# SDP Memo 071: LOFAR pipeline DALiuGE

Document number…………………………………………………...SDP Memo 071
Document Type………………..……………………………………..MEMO
Revision……………………..………………………………………….1
Author……………………….…………………………………………Grange & Wu
Release Date………………….…………………………………...2018-09-02
Document Classification……..……………………………………. Unrestricted

| **Lead Author** | **Designation** | **Affiliation** |
|---|---|---|
| Yan Grange | | ASTRON |
| Signature & Date: | | |

| Revision | Date of Issue | Prepared By | Comments |
|---|---|---|---|
| 1 | 2nd September 2018 | Yan Grange<br>Chen Wu | Initial publication |

## SDP Memo Disclaimer

## Table of Contents

## Introduction

This memo reports the work associated with the SDP ticket (TSK-1878). The main goal of this ticket is to demonstrate the suitability of the DALiuGE execution framework (Wu et al. 2017) for the execution of pipelines of radio astronomy, in this case LOFAR (van Haarlem et al. 2013).

The memo is organised as follows: The next section discusses details on the hardware and pipelines used in the demonstrations and gives a short introduction to DALiuGE. The next section shows how to set up and execute the demonstration pipelines. This is followed by an overview of the graphs of the ported pipelines. This is all followed by a set of recommendations for the DALiuGE developers based on the experience with porting the LOFAR pipelines, and a conclusion.

## Environment setup

This section briefly introduces the hardware and software used to perform the demonstration. All software is available via public github repositories and the versions used to generate the figures in this memo are tagged “sdp-0.1” so that the demonstration could be reproduced.

### Software

The pipelines that we discuss here are all executed using the LOFAR software suite, version 2.21.4 and prefactor version 2.0.2, extended with several python wrappers. The specifics of the build process can be found in the Dockerfiles corresponding to each pipeline.

#### DALiuGE

The Data Liu[1] Graph Engine (DALiuGE; Wu et al. 2017) is an execution framework based on graphs. It is specifically designed to support the very large-scale processing needed for the reduction of interferometric radio astronomy data sets. Large data sets from existing radio telescopes have already been processed using DALiuGE. DALiuGE was originally developed as a prototyping activity within the Square Kilometre Array (SKA) Science Data Processor (SDP) design consortium aimed at prototyping the execution framework of the architecture proposed for the SDP.

#### Pipelines

##### The CALIB pipeline

The first set of experiments were performed using a pipeline that defines a very simple imaging procedure provided by a LOFAR user (see e.g. Iacobelli et al. 2013 for work based on this pipeline). In this work, this pipeline will be referred to as the *CALIB* pipeline. The code is a very straight-forward python script that executes commands in sub-processes. Even though this script is not fully representative, we believe that it represents a minimal working example of an imaging pipeline and porting it to DALiuGE constitutes the first step towards

[1] Liu represents the Chinese character for “flow”

a full-scale graph execution with DALiuGE. Based on those considerations, we will first analyse the porting of CALIB to DALiuGE.

#### Prefactor

The pipeline that is being introduced to generate LOFAR images in production is the prefactor pipeline (van Weeren et al. 2016; Williams et al. 2016)[2]. The goal of this pipeline is to make an initial (direction independently) calibrated image that can be used for facet calibration using the factor pipeline (hence the name *pre*factor). Because this pipeline is developed to be used in production, it would be a very good test case to port to DALiuGE. An added advantage could be that any work on this pipeline could at a later stage be extended with the facet calibration. Also, the prefactor pipeline is used as the demonstrator for the European Open Science Cloud (EOSC) projects related to LOFAR, aimed at reimplementing pipelines using the Common Workflow Language, making a future comparison of different pipeline and execution framework implementations to DALiuGE possible.

The prefactor pipeline consists of three parts. The first part derives the calibration solutions based on a calibrator observation. The second part applies those solutions to the observed target. The last part finally generates an image of the calibrated target. In the following, we will refer to the first step only, since this is the only step that was ported to DALiuGE for this work.

### Hardware

We performed the experiments that are described in this document on the ASTRON node of the Distributed ASCI Supercomputer 5 (DAS5; Bal et al. 2016). The system consists of a heterogeneous set of machines. A docker daemon has been deployed to several nodes, including the head node. For this work Docker version 18.06.0-ce has been used

Four nodes in the system run a docker daemon that is used to deploy a version of the LOFAR software stack and a version of DALiuGE. However, since the only shared file system available is an NFS file system that does not allow file locking, the pipeline had to be executed locally, meaning that testing with more than one concurrent node could not succeed.

