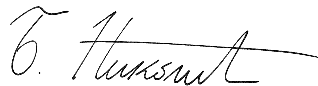
Practical Distributed Data Processing using SWIFT/T & CASA

Bojan Nikolic
Astrophysics Group, Cavendish Laboratory, University of Cambridge, UK

August 4, 2017



Revision History

Revision	Date	Author(s)	Description
1.0	2017-08-04	BN	Initial public version

1. Introduction

Accelerating the time-to-solution for analysis of radio interferometry data is desirable for a number of reasons, e.g.:

- More efficient use of scientist/analyst time when the analysis is not fully automated
- More efficient use of valuable computing resources, e.g., very fast storage
- More timely response to time-critical phenomena, whether they are of astronomical origin (i.e., transient sources) or in the telescope (e.g., a fault developing which is subtly corrupting the data)

The speed of interferometric data processing can be limited by a variety of computing resources, including at least:

- Storage input/output throughput
- Working memories capacities, latencies and throughputs
- CPU instruction throughput and latencies

The balance of how these factors determine the speed of processing depends on the configuration of the telescope, type of observation and processing required. The balance is predictable in

principle, but not easily so, and this balance has very substantially changed in the past with the evolution of computer hardware architectures.

For this reason, the way to accelerate the time-to-solution for a general-purpose radio astronomy problem is not obvious. My subjective experience is that improvements made view a view of a particular use case tend to have mixed, but on average a positive, success in the general.

In this memo I investigate distributing CASA[1] processing by combining it with the SWIFT/T[2] system described by Wozniak et al. (2013), with the ultimate objective accelerating time to solution.

SWIFT/T consists of a programming language (SWIFT) and a *dataflow* engine (Turbine). The word dataflow is used here in the technical sense of of Johnston et al. (2004). The most relevant feature of SWIFT language is the use of strictly single assignment data structures (an idea originating, I believe, in the *I-Structures* of the *Id* programming language, see Arvind et al. 1989) which makes it easy for the SWIFT to infer the dataflow semantics of the program.

2. Objectives

The objectives of the work described here were:

1. Evaluate the usability of a SWIFT/T+CASA system for astronomical data reduction
2. Initial evaluation of scalability
3. Evaluate programmer time required to interface CASA into an execution framework as a Python module

3. Choice of CASA

There are three primary reasons for choosing CASA for this work:

1. It is probably the most widely used package for interferometric data analysis today & is the official data reduction package for both ALMA and JVLA
2. CASA is probably at the upper limit of complexity of packages likely to be encountered in astronomy. For example:
    a) It has more than 35 direct package dependencies (according to my count when building CASA with NIX, see Nikolic 2016)
    b) It is written in a mixture of C++ (1990s vintage through to modern), Fortran, Python, X86 Assembly, XML & XSLT, CMake, YACC grammar, SWIG, Shell, Perl and no doubt some other languages
    c) It calls executables as sub-processes and makes use of system services such as DBUS

[1] `https://casa.nrao.edu/`

[2] This is completely unrelated to (and predates I believe) Apple's SWIFT language

d) It is currently supported on Linux and OS/X but some of the code has previously supported a wider variety of systems

3. I have been analysing some HERA data with CASA, and it is likely CASA will be used for the commissioning phase of the SKA

4. Choice of SWIFT/T

The choice to use SWIFT/T was motivated by:

1. The SWIFT language appears a promising way to make writing dataflow programs accessible to users
2. Native capability for file-based inputs and outputs (as used by CASA)
3. Easy to run on a large-scale production cluster as a user-built application

5. Design & Implementation

The design is simply to use CASA as an in-process Python module which is invoked through SWIFT/T's existing mechanisms for Python functions. This approach:

- Significantly increases efficiency versus calling CASA as an external process because of the significant overheads of starting a new process and the CASA dynamic loading library loading, relocation, etc. In the present approach CASA dynamic libraries remain loaded and initialised throughout the lifetime of each worker.
- Reduces code complexity associated with marshalling parameters to the process command line

In line with the architecture of both CASA tasks and the SWIFT/T system, the primary data inputs and outputs are through files on a shared filesystem.

5.1. Building a combined SWIFT/T & CASA installation

The objective is for SWIFT/T to use CASA as an in-process Python module. For this to work correctly SWIFT/T must be built against the version of Python supplied with CASA (I used the NRAO pre-built CASA version 4.7.2). With a small patch (Appendix A.1) to the SWIFT/T Python include path detection this worked fine.

