SKA1 SDP Execution Frameworks Prototyping Report

Document Number..... SKA-TEL-SDP-0000117
Document Type..... REP
Revision..... 01
Authors.....V. Allan, B.Nikolic, M. Farreras, T. Cornwell, R. Lyon
Date..... 2018-10-31
Document Classification..... UNRESTRICTED
Status..... Released

Name	Designation	Affiliation	Signature	
Authored by:				
Verity Allan	SDP Configuration Manager	University of Cambridge	Verity Allan (Oct 24, 2018)	
			Date:	
Owned by:				
			Date:	
Approved by:				
			Date:	
Released by:				
Paul Alexander	SDP Project Lead	University of Cambridge	Paul Alexander (Oct 25, 2018)	
			Date:	

DOCUMENT HISTORY

Revision	Date Of Issue	Engineering Change Number	Comments
01	2018-10-31	-	Prepared as a report for SDP CDR

DOCUMENT SOFTWARE

	Package	Version	Filename
Word processor	Google Docs		SKA-TEL-SKO-0000000-01_GenDocTemplate
Block diagrams			
Google docs Add-ons	Cross Reference Table of contents		Used for figure & table numbering and references. Used for heading numbering.

ORGANISATION DETAILS

Name	SDP Consortium
Lead Organisation	The Chancellor, Masters and Scholars of the University of Cambridge The Old Schools Trinity Lane Cambridge CB1 ITN United Kingdom
Website	www.ska-sdp.org

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

N/A

LIST OF ABBREVIATIONS

API	Application Programming Interface
ARL	Algorithm Reference Library
CASA	Common Astronomy Software Applications

COTS	Commercial Off The Shelf
CSD3	Cambridge Service for Data Driven Discovery
CUDA	Nvidia-provided API for GPUs
DALiuGE	Data Activated Liu Graph Engine
EF	Execution Framework
GIL	Global Interpreter Lock
GPU	Graphics Processing Unit
HPC	High Performance Computing
ICAL	Iterative Self-Calibration pipeline
ICRAR	International Centre for Radio Astronomy Research
JNI	Java Native Interface
JVM	Java Virtual Machine
MPI	Message Passing Interface
PDR	Preliminary Design Review
SDP	Science Data Processor
SIP	SDP Integration Prototype
SKA	Square Kilometre Array
SWIG	Simplified Wrapper and Interface Generator
TIMG	A simple Imaging pipeline
TRL	Technology Readiness Level

1 Introduction

The Execution Framework has, from the moment we considered its inclusion in the SDP design, been a high priority for our prototyping work. This began as part of our "horizontal prototyping" work, and was focussed particularly on to Execution Frameworks after the Preliminary Design Review (PDR) and delta-PDR reviews indicated that the Execution Frameworks area contained considerable risks for the SKA.

The Execution Framework module is an important part of the SDP Architecture [AD01]. It provides the mechanism for executing pipelines at scale in the SDP, using data-parallel processing techniques. The architecture has been designed to allow the SDP to use different execution frameworks (suited to the pipeline being executed), while using common processing components. Some familiarity with this aspect of the SDP Architecture is assumed in this document.

This work has been conducted by multiple teams across the SDP Consortium. The Horizontal Prototyping team[1] worked on using Swift/T [RD11]; the ICRAR team[2] created DALiuGE, to develop a radio-astronomy specific data-driven framework. (A full description of the DALiuGe prototyping can be found in RD15, with results described in RD13, RD08. The COTS Execution Frameworks (COTS EF) prototyping team[3] was set up to reduce risks associated with the SDP architecture for Execution Frameworks, Processing Components and Workflows, and to verify architectural decisions. The SDP China team[4] focussed their work on Apache Spark [RD23], which was identified at delta-PDR as a leading candidate for prototyping [RD07]. Experiments with Dask were also carried out as part of the development of the Algorithm Reference Library (ARL) [RD14, RD16]. For a fuller understanding of our prototyping efforts, see the referenced memos and reports.

There are many existing pieces of software that can fit into the Execution Framework paradigm, from Apache Spark to Swift/T to Dask, as well as DALiuGE. The work is heavily connected to the ARL [RD18]; the COTS EF team have been using ARL components as the common processing components in their StarPU prototyping. This interface work is complementary to the SDP Integration Prototype (SIP)[RD17]; while the COTS EF team have been focussing on the interface between the Execution Framework and the Processing Components, the SIP team have been focusing on the control and queue interfaces, and have tested deploying workflows run by Apache Spark, MPI and Dask.

This report sets out why we investigated certain Execution Frameworks; it reports key lessons on what we found; reports on the risks reduced or analysed by this work; and presents suggestions for how to continue work on Execution Frameworks.