## Pipeline development and execution

### Managers drops, Logical graph and physical graph

In DALiuGE, the main unit is called a drop. A *drop* represents either a piece of data (either a file or a memory address) or an executable object (e.g. a command line executable, python script, compiled C library). A pipeline is represented in the form of a graph in which drops are connected (the connections representing the flow of data in the system.

A *logical graph* is an abstract representation of a pipeline in generic terms. It defines how the data flows between drops. Based on the logical graph and information on the available hardware, a *physical graph* can be generated. This graph represents a mapping of each processing unit. If a pipeline processes a single data set, the logical and physical graphs are essentially the same.

[2] http://github.com/lofar-astron/prefactor

In DALiuGE, the workflow is to create a logical graph and let the system convert that in a physical graph, which uses the *physical graph Template Partitioning,* which maps a physical graph to hardware nodes, essentially showing what block of the processing chain is executed on what node, and how those are connected. DALiuGE implements several algorithms to perform this action. Each algorithm strives to minimise a certain value (e.g. amount of data movement, execution time). In this work, we only use “Algorithm 1” which minimises the amount of data movement in the system.

The compute cluster is set up as a *data island*. Centrally, there is one *data island manager* (dim) that takes care of distributing the work over the worker nodes. In this work we will be using the DALiuGE web interface to launch the jobs on the cluster.

## Pipeline setup

The DALiuGE pipelines can be found on github. In the following, all shell commands, as well as file and directory names are documented in `typewriter font` inside grey cells. We assume that the root working directory is available through the variable ${WORK_ROOT}. Each line containing a command starts with a `$`.

For both pipelines, the setup of the corresponding container has been put in a github repository. To ensure reproducibility we created a pre-release in both githubs (named sdp-0.1) of the version of the code used for the experiments in this document.

## Setting up DALiuGE

To run, the following steps need to be executed:

1. Checkout the DALiuGE framework:

```
$ cd ${WORK_ROOT}
$ git clone https://github.com/ICRAR/daliuge
$ cd DALiuGE
```

2. Build the Docker images for DALiuGE (edit the Dockefile to not pull the master branch but the tag you actually want to build; in our case version 0.5.0):

```
$ cd ${WORK_ROOT}/DALiuGE/docker/base
$ ./build.sh
$ cd ${WORK_ROOT}/DALiuGE/docker/dfms
$ ./build.sh
```

## Prefactor preparation

For prefactor, the installation is very similar

```
    $ cd ${WORK_ROOT}
    $ git clone
https://github.com/ygrange/prefactor-DALiuGE
    $ cd prefactor-DALiuGE
    $ git checkout tags/sdp-0.1
    $ cd lofar-dlg-base
    $ ./build.sh
    $ cd ../lofar-dlg
    $ ./build.sh
```

**NB** This step may fail because the version of wcslib has been removed. Please change the version to a version present on the csiro ftp server[3]. Also note that the user id within the image is hard coded (to 1024; which is the id of the user used for our experiments). Please adapt to your own situation by replacing the line “ENV UID 1024” in the file `Dockerfile` by “ENV UID {my_id}” where {my_id} is the user id of the user executing the pipelines (and owning the work directories and source files) on the host system.

## CALIB preparation

For practical reasons, the CALIB Dockerfile depends on the prefactor docker files. So if you only want to repeat the CALIB experiment, you still have to build the prefactor images as well (or build your own docker image with LOFAR, awimager, and DALiuGE compiled in). To get up and running with CALIB, download and build the repository

```
    $ cd ${WORK_ROOT}
    $ git clone https://github.com/ygrange/CALIB
    $ cd CALIB
    $ git checkout tags/sdp-0.1
    $ docker build -t dlg-calib/centos7 .
```

Again, **Note** that the user id within the image is hard coded. Please adapt to your own situation as explained in the paragraph above (in the file `Dockerfile` inside the `lofar-dlg-base` directory).

## Starting the DALiuGE services

In what follows, we will assume the cluster that we execute on consists of one head node, that runs both the DALiuGE web editor and the data island manager.