A slight complicating factor to be aware of is that CASA itself will execute some UNIX commands as sub-processes invoked by their absolute path[3]. This, for example, has an implication on the versions of *ld-linux* shared object interpreter that can be used by the various software components.

[3] An example is the *split* task which calls through to CASACORE, which invokes */bin/cp -r*

5.2. Exposing CASA Tasks as SWIFT/T functions

SWIFT/T is able to call Python code generated at run-time. The code is invoked through the python_persist [4] function. Exposing the CASA tasks to SWIFT/T consists of creating wrapper SWIFT/T functions which expose the data dependencies and parameters of CASA task at the SWIFT/T layer. Automatic generation of these wrappers from the CASA XML task descriptions should be possible but in the present instance a selection of tasks were wrapped by hand. The code for the wrapping is shown in Appendix B.

All non-file input and output to the Python tasks is marshalled through the input and output strings of the python_persist function. For CASA task-based processing the overheads are irrelevant since tasks always take input data as files and output data as files but in more general usage of Python this would likely be desirable to improve.

The file based inputs and outputs to the CASA tasks are handled and coordinated through the built-in SWIFT/T file type and associated mechanisms. A small patch (Appendix A.2) was necessary so that SWIFT/T correctly deletes directories which are used instead of files by CASA for input and output.

Wrapping of the image plotting task viewer needed a small adjustment to CASA due to the fact that as distributed CASA will keep a dangling xvfb process. This would lead to zombie child processes of the SWIFT/T worker preventing correct termination of the job. The trivial patch is shown in A.3.

5.3. A sample SWIFT/T+CASA application

The final part of implementation is an example application to drive the SWIFT/T+CASA system to produce useful outputs. For this initial test I selected a relatively simple application which I am using to check the stability of the HERA telescope over time. The application consists of deriving the antenna delays and gains solutions from one short observation when the galactic centre was transiting (see Nikolic et al., 2016, for details), and then imaging individually a whole set of observations with these calibration solutions applied in order to visually inspect the quality of the imaging. The initial calibration is not task-parallel but the remaining processing is entirely parallel without further interactions.

The source code for this application is 62 lines long and is shown in Listing1.

Listing 1: Example application that calculates delay and gain calibrations from on observations, and using these calibrations images a whole set of observations outputing plots of the cleaned images of each observation as the final pipeline product.

```
1 /* -*- mode: c; -*- */
2
3 import io;
4 import files;
5 import python;
6 import unix;
7 import swiftcasa;
```

[4] The persistent version is always necessary as CASA loads numpy which can not be unloaded cleanly from the Python interpreter

```
8
9
10 (file o) stdflag(file i)
11 {
12   file myms2 = casa_flagdata(i, "manual", "82");
13   file myms3 = casa_flagdata(myms2, "manual", spw="0:0~65");
14   file myms4 = casa_flagdata(myms3, "manual", spw="0:377~387");
15   file myms5 = casa_flagdata(myms4, "manual", spw="0:850~854");
16   file myms6 = casa_flagdata(myms5, "manual", spw="0:930~1024");
17   file myms7 = casa_flagdata(myms6, "manual", spw="0:831");
18   file myms8 = casa_flagdata(myms7,  "manual", spw="0:769");
19   file myms9  = casa_flagdata(myms8, "manual", spw="0:511");
20   file myms10 = casa_flagdata(myms9, "manual", spw="0:913");
21   file myms11 = casa_flagdata(myms10, "manual", autocorr=true);
22   o= myms11;
23 }
24
25 (file caltabs[]) gccalibrate(file i)
26 {
27   file m1=casa_importuvfits(i);
28   file m2=stdflag(m1);
29   file model= mkinitmodel("J2000␣17h45m40.0409s␣-29d0m28.118s",
       1.0);
30
31   file m3 = casa_ft(m2, model, usescratch=true);
32   file c1  = casa_gaincal(m3, [], gaintype="K", solint="inf",
       refant="10",
33                          minsnr=1, spw="0:100~130,0:400~600",
                              calmode="ap");
34   file c2  = casa_gaincal(m3, [c1], gaintype="G", solint="inf",
       refant="10",
35                          minsnr=2, calmode="p");
36   caltabs=[c1,c2];
37 }
38
39 /* Standard Galactic Centre Processing */
40 (file oimgdir) image(file i, file caltabs[])
41 {
42   trace("Processing" , filename(i));
43   file m1=casa_importuvfits(i);
44   file m2=stdflag(m1);
45   file m4 = casa_applycal(m2, caltabs);
46
47   file m5= casa_split(m4);
48   file i1 = casa_clean(m5, 500, [512, 512],
49                        "250arcsec", "0:150~900",
50                        "ellipse[[17h45m00.0s,-29d00m00.00s␣],␣[␣11
                             deg,␣4deg␣]␣,␣30deg]");
51   oimgdir=i1;
```

```
52
53 }
54
55
56 file c[] = gccalibrate(input_file("/fast/tmpswift/inputs/zen
      .2457545.48011.xx.HH.uvcU.fits"));
57 infs = glob("/fast/tmpswift/inputs/*.uvcU.fits");
58 foreach infile,i in infs
59 {
60   file o =image(infile, c);
61   file oo <"plot%i.png" % i > = casa_viewer(o, ".image");
62 }
```