[1] Steven Pickles, David Terrett, Brian McIlwrath

[2] Andreas Wicenec, Chen Wu, Rodrigo Tobar, Markus Dolensky, V. Ogarko, David Pallot, Richard Dodson, Kevin Vinsen

[3] Bojan Nikolic, Fred Dulwich, Montse Farreras, Chris Hadjgergiou, Arjen Tamerus, Vlad Stolyarov, plus contributions from Tim Cornwell and Peter Wortmann.

[4] Feng Wang, Qiuhong Li, *et al*.

2 References

2.1 Applicable documents

The following documents are applicable to the extent stated herein. In the event of conflict between the contents of the applicable documents and this document, **the applicable documents** shall take precedence.

[AD1] SKA-TEL-SDP-0000013, SDP Architecture, Rev 06
[AD2] SKA-TEL-SDP-0000052, SDP Risk Register, Rev 08

2.2 Reference documents

The following documents are referenced in this document. In the event of conflict between the contents of the referenced documents and this document, **this document** shall take precedence.

Reference Number	Reference
[RD02]	SKA-TEL-SDP-0000054 Rev 03 SDP Prototyping Report Overview
[RD04]	Java Native Interface Specification https://docs.oracle.com/javase/8/docs/technotes/guides/jni/spec/jniTOC.html
[RD05]	SKA-TEL-SDP-0000130 SDP Memo 50: The Accelerator Support of Execution Framework http://ska-sdp.org/sites/default/files/attachments/sdp_memo-50_wf_signed.pdf
[RD06]	SKA-TEL-SDP-0000140 SDP Memo 034: Practical Distributed Data Processing using SWIFT/T & CASA http://ska-sdp.org/sites/default/files/attachments/2017-08-casaswift.pdf
[RD07]	SKA-TEL-SDP-0000072 Can SDP Use Existing Big Data Systems? http://ska-sdp.org/sites/default/files/attachments/ska-tel-sdp-0000072_01c_rep_sdpmemobigdatasystems_-_signed.pdf
[RD08]	SKA-TEL-SDP-0000174 SDP Memo 078: Scalability Testing using DALiuGE on Tianhe-2 and Pawsey http://ska-sdp.org/sites/default/files/attachments/sdp_memo_78_scalability_testing_using_daliuge_v1.0.pdf
[RD09]	SKA-TEL-SDP-0000082 SDP Memo: Data-Driven Architecture Prototyping Report http://ska-sdp.org/sites/default/files/attachments/ska-tel-sdp-0000082_c_rep_sdpmemodatadrivenprototyping_-_signed.pdf
[RD10]	SAK-TEL-SDP-0000083 SDP Memo Data Flow Prototyping Report http://ska-sdp.org/sites/default/files/attachments/ska-tel-sdp-0000083_c_rep_sdpmemodataflowprototyping_-_signed.pdf

[RD11]	SKA-TEL-SDP-0000085 SDP Memo: Horizontal Prototyping Report http://ska-sdp.org/sites/default/files/attachments/ska-tel-sdp-0000085_c_rep_sdpmemohorizontalprototyping_-_signed.pdf
[RD12]	SKA-TEL-SDP-0000084 SDP Memo: MeerKAT Report http://ska-sdp.org/sites/default/files/attachments/ska-tel-sdp-0000084_c_rep_sdpmemomeerkatdataflow_-_signed.pdf
[RD13]	SKA-TEL-SDP-0000159 SDP Memo 039: Full-Scale DALiuGE Data Simulation and Reduction on Tianhe-2 http://ska-sdp.org/sites/default/files/attachments/sdp_memo_-_daliuge_scalabilty_test_on_tianhe2.pdf
[RD14]	SKA-TEL-SDP-00000150 SKA1 SDP Algorithm Reference Library (ARL) Prototyping Report
[RD15]	SKA-TEL-SDP-0000153 SKA1 SDP DALiuGE prototyping report
[RD16]	SKA-TEL-SDP-0000177 SDP Memo 81 Combining Task-based Parallelism and Platform Services within a Science Pipeline Prototype
[RD17]	SKA-TEL-SDP-0000137 SKA1 SDP SIP prototyping report
[RD18]	ARL Code https://github.com/SKA-ScienceDataProcessor/algorithm-reference-library
[RD19]	SKA-TEL-SDP-XXXXXX Pipeline Working Sets and Communication http://ska-sdp.org/sites/default/files/attachments/pipeline-working-sets.pdf
[RD20]	StarPU http://starpu.gforge.inria.fr/
[RD21]	Protocol buffers https://developers.google.com/protocol-buffers/docs/overview
[RD22]	SKA-TEL-SDP-0000083 Evaluating Data Flow Execution Environments: Regent and Legion as an example
[RD23]	SKA-TEL-SDP-XXXXXX Evaluating and Investigating the Execution of an Astronomical Pipeline on Spark
[RD24]	http://matthewrocklin.com/blog/work/2017/07/03/scaling
[RD25]	A Processing Pipeline for High Volume Pulsar Data Streams, R.J. Lyon *et al*. https://arxiv.org/abs/1810.06012
[RD26]	https://cffi.readthedocs.io/
[RD27]	SKA-TEL-SDP-0000148 SDP Memo 065: Fast Implementation of SKA Algorithm Reference Library