On each of the worker nodes, a DALiuGE node manager should be started. Using the following command

```
    $ docker run {image} dlg nm -w {workdir} -H 0.0.0.0 -l
{logdir} -vv
```

In this command, the place holder {image} should be replaced by the docker image used for the experiment (i.e.  dlg-calib/centos7 or dlglofar/centos7) {workdir} and {logdir} should respectively be replace by the working directory and logging directory, within the image. We use the -H flag to ensure that DALiuGE is bound to any of the interfaces. Alternatively a specific interface IP can be specified here. The last flag is optional but turns on a significant amount of logging, which should make debugging more easy. For sharing data sets between host and container, users are referred to the relevant parts of the Docker documentation[4].

To start the data island manager on the head node, a similar command is used

```
    $ docker run {image} dlg dim -w {workdir} -H 0.0.0.0 -l
{logdir} -N {node01},{node02} -vv
```

[3] ftp://ftp.atnf.csiro.au/pub/software/wcslib; note that wcslib.tar.bz2 is a link to the most recent version.
[4] https://docs.docker.com/storage/bind-mounts/

The {image} placeholder could basically be any image that has DALiuGE installed (e.g. dfms/centos7) since it does not need to be able to execute any processing jobs. Using the -N flag, we specify two nodes that are managed by this island manager (identified by the placeholders {node01} and {node02}, which should either be hostnames or ip addresses).

The next step is starting the web service and execute the pipelines, a local copy of the logical graphs is needed. It can be downloaded from github.

```
   $ cd ${WORK_DIR}
   $ git clone
https://github.com/ICRAR/daliuge-logical-graphs
```

Then, the web interface can be started using

```
   $ docker run -v ${WORK_DIR}:/dlg-logical-graphs {image}
dlg lgweb -d /dlg-logical-graphs -t {lgweb-workdir}
```

Where the {lg-web-workdir} placeholder should be a directory where the manager will write its physical graphs. **Note** that the `dlg-logical-graphs` directory will only be writable (and therefore graphs be editable and savable) if a image is used that has a user with the same user id as the owner of the directory. In this case it may therefore be wise to use the dlglofar/centos7 image.

For commodity, the prefactor-DALiuGE repository contains a directory called `compose-files`, containing docker-compose yml definitions for an example setup of head node and worker nodes. Do note that the compose files will most probably not work since ip addresses, usernames and data locations are hard coded.

## Pipeline graphs and execution

In this section, we show the logical graphs of the CALIB and prefactor pipelines. Then, we show an example of a physical graph of the pipeline running on the demo data. The demo data can be found on the ASTRON ftp server [5] in a directory named CALIB or prefactor.

### Execution of the pipelines in the web interface

With the lgweb system running on the head node, one can now open a browser and connect to the head node ip/host name on port 8084. Here, a gui is shown in which one can select, edit and run pipelines. For execution, one first needs to select a pipeline in the list on the right-hand side and click the “Translate” button. As mentioned before, in what follows we will be using “Algorithm 1” with 1 node for CALIB and 4 for prefactor, and otherwise default settings. Please note that in the screen where the Physical Graph Template Partitioning has been created, the default value of the DALiuGE Manager host should be changed from “localhost” to the ip/hostname of the head node. To execute, one can then click the “Generate & Deploy Physical Graph” button and wait for the pipeline to complete.

[5] ftp://ftp.astron.nl/outgoing/SDP_grange/ ; please note that this directory can be automatically cleaned regularly. Please contact one of the authors for access to the data if needed and unaccessible.

## CALIB pipeline graph

The logical graph of the CALIB pipeline is shown in Fig 1. The CALIB pipeline is very straightforward since every step operates on the same subband and no fitting is performed on the calibration solutions of each subband. The whole pipeline is within the same scatter block, since all steps are executed for every subband.

Since several steps modify tables and do not generate output files, those are executed using a wrapper, written in python, that writes a Donemark file, which is used to trigger the next step to be executed.

The physical graph of an execution of the pipeline using two sub bands is shown in Fig, 2. The two “tracks” represent the units of work (there are two because we are processing two sub bands). Green blocks represent jobs that have successfully ended, dark yellow blocks are still executing. Light yellow blocks are blocks that still need to be executed. Failed blocks are coloured in red. Whenever a block fails, all next block will also become red.