6. Results

6.1. Usability

This is inevitably largely subjective, but my observations regarding usability are as follows:

1. The structure and form of the SWIFT/T application program (shown in Listing1) is well matched to the structure of CASA tasks. The SWIFT/T program better follows the logical structure of the CASA tasks than the native Python API, where the fact that primary inputs and outputs are files on disk is not obvious in the structure of the program.
2. The SWIFT/T compiler produces error message inadequate for a typical audience, i.e., users not accustomed to writing syntactically correct programs but rather iterating toward them through the error messages. (The error messages are fine for people with a general experience in software engineering.)
3. It is easy to write modular programs and very rapidly change overall data processing pipeline in the program.
4. Execution on a large production cluster through a batch job management system is just as easy and single-node command line mode execution

6.2. Scalability

Simple initial tests of scalability were performed using the simple example application in Listing1. Only 23 input datasets were used, limiting available parallelism. Scalability was investigated by varying the number of SWIFT/T workers on both a single node and a cluster

On the single node tests, the program /usr/bin/time was used to record the wall clock time as well other parameters, including the total number of pages read and written to disk. On the Darwin cluster the total MPI execution time reported by the Turbine PBS script was used as the wall-clock run time. The results of simple initial scalability tests are shown in Figure 1.

In the single node tests, the /usr/bin/time program reported total file system input was about 33 GigaBytes (GB) read and 15 GB written. For the fastest execution this corresponds to an

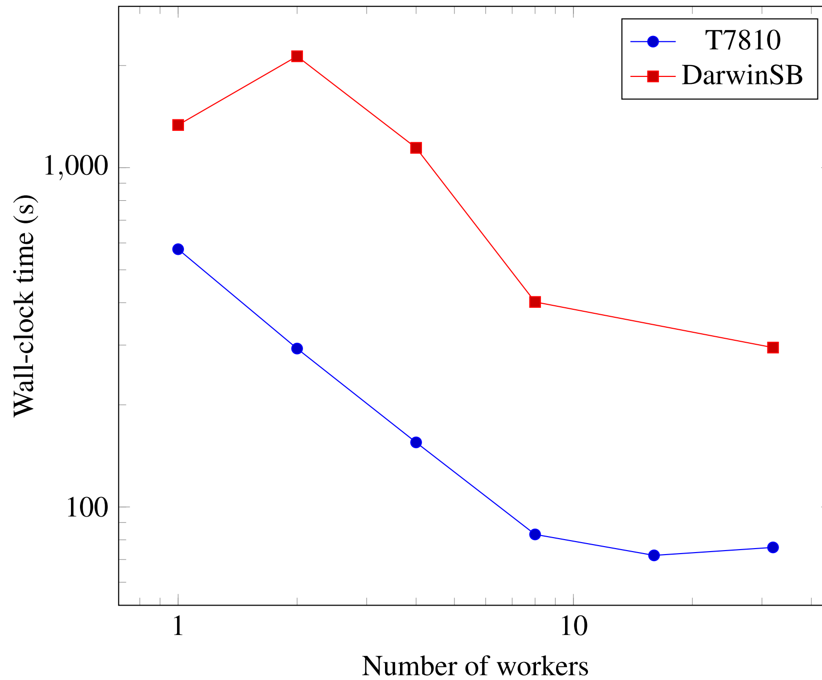


Figure 1: Scalability tests on a small data-set consisting of 23 HERA-19 measurement sets, each with a single phase centres and 10 minutes in duration. T7810: Single Node Dual Socket Xeon E5-2680 v3 with 64 GB ECC RAM and all input and intermediate data on an Intel P3700 NVMe storage PCI-connected storage running ZFS. NB: In all cases one ADLIB server was used, so total number of processes was one higher than the worker number. DarwinSB: University of Cambridge's Darwin system, Sandy Bridge architecture nodes, 4 processes per node, 128 GB memory per node, input data on Research Data Store Lustre file-system, intermediate and output data on scratch Lustre filesystem.

