Horizontal Prototyping Interim Report

Document Number……SKA-TEL-SDP-0000085
Document Type……..……...REP
Revision……..C
Author……..……D. Terrett
Release Date……..……..…2016-03-24
Document Classification……..…… Unrestricted
Status……..…… Draft

Lead Author	Designation	Affiliation
David Terrett	ARCH.SWE Team Lead	RAL
Signature & Date:	**Signature:** David Terrett (Mar 23, 2016) **Email:** david.terrett@stfc.ac.uk	

Released by	Designation	Affiliation
Paul Alexander	SDP Project Lead	University of Cambridge
Signature & Date:	**Signature:** Paul Alexander (Mar 23, 2016) **Email:** pa@mrao.cam.ac.uk	

Version	**Date of Issue**	**Prepared by**	**Comments**
C	2016-03-24	David Terrett	

ORGANISATION DETAILS

Name	Science Data Processor Consortium

Table of Contents

Introduction

The role of the horizontal prototype is to confirm the correctness and completeness of the interface (both internal and external) of the SDP. It is not expected to demonstrate scalability to SKA size but should demonstrate data parallelism at the high level. It is not expected to be efficient or use novel algorithms. However, construction of the prototype proper cannot begin until the SDP architecture is defined in more detail than it currently is and the interfaces described in more detail. So, in the mean time, the ARCH.SWE group has been exploring the feasibility of using CASA to implement pipeline components and Swift/T for describing data-flow.

This memo describes the lessons learned from the horizontal prototyping work to date. Efforts so far have focused on the imaging pipeline and have used CASA exclusively for implementing pipeline components and mostly Swift/T for describing data-flow and providing an execution environment.

Working with CASA

Building CASA

The binary CASA distributions prepared by NRAO will run on pretty much any Intel Linux system but to make use of it on more exotic systems, or to find and fix bugs it is necessary to build it from source. The source is freely available and comes with CMAKE build scripts but the only systems that are “supported” by the CASA developers are RedHat releases 5 and 6[1] and it is necessary to install a considerable quantity of CASA specific software in system directories (and hence root access is required).

[1] Release 7 will probably be added sometime soon and release 5 dropped.

The build procedure is not well documented. The various sets of instructions that can be found on the web are not up-to-date and some are completely obsolete and misleading. However, although the build scripts build all the binaries and shareable libraries they do not generate the same set of files as a binary distribution.

The build system depends on the presence of many "3rd Party" libraries. Most of these are readily available on "normal" Linux systems and the few that aren't (e.g. pgplot, wcslib) are straight-forward to build and install. However, the lack of 3rd party libraries – or the requisite versions thereof - can be an issue on HPC systems where packages are often not up-to-date and/or not installed in the "usual" place. Although the build system is designed to cope with this, configuring it correctly is quite a laborious process and not always intuitive. It is also fairly fragile, updating the source code to the latest version can cause the build to break, sometimes in ways that are hard to diagnose and fix. The time taken to rebuild from scratch (several hours on a modest workstation) adds to the pain of troubleshooting build problems.

Architecture

CASA is organised into four parts:

- casacore contains libraries that implement a wide range of general purpose functions including the CASA tables and the measurement set. It is mostly C++ but there is some fortran.
- code implements the radio astronomy algorithms as a set of C++ classes and the GUI utilities for working with measurement sets.
- asap contains algorithms specific to single dish analysis.
- gcwrap implements the casapy python interface.

The "casacore" part is quite well commented and the Doxygen generated on-line documentation is reasonably complete. The same, unfortunately, cannot be said of "code". (We haven't looked at "asap").

The interfaces with which most CASA users are familiar are called "tasks". These are implemented as Python scripts each of which is typically several hundred lines of uncommented and poorly structured code which is mainly concerned with generating parameter defaults that depend on the values of other parameters. However, for a given set of parameters, the code can be reduced to a small (e.g. less than ten) number of calls to the Python to C++ interface layer. Deducing what calls are made by inspection is difficult but with a bit of Python trickery it is possible to log what happens when a task is run and then reproduce the calls in a simple Python script. This is how the scripts that implement the components in the Swift/T pipeline prototypes were created.

Data storage

CASA stores (almost) all data in what it calls "tables". These are organised as rows and columns, like a traditional database but additional information can be stored in "properties". Both

the entire table and individual columns can have properties and table properties can be links to other tables.

The physical representation of a table is normally a collection of files stored in a directory. One file contains the column definitions and the table properties; the data contained in the columns are stored in separate files, usually one file per column for the main data table. The reading and writing of the columns is implemented by software objects called “storage managers” and CASA contains several storage manager implementations optimised for different usage scenarios. It is possible to implement additional storage managers and have CASA load them as needed. This is not (as far as we know) documented anywhere but a couple of example storage managers have been written by SDP and it is not particularly difficult. When accessing an existing column, the correct storage manager is used automatically but if a new column is added to a table, the storage manager used is either defined by the applications code or defaults to the standard storage manager. We believe that the default can be changed but only by rebuilding CASA from source.

There is no requirement that a storage manager uses files to store the data. However, the use of a file system to store the index files and to manage the links to sub-tables is probably too deeply embedded in the code for it to be replaced by some other mechanism. Also, the IOCTL lock facility is used to prevent simultaneous write access to a table from more than one thread or process. We fear that this facility may introduce additional issues with a distributed-memory, multi-process pipeline, where the locking mechanisms of the underlying distributed filesystem might not be as robust as needed by the CASA code and envisaged by its developers.