**Note** that failure is defined by the exit status of the executable. Since we use several python wrappers around executables that can have a correct exit status for a failing job, the error may actually be part of the last block that “succeeded”, which may make debugging a bit more complex. However, running the node managers with significant verbosity makes it possible to both see what command was executed and what the output of that command was, for each command on each node.

## CALIB pipeline demo data

The data used in the demonstration is the data of two sub target bands and two calibrator sub bands of LOFAR obsid L182806, averaged in time by a factor 64. Each of the sub bands are 2.5 GB in size.

**Note** that the CALIB pipeline originally used the LOFAR global sky model database to obtain sky models for the data. Since this database has not been accessible from the outside and more mature imaging pipelines use different methods to obtain sky models, we added a sky model for the demo data in the demo image. This means that for using the pipeline on a different data set, the sky model provided in the repository should be replaced by a sky model of the observed target before building the docker image.

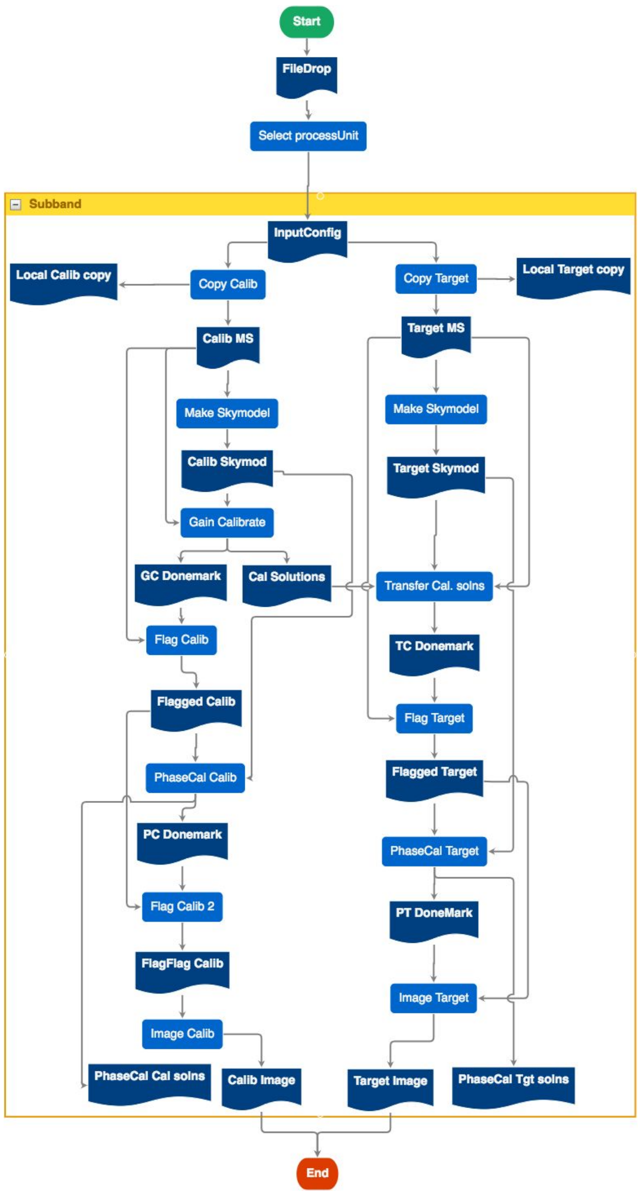

**Figure 1.** Logical graph of the CALIB pipeline. File drops are indicated in dark blue and application drops (i.e. shell executables) in light blue.