average read rate of 460 MB/s and simultaneous *average* write rate of 200 MB/s. Although the P3700 is capable in ideal conditions of even higher read speeds (up to 2,800 MB/s is claimed in the sales literature), these very high achieved I/O figures suggest that the plateau in scalability at around 8 workers is due to throughput limitations of the I/O subsystem.

Run times on the Darwin cluster were, in most cases, substantially longer than on the single node. Running CASA interactively showed similar performance. I have not determined the reason for this; it seems most likely factor is the performance of the file systems used for the type of I/O access that CASA does.

6.3. Programmer time

I have significant experience with CASA and some experience of SWIFT/T, and some experience of calling CASA from SWIFT/T as an external application. This experience set is obviously not at all typical. Nevertheless it is worth documenting the time taken for this work as a potential guide for prototyping of integration of CASA with other execution environments.

A total of about 8 days of work spread over two weeks was used for this work, approximately broken down as follows:

2 days Compiling SWIFT/T against CASA Python on a single node, resolving all run-time paths etc. The main challenge was ensuring a compatible libc and linker was used for all of the components.

3 days Writing the wrappers to the CASA tasks and the example main application. Main challenge was the semantics of the *wait deep* statement, and dataflow semantics associated with the Python calls.

2 days Compilation on the Darwin cluster. A significant amount of time was spent installing NIX as a single-user non-root install, which in the end was not used for the compilation of the final SWIFT/T+CASA version.

1 day Scalability tests

1 day Writing this document

While this has produced what seems to be a practical system, it should be noted that very significantly more time would need to be invested to make a production-ready system that is well packaged, and has sufficient features, tests and documentation to be usable by a range of people.

Acknowledgements

I thank Justin Wozniak and Tim Amstrong for clarifying two documentation issues with SWIFT/T through GitHUB. This work was in part supported by the EC ASTERICS project.

A. Patches

A.1. Python

```
modified   turbine/code/configure.ac
@@ -641,9 +641,9 @@ then
         AC_MSG_ERROR([Unable to find given python executable in PATH])
     fi
     TURBINE_CODEDIR=`pwd`
-    PYTHON_INCLUDE_FLAGS=$(${PYTHON_EXE} ${TURBINE_CODEDIR}/scripts/python-config.py --include-flags) ||
     AC_MSG_ERROR([Failed to execute python-config.py --include-flags])
-    PYTHON_LIB_FLAGS=$(${PYTHON_EXE} ${TURBINE_CODEDIR}/scripts/python-config.py --lib-flags) ||
     AC_MSG_ERROR([Failed to execute python-config.py --lib-flags])
-    PYTHON_LIBDIR=$(${PYTHON_EXE} ${TURBINE_CODEDIR}/scripts/python-config.py --lib-dir) || AC_MSG_ERROR([
     Failed to execute python-config --lib-dir])
+    PYTHON_INCLUDE_FLAGS=-I$(${PYTHON_EXE} -m sysconfig | grep platinclude | perl -lne 'print $1 if /"(.+?)
     "/' ) || AC_MSG_ERROR([Failed to get Python Include directory])
+    PYTHON_LIB_FLAGS=-l$(${PYTHON_EXE} -c "import distutils.sysconfig; print distutils.sysconfig.
     get_config_vars().get('LDLIBRARY')"  | perl -lne  'print $1 if /lib(.+)\.so/' ) || AC_MSG_ERROR([Failed
     to get Python linker flags ])
+    PYTHON_LIBDIR=$(${PYTHON_EXE} -c "import distutils.sysconfig; print distutils.sysconfig.get_config_vars
     ().get('LDSHARED')" | perl -lne  'print $1 if /-L(.+)/' ) || AC_MSG_ERROR([Failed to get Python shared
     library directory])
     PYTHON_NAME=$(${PYTHON_EXE} ${TURBINE_CODEDIR}/scripts/python-config.py --lib-name) || AC_MSG_ERROR([
          Failed to execute python-config --lib-name])
     AC_MSG_RESULT([Python enabled])
     AC_MSG_RESULT([Using Python include flags: ${PYTHON_INCLUDE_FLAGS}])
```

