registers and manages the different possible instances of of a *DaliugeApplication*. .

Contains a list of all applications and creates/instantiates the correct one based upon the "name" of the Daliuge DynLib drop. Maintains a registry of possible applications and selects - based upon a name which one will be instantiated.

class **NEUtils**
set of static utility functions for the NE manipulation

These are just a set of static functions I use more than once

CHAPTER

FIVE

INDICES AND TABLES

- genindex
- modindex
- search

INDEX

A