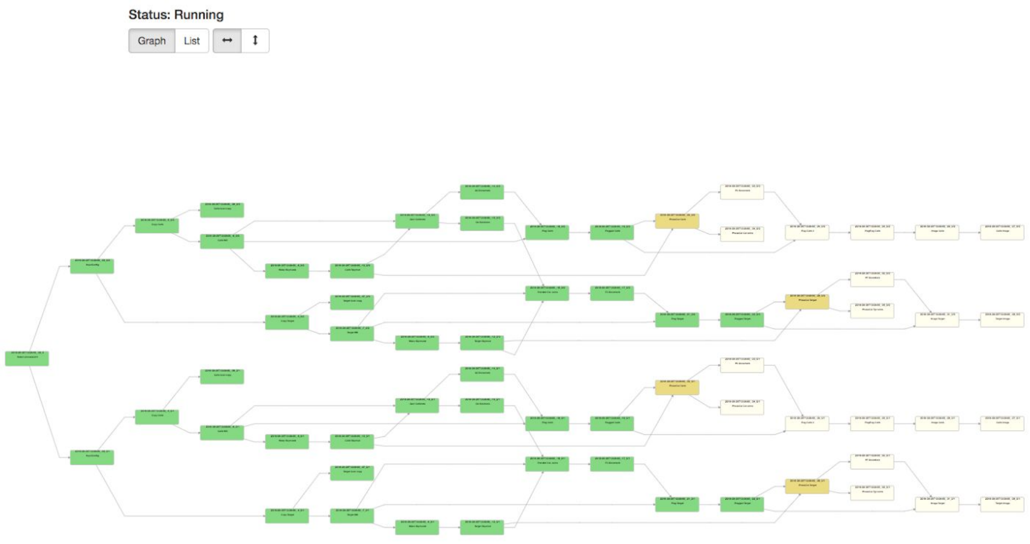

**Figure 2.** Physical graph of the CALIB pipeline for two subbands of LOFAR. Green units represent units of work that have successfully completed. The dark yellow boxes represent the work units that are currently executing and the light yellow boxes represent jobs that will be executed.

## prefactor pipeline graph

The logical graph of the prefactor pipeline is shown in Fig 2. After a few steps that are executed for each subband independently, the calibration solutions are merged to a single HDF5 file after which several values are smoothed and fitted across sub bands. Also in the prefactor we made use of the concept of python drops (i.e. python functions, shown in cyan) and application drops (i.e. command line executables, shown in light blue).