A.2. SWIFT/T

```
modified   turbine/code/lib/files.tcl
@@ -502,7 +502,7 @@ namespace eval turbine {
         if [ info exists mktemp_files ] {
             foreach f $mktemp_files {
               # Note: this does not raise error if file not present
-              file delete $f
+              file delete -force -- $f
             }
         }
     }
```

[back]

A.3. CASA

```
*** /data/p/casa-release-4.7.2-el7/lib/casa/bin/Xvfb    2017-07-29 23:05:23.015949570 +0100
--- /data/p/casa-release-4.7.2-el7/lib/casa/bin/Xvfb~   2016-05-25 16:05:54.000000000 +0100
***************
*** 13,18 ****
  }

  die "could not find Xvfb binary..." unless $xvfb;
! exec($xvfb, "-terminate", @ARGV);


--- 13,18 ----
  }

  die "could not find Xvfb binary..." unless $xvfb;
! exec($xvfb, @ARGV);
```

B. Wrapper code for CASA tasks

```
/* -*- mode: c; -*- */
/* Main SWIFT module for supporting use of CASA */

/* Rules for binding
- Always python_persist (CASA loads numpy, which can not be re-initialised)
- wait [deep] on all file inputs to CASA task (SWIFT/T can not see inside)
 */


import io;
import python;
import unix;
```

```
import string;

app (file o) noop2(string x)
{
  "true" o x;
}

app (file o) cpr(file i)
{
  "cp" "-r" i o;
}

(string o) python_filelist(file i[])
{
  string x[];
  foreach v,dx in i
  {
    x[dx]="'"+filename(v)+"'";
  }
  o=sprintf("[%s]", join(x, ", "));
}


(file vis) casa_importuvfits(file uv)
{
  wait(uv) {
    vis=noop2(python_persist("import os; os.remove('%s'); import casa; casa.importuvfits('%s', '%s'); " %
                             (filename(vis), filename(uv), filename(vis))));
  }
}

(file ovis) casa_flagdata(file vis, string mode="", string antenna="", string spw="",
                          boolean autocorr=false)
{
  // vis has to be filled before calling the CASA task. STC does not seem
  // to pick this up
  wait(vis) {
    python_persist("import casa; casa.flagdata('%s', flagbackup=True, mode='%s', antenna='%s', spw='%s',
         autocorr=bool(%b)); " %
                         (filename(vis), mode, antenna, spw, autocorr))=>
    ovis=vis;
  }
}

(file ovis) casa_ft(file vis, file complist, boolean usescratch=true)
{
  wait(vis, complist) {
    python_persist("import casa; casa.ft('%s', complist='%s', usescratch=bool(%b));" %
                         (filename(vis), filename(complist), usescratch))=>
    ovis=vis;
  }
}

/* Basic gaincal
*/
(file ocal) casa_gaincal(file vis, file gaintable[],
                         string gaintype="", string solint="",
                         string refant="", float minsnr=1.0,
                         string spw="", string calmode="" )
{
  wait(vis) {
    wait deep (gaintable) {
      ocal=noop2(python_persist("""
import os; os.remove('%s');
import casa;
casa.gaincal('%s', caltable='%s',
             gaintable=%s,
             gaintype='%s', solint='%s', refant='%s',
             minsnr=%f, spw='%s', calmode='%s');
""" %
                   (filename(ocal),
                    filename(vis), filename(ocal),
                    python_filelist(gaintable),
                    gaintype, solint, refant,
                    minsnr, spw, calmode
                    )));
    }
  }
}

(file ovis) casa_applycal(file vis, file gaintable[])
{
  wait(vis) {
    wait deep (gaintable) {
```

```
      python_persist("""
import casa;
casa.applycal('%s',
             gaintable=%s);
""" %
                   (filename(vis),
                    python_filelist(gaintable))) =>
    ovis=vis;
    }
  }
}

(file ovis) casa_split(file vis, string datacolumn="corrected", string spw="")
{
  wait(vis) {
    ovis=noop2(python_persist("""
import os; os.remove('%s');
import casa;
casa.split('%s', '%s', datacolumn='%s', spw='%s');
""" % (filename(ovis), filename(vis),
       filename(ovis), datacolumn, spw)));
  }
}

/* CLEAN has multiple outputs (restored image, clean comps, etc) hence
   the output is to a directory
*/
(file oimgdir) casa_clean(file vis, int niter,
                          int imsize[],
                          string cell,
                          string spw,
                          string mask,
                          string weighting="briggs",
                          float robust=0,
                          string mode="mfs",
                          int nterms=1)
{
  wait(vis) {
  oimgdir=noop2(
python_persist("""
f='%s';
import os; import shutil;
if(os.path.exists(f)):
   if (os.path.isdir(f)):
      shutil.rmtree(f)
   else:
      os.remove(f)
os.mkdir(f)
import casa
casa.clean(vis='%s', imagename='%s/img', niter=%i,
           weighting='%s', robust=%f,
           imsize=[%i,%i],
           cell=['%s'],
           mode='%s',
           nterms=%i,
           spw='%s',
           mask='%s');
"""%
   (filename(oimgdir),
    filename(vis),
    filename(oimgdir),
    niter, weighting, robust, imsize[0], imsize[1],
    cell, mode, nterms, spw, mask)));

}}

(file oplot) casa_viewer(file img, string extn)
{
  wait (img) {
  oplot=noop2(
python_persist("""
f='%s';
import os; import shutil;
if(os.path.exists(f)):
   if (os.path.isdir(f)):
      shutil.rmtree(f)
   else:
      os.remove(f)
import casa
casa.viewer(infile='%s',
       outfile='%s',
       gui=False,
       outformat='png')
import task_viewer
```

```
task_viewer.ving.done()
import psutil
current_process = psutil.Process()
children = current_process.children(recursive=True)
for child in children:
    child.kill()
"""%
   (filename(oplot),
    filename(img)+"/img"+extn,
    filename(oplot)
    )));
  }
}

(string res) casa_vishead(file vis, string hdkey)
{
  wait(vis){
    res=python_persist("""
import casa;
r=casa.vishead(vis='%s',
       hdkey='%s',
       mode='get');
""" %
(filename(vis), hdkey), "repr(r[0]['r1'])");
  }
}


/* Create a component list with a single component, e.g., for use in
   initialising the calibration process
 */
(file omodel) mkinitmodel(string direction, float flux, string shape="point", string fluxunit="Jy")
{
  omodel=noop2(
  python_persist("""
f='%s';
import os; import shutil;
if(os.path.exists(f)):
   if (os.path.isdir(f)):
      shutil.rmtree(f)
   else:
      os.remove(f)
import casac;
cl=casac.casac.componentlist();
cl.addcomponent(flux=%f,
                fluxunit='%s',
                shape='%s',
                dir='%s')
cl.rename('%s')
cl.close()
""" %
                 (filename(omodel), flux, fluxunit, shape, direction, filename(omodel))));
}
```