3 Motivation and Scope

Execution Frameworks have different qualities and properties and no one existing framework yet meets all SDP requirements - for example, implementation of calibration within a graph-based representation is challenging in most frameworks.

Early goals:

- Prototype of COTS dataflow execution engine with COTS processing components
  - SWIFT/T used with CASA components on commodity Lustre cluster [RD11]
- Explore systems for running radio astronomy workflows in parallel
  - Legion/Regent - proposed as part of Industry consultation package, as its memory mapping interface provides explicit programmer control of data placement, hence allowing task assignment to processors to take into account data locality and memory requirements. Prototyping found that the programming environment was complex and not well documented, and prone to problems. Our technology evaluation at delta-PDR found that these technologies were unlikely to yield good results for the SDP [RD22].
  - Cloud Haskell
- Create a purpose-designed execution framework for radio astronomy
  - DALiuGE: DALiuGE - early work reported in [RD09]. Based on the version of our architecture submitted at PDR. The overall SDP architecture has diverged from the original concept; however, DALiuGE could be run as an Execution Framework using the SDP execution control mechanism, as described in [AD01]. The more detailed scope and evolution is described in [RD15]; only key results and findings will be summarised here.

After delta-PDR, we reviewed the goals of our Execution Frameworks, and reconsidered the Execution Frameworks we were targeting. Across the Consortium, we had:

- identified Spark as a candidate technology at delta-PDR;
- the opportunity to continue with exploring DALiuGE, including the potential to explore DALiuGE at large scale;
- identified Dask as a framework in the official SKA programming language;
- been recommended StarPU by industry partners as the leading solution currently in deployment;
- identified Apache Storm for real-time pipelines as a strongly supported streaming framework;
- identified MPI as the currently best-supported HPC solution.
- We also intended to investigate reusing MeerKAT data processing software [RD12]; however, IP issues meant that we were unable to conduct any prototyping.[5]
- created a new team to conduct prototyping in support of the architecture (the COTS EF team), which focussed on prototyping interfaces to processing components, in order to reduce risk.

Thus, the goals of the work to CDR were to:

- Explore the performance of existing frameworks:
  - Apache Spark
  - DALiuGE
  - Investigate accelerator support in Execution Frameworks[RD05]
  - Apache Storm for real-time pipelines [RD25]

[5] The MeerKAT data processing system has evolved since the description presented in RD12.

- Interface frameworks to the SDP Control System (SIP)
- Interface frameworks to radio astronomy processing components, looking for component reuse across frameworks, using the following frameworks:
    - Dask
    - StarPU
    - MPI

3.1 Architectural Impact

The key architectural decisions to verify were:

1. Verify that we can use task-based parallelism to quickly process SDP data
2. Verify that we can use the same components in multiple Execution Frameworks (using ARL components)

Task-based parallelism alone has not previously been used (as far as we are aware) to process astronomical data at scale. Either scripts have been used to iterate over a data set (e.g. frequency by frequency), or, if the workflow has been parallelised (as has been done in ASKAPSoft), a message passing MPI architecture has been used, relying on hard-scheduled and bulk-synchronous tasks. It is a critical part of our architecture that a task-based parallel execution engine (where some of the scheduling and data movement can be organised by the workflow) can be used for the vast majority of pipelines workflows; hence, it was a high priority to verify our understanding that task-based parallelism with some level of dynamic scheduling would be suitable for the SDP data processing problem.

Because earlier prototyping suggested that current Execution Frameworks did not have all the features that we need, we have not yet picked a single Execution Framework as the preferred candidate. Indeed, we may never do so - the needs of the streaming real-time workflows are very different from those of batch processing, so it is not unreasonable to expect to support two different Execution Engines for the two different use cases. However, reimplementing all the processing code for every supported Execution Framework would be a lot of work; hence we architected reusable processing components. The aim of the Execution Frameworks interface prototyping has been focused on demonstrating that it is possible to define a single set of components with associated interfaces that can be practically used from multiple execution engines without large additional software engineering effort.