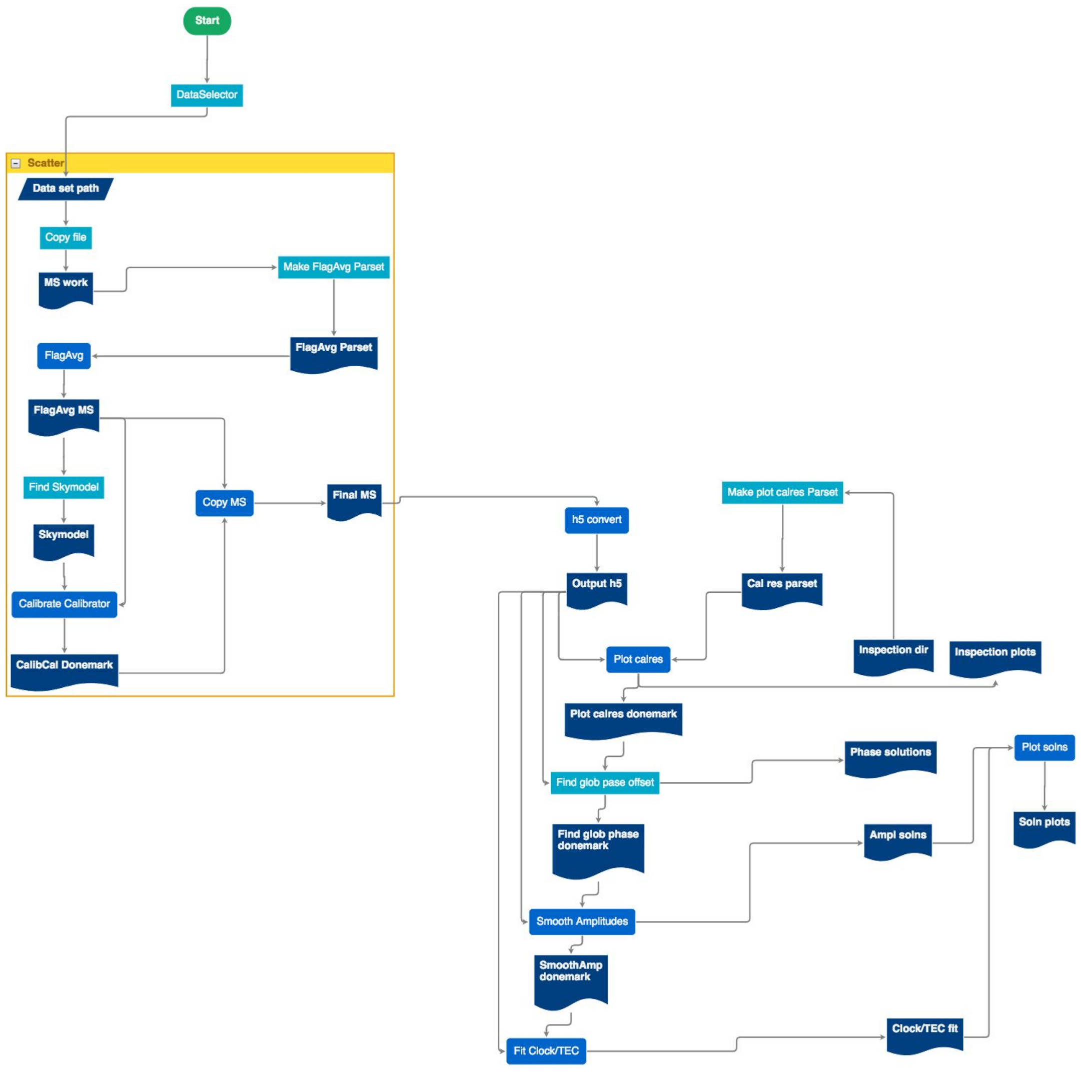

**Figure 3.** Logical graph of the prefactor pipeline. File drops are indicated in dark blue, pyton drops in cyan, and application drops (i.e. shell executables) in light blue. The in-memory drop block is visualised as a dark-blue trapezoid.

## Prefactor pipeline demo data

The data used for the demo of prefactor is based on data from the first 20 sub bands of the LOFAR observation with obsid L570741 (target) and L570745 (calibrator). The choice for 20 is based on the fact that prefactor needs this amount of sub bands to complete. Because the goal of the demonstrator is to test the functionality of DALiuGE, we select the data of the first 10 time slots of each subband, making the pipeline run through rather quickly. The full version of the data can be found in the LOFAR long-term archive.

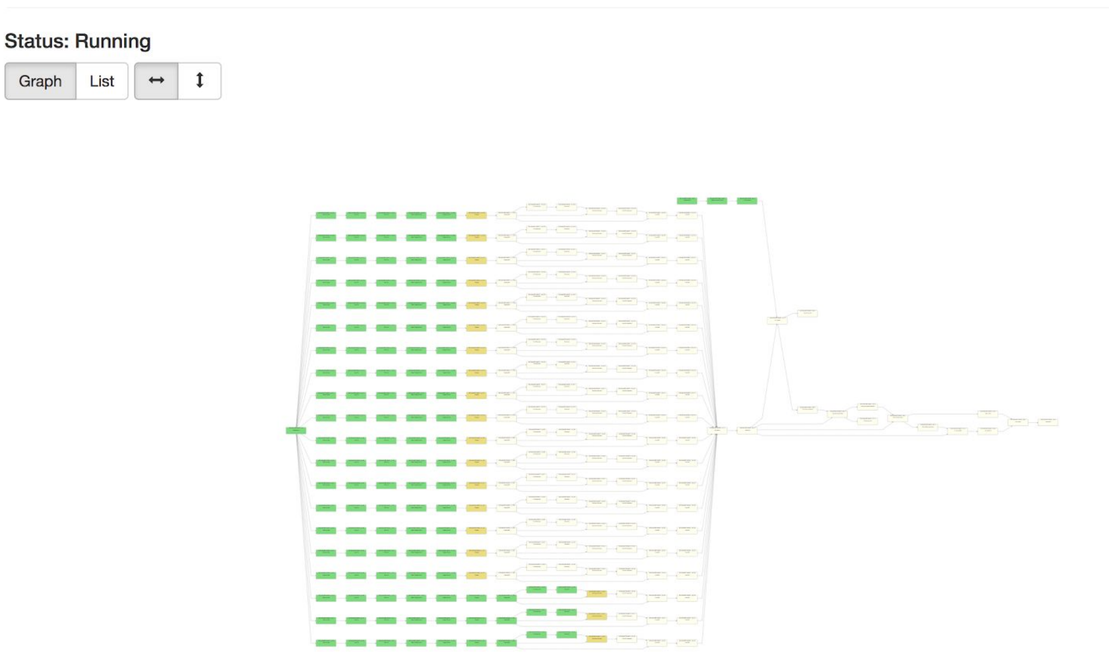

**Figure 2.** Physical graph of the prefactor pipeline for two subbands of LOFAR. Green units represent units of work that have successfully completed. The dark yellow boxes represent the work units that are currently executing and the light yellow boxes represent jobs that will be executed.