Bugs

Running CASA on systems with many processors has uncovered two race conditions in the CASA core (both now fixed in the latest development distributions) and another, as yet undiagnosed, problem that causes the imageconcat function to crash. This suggests that using CASA in unconventional ways will require some investment in maintaining the CASA code itself.

ASKAPSoft

The Australian “SKA Pathfinder” – ASKAP – uses its own MPI-enabled C++ tasks to perform all aspects of its data analysis. This direct use of MPI allows ASKAPSoft to run efficiently on many existing HPC systems. It uses its own parameter and logging systems and has the advantage of using existing investments in C++ astronomical algorithms.

ASKAPSoft comes with its own slightly extended CMAKE system driven from Python. It uses Casacore and LOFAR as 3rd Party dependencies – in addition to, obviously, requiring the 3rd Party packages already mentioned in the context of building Casa itself. The ASKAPSoft build process is vastly better than that of Casa itself. All the ASKAPSoft dependencies can be automatically compiled from source using their appropriate software tools (not necessarily CMAKE).

It has proved possible to compile ASKAPSoft (with and without debugging) on various workstations and HPC systems and to compile it on the Hartree "Wonder" systems using both the standard Gcc and the Intel C++ compilers.

Swift/T

The "Swift" in Swift/T is a programming language[2] that is designed to facilitate writing parallel programs. It superficially resembles C so is accessible to programmers with standard skillsets but with the novel[3] feature that variables can only have a value assigned to them once. This takes a bit of getting used to but it is this feature that allows the compiler to deduce the parallelism inherent in a program.

The compiler does not expand the program into a static execution graph which means that:

1. A program can be parameterised by things like number of frequency channels and the values of such parameters supplied at run time facilitating scalability testing.

2. Flow control decisions are made at run time depending on data generated by computation. So, for example, an iterative loop could be terminated when a solution has converged rather than have a fixed number of iterations.

The language is small (compared to something like C) and is easy to learn, the only difficulty is in understanding how recursion is used to replace iteration. Swift has a "for" loop construct for iteration but it only covers relatively simple cases.

In the case of the horizontal prototype the "real work" is done by external programs launched by Swift; these are called leaf tasks. So far, the leaf tasks are all Python scripts that use the casac package but there is very little restriction on how a leaf task is implemented.

The execution of a Swift program is controlled by "Turbine" – a distributed, many-task dataflow engine. The node on which a leaf task is run can be controlled but this aspect has not yet been explored.

The most obvious deficiency, in the context of the SDP, is the lack of any error handling mechanism. If a leaf task fails, either crashing or exiting with some error condition, the entire program aborts. Moreover, a failed Swift program sometimes does not abort the cluster job itself, leading to a situation where a failure is detected by having the job aborted by a job scheduler due to exceeding the wall time execution limit.

[2] not the same as MacOS swift language.
[3] unless you have used a functional language

In addition to this issue there are worries about the extensibility and maintainability of Swift/T itself – in addition to its possible efficiency if it was used in a production environment. Also there are a lack of debugger tools to investigate its detailed run-time efficiency.

The Horizontal Prototype

The “prototype” horizontal prototype implements the data distribution stages of the imaging pipeline, including splitting the data over frequency and time domains and applying phase rotations for faceting except that the steps of FFTing and de-gridding the sky model has been replaced by a call to the imager task “ft” method. This is because CASA does not implement a user callable function for de-gridding[4].

A typical CASA function modifies a measurement set rather than creating a new one so each of the leaf functions makes a copy of its input data even in cases where this shouldn’t be necessary. For example, imager.ft opens the model measurement set for write access even though it does not actually write to it. This data copying, very likely, limits the throughput that can be achieved on an HPC like system.

One possible way of circumventing this problem is to use a copy-on-write file system so that only files (or parts of files) that are actually modified are copied. An experimental version of the pipeline that used the “unionfs” file system demonstrated that this is feasible and approximately halved the amount of temporary disk space used. However, this has only been run on a workstation because “fuse” (filesystem in user space) is not available on the HPC systems to which we have access.

Runs with simulated data on the Cambridge Darwin system have been successful with 64 frequency channels, 60 timeslots and 9 facets and therefore ~35000 potentially parallel tasks running on 512 cores. The job fails if the number of frequency channels is increased to 1024, probably because of the large number of temporary files being created.

Extending the Swift/T program to cover the entire continuum imaging pipeline is probably straight-forward but finding suitable implementations of the components may not be. There is not (as far as we know) a one-to-one correspondence between the all the components as currently defined and CASA functions accessible from python. Possible sources of component implementations are:

1. If the required functionality is implemented somewhere in the CASA libraries, write stand-alone programs that call the appropriate functions. This has already been done for some simple cases.
2. Use parts of the ASKAP software. ASKAPsoft is structured as a set of stand-alone programs that could be used in isolation. The feasibility of this has not been tested.
3. Use parts of the LOFAR software in a similar way.

[4] As far as I can tell

Given the current makeup of the ARCH.SWE group, all three demand significant input from PIP.IMG and 2 and 3 probably needs assistance from people familiar with the workings of the ASKAP and LOFAR software respectively.