3.1.1 Architectural Decisions

We needed a wrapper interface that was callable from a wide variety of programming languages (i.e. from the languages used by the Execution Framework). C is easily callable from C, C++, Java and Python, and hence is easily callable from Execution Frameworks using those languages.
The C API is callable from other languages, such as Haskell and Swift/T (which were explored in earlier prototyping efforts), as C is a very portable language. Thus using C as the wrapper code allows portability across many potential future execution frameworks.
It would also be possible to write all the components in C; however, it is not clear that this is required for all components. While some high-performance components may need to use C for the performance speed-ups, some components will be written to run on GPUs and hence use CUDA (or similar), and there may be utility code written in Python (which is the official programming language

of the SKA). Thus writing a C wrapper has advantages for calling processing components from other languages, and also does not force a decision about what language processing components should be written in. This decision should be reviewed in the light of lessons learnt (Section 4).

3.2 Risks addressed

Risks being addressed [AD02]:

- SDPRISK-390: Execution Framework is immature at production
  - Exposure was: Extreme; Residual Exposure is: Low
  - We will be using Set-based design in construction to reduce risk associated with a low TRL[6]
  - The DALiuGE, Dask, and Spark prototyping has given us more confidence in our architectural decisions in this area, and that there will be a solution available, even if it is not achieving optimal performance. [RD14, RD15, RD23]
- SDPRISK-361: Logical Graph to Physical Graph translation in the baseline SDP Architecture
  - Exposure: High, Residual Exposure is: None.
  - Mitigated by DALiuGE scaling tests [RD13, RD08], and by testing multiple Execution Frameworks, some of which use some level of dynamic scheduling.
- SDPRISK-343: Incomplete Interface Description between pipelines components & Execution Framework
  - Exposure was: Extreme; Residual Exposure is: Medium
  - This is the main risk that COTS EF prototyping worked to reduce. The ARL now provides good separation between pipelines components and workflow components; a route to providing a fully generic interface to workflow components in Python and C is proposed.
- SDPRISK-344: Data rate between nodes is underestimated
  - Exposure was: High; Residual Exposure is: Medium
  - The Execution Framework prototyping was not intended specifically to reduce this risk; however, we would expect to uncover any remaining high risks in this area in our prototyping once the prototype is advanced enough and we have conducted large-scale tests. Some modelling of this has been undertaken by other members of the consortium. [RD19].

4 Important Results

4.1 Main achievements

- The ARL was successfully used as a reference library, with components used in Dask and StarPU pipelines:

  The ARL is being used as a library of components, which can be used in different pipelines. This validates the architectural decision that we can use the same components in multiple execution frameworks, albeit so far only for a simple workflow. These components have been wrapped, as per our architecture, and as described below. (The components were used unwrapped in Dask, as Dask was able to access the processing components directly.) The ARL now provides “a coherent

[6] Set-Based Design “is a practice that keeps requirements and esign options flexible for as long as possible during the development process... SBD... explores multiple options, eliminating poorer choices over time.” https://www.scaledagileframework.com/set-based-design/

set of data models and application programming interfaces (APIs) for state-free functions"[RD-14]. This helps address SDPRISK-343.

- A C API to ARL was implemented for a number of components and functions in ARL:
  The processing components wrapper code was written in C and Python`/libcffi` [RD26].
  We have wrapped a simple pipeline (TIMG),[7] and we have run the TIMG pipeline in the C-based Execution Framework StarPU. This practical demonstration has substantially reduced the risks associated with the interface between pipelines components and Execution Frameworks. We have also wrapped some components used in part of the ICAL pipeline. This also helps address SDPRISK-343. The code is available in RD-18, in `algorithm-reference-library/ffiwrappers`.
- A Java interface to ARL was implement for a small subset of the ARL components and functions:
  - This was implemented through a SWIG auto-generated JNI wrapper which binds to the ARL C-api described above.
  - It was verified by a simple serial Java program.
  - It uses the C API previously defined to interface to ARL components, which are then called through the C wrappers by the JNI (Java Native Interface) [RD04]. The JNI allows the Java Virtual Machine (JVM) to call routines in other languages (in this case, C). We have tested this with a simple Java program. This should allow Java-based Execution Engines such as Apache Spark to call the C-wrapped ARL functions. Thus we have shown that the C bindings can be called from any reasonable Execution Framework, as expected. However, we have not done sufficient prototyping to uncover any issues with this. The code implementing this is in RD-18; in `algorithm-reference-library/ffiwrappers/java`. Again, this reduces SDPRISK-343.
- We have used ARL + Dask to run the ICAL pipeline, one of our most complex workflows [AD01, Imaging Workflow View, ICAL Workflow View]. This substantially reduces risks associated with running real astronomy workflows. [RD-