## Recommendations based on the experience

Based on the experience in porting LOFAR-based pipelines to DALiuGE, we have several recommendations to improve DALiuGE. In this sections, those are listed. Most recommendations are concerned with (1) implementing functionalities that would make these specific pipelines more readable and (2) minimising the number of wrappers that have to be written in order to port the pipelines. It should explicitly be noted that those considerations have no direct relation to the design principles of DALiuGE, and may very well be contrary to them. In this section, the recommendations are grouped by themes, and ordered by their importance, as perceived by the authors.

### Data access and storage.

1. To save space on disk, several steps in the pipelines modify the input data product in place, meaning that the concept of “input” and “output” file does not exist as such. As visible in both logical graphs, the way to work around this is to wrap the command in a script that writes out a DoneMark file (or an In-Memory Drop). This is because currently DALiuGE strictly follows the dataflow principle where a data drop is the only way to express the dependency between two application Drops.. This issue is being resolved in the latest DALiuGE EAGLE editor, where an Event edge (with a customizable ‘type’) can be used to link two Application Drops that do not need to have common input or output data items.

2. In some cases, the processing can be distributed over nodes that each contains a subset of the data on local storage to circumvent possible issues that may arise when using network file systems. Also, LOFAR processing on an NFS mounted file system proved to be unreliable because of locking issues. Right now, both pipelines start with an implicit copy step to move the data away from NFS to local storage. A specific copy drop that keeps track of file locality may be very useful for data management. This issue would be easy to tackle when using tools specifically developed with DALiuGE in mind, since those could write output to Memory Drops, which support remote reads and writes. But that means the logical graph needs to add these memory drops, which are responsible for transferring data across the cluster (without being instructed to do so explicitly), dumping the data as a temporary files on local disks, and passing the file handler back to BashShell Drops once they complete their execution(e.g. data transferring). Since the BashShell Drops cannot directly access Memory Drops (e.g. a shell script cannot directly access memory blocks of another running process), in this proof-of-concept pipeline, we have decided to use the (shared) file systems for convenience, which caused the above issue. This issue leads to two recommendations. First, the logical graph editor should allow pipeline developers to specify that a group of Drops should always be executed on the same compute node. The other suggestion would be to investigate if the file system access (open, seek, read, write,  etc) of an application could be handled through a library which is part of DaLiuGE.
3. In the current version of DALiuGE, file drops ar either named by the system, or have a hardcoded name. In some cases, a command will for example add a postfix to a file name provided. It would be good to have the option to configure a file drop in such a way that it will have a file name containing a parametric part (e.g. figure_drop_{XYZABC}.png, where XYZABC is filled in by DALiuGE at run time). Also, a configurable output location may be a directory that is expected to exist, instead of a file. Having the option to have a directory drop generated at run time would be helpful in this case.

### Access of non-file data

1. In some parts of the pipeline, a shell command is used to get a small amount of data, that will be used as a parameter for the next command. Think about a command that would be used to get the name of the source from a measurement set. The current way to implement this is to wrap the fist command in a python drop that writes to a memory drop, which is read by the next python drop that wraps the next command. Basically this would lead to two requirements. Fist the piping of stdout of an appdrop to a MemoryDrop. Second, the mapping of a MemoryDrop to the contents of a command line parameter. It would be nice if one could redirect stdout of an application drop to a memory drop, and use the corresponding value as a parameter on the command line of a different command in stead. The main use case for this would be porting already existing pipelines to DALiuGE, since pipelines written with the system in mind would have different ways of implementing this.
2. In some cases, the number of command line arguments of a command can be flexible. If the command is a python function that expects an arbitrary number of options (i.e. **kwargs or *args), it would be very useful if DALiuGE would allow the

pythondrop to have access to all the argsXX set in the DROP parameters in a dict, like the default behaviour of **kwargs.