References

Arvind, Nikhil, R. S., and Pingali, K. K.: I-structures: Data Structures for Parallel Computing, ACM Trans. Program. Lang. Syst., 11, 598–632, doi:10.1145/69558.69562, http://doi.acm.org/10.1145/69558.69562, 1989.

Johnston, W. M., Hanna, J. R. P., and Millar, R. J.: Advances in Dataflow Programming Languages, ACM Comput. Surv., 36, 1–34, doi:10.1145/1013208.1013209, http://doi.acm.org/10.1145/1013208.1013209, 2004.

Nikolic, B.: Using Purely Functional Build Systems for Software Development in Radio Astronomy, Tech. rep., University of Cambridge, http://www.mrao.cam.ac.uk/~bn204/publications/2016/2016-10-nix-casa.pdf, 2016.

Nikolic, B., Carilli, C., and Sims, P.: Calibrating HERA-19 with a Galactic Centre Observation, Tech. Rep. HERA Memo 22, HERA Project, http://reionization.org/wp-content/uploads/2013/03/HERAmemo22-GC_imaging_cal.pdf, 2016.

Wozniak, J. M., Armstrong, T. G., Maheshwari, K., Lusk, E. L., Katz, D. S., Wilde, M., and Foster, I. T.: Turbine: A Distributed-memory Dataflow Engine for High Performance Many-task Applications, Fundam. Inf., 128, 337–366, doi:10.3233/FI-2013-949, http://dx.doi.org/10.3233/FI-2013-949, 2013.