### Pipeline specification

1. The Common Workflow Language (CWL; Amstutz et al. 2016) has been developed to make it possible to describe workflows in a way that is independent of the execution engine. Several institutes in the radio astronomical community are investigating its viability as a system to describe radio astronomy workflows. The language seems to define a subset of the possibilities offered by DALiuGE. It would however lower the threshold to experiment with DALiuGE, port pipelines to DALiuGE or execute user-written pipelines on a DALiuGE-based system if pipelines written in CWL could be converted to and/or executed by DALiuGE. For example, it appears rather straightforward to let DALiuGE generate CWL-based physical graph templates.

### GUI recommendations

1. When experimenting, it is often not fully obvious on what node a job failed which means all logs need to be processed. It would help if in the data island manager, the nodes could be tagged, so that those tags would be printed in the web GUI when running a pipeline. Being able to access the logs from within the web interface may also be a useful feature, even  in large-scale production environments. This feature is currently being integrated in SDP TSK-2546
2. In the design GUI, copy/pasting blocks or parts leads to double ids occurring since the id in the drop parameter is copied together with the drop. I would be useful to have this automatically changing.
3. In the prefactor pipeline example, there are 20 measurement sets that have to be distributed. To change this number, both the number in the “scatter” box in DALiuGE and the number in the python script that is used to select the data (the “DataSelector”) need to be changed. It would be nice if this would be configurable in a single place. Since the number of files of an observation may not a priori be known, being able to specify it at run time may be useful. This does however complicate the conversion of a logical to a physical graph.

### Other recommendations

1. Since all the current pipelines are hosted on github, it would be very useful if a script would exist to visualise the pipeline as it is on github, in a similar way as is done by the CWL viewer[6] for CWL.
2. When debugging, it would be very helpful if DAliuGE allows developers to rerun the pipeline from a certain stage, which could potentially save  significant amounts of time.

## Conclusions

In this document, the effort to port two LOFAR-based pipelines to the DALiuGE framework is presented. Since the demonstrations are based on Docker containers, the experiments are reusable and reproducible. In conclusion, the DALiuGE framework can be

[6] https://view.commonwl.org/; see the examples on the web site.

used to executed those already existing LOFAR-based, albeit with some changes to the way the commands are executed.

## References

Amstutz, P., M. R. Crusoe, N. Tijanić, B. Chapman, J. Chilton, M. Heuer, A. Kartashov, et al. 2016. “Common Workflow Language, v1.0.” https://doi.org/10.6084/m9.figshare.3115156.v2.

Bal, Henri, Dick Epema, Cees de Laat, Rob van Nieuwpoort, John Romein, Frank Seinstra, Cees Snoek, and Harry Wijshoff. 2016. “A Medium-Scale Distributed System for Computer Science Research: Infrastructure for the Long Term.” *Computer* 49 (5): 54–63.

Haarlem, M. P. van, M. W. Wise, A. W. Gunst, G. Heald, J. P. McKean, J. W. T. Hessels, A. G. de Bruyn, et al. 2013. “LOFAR: The LOw-Frequency ARray.” *Astronomy & Astrophysics* 556 (August): A2.

Iacobelli, M., M. Haverkorn, E. Orrú, R. F. Pizzo, J. Anderson, R. Beck, M. R. Bell, et al. 2013. “Studying Galactic Interstellar Turbulence through Fluctuations in Synchrotron Emission: First LOFAR Galactic Foreground Detection.” *Astronomy & Astrophysics* 558 (October): A72.

Weeren, R. J. van, W. L. Williams, M. J. Hardcastle, T. W. Shimwell, D. A. Rafferty, J. Sabater, G. Heald, et al. 2016. “LOFAR FACET CALIBRATION.” *The Astrophysical Journal Supplement Series* 223 (1): 2.

Williams, W. L., R. J. van Weeren, H. J. A. Röttgering, P. Best, T. J. Dijkema, F. de Gasperin, M. J. Hardcastle, et al. 2016. “LOFAR 150-MHz Observations of the Boötes Field: Catalogue and Source Counts.” *Monthly Notices of the Royal Astronomical Society* 460 (3): 2385–2412.

Wu, C., R. Tobar, K. Vinsen, A. Wicenec, D. Pallot, B. Lao, R. Wang, et al. 2017. “DALiuGE: A Graph Execution Framework for Harnessing the Astronomical Data Deluge.” *Astronomy and Computing* 20: 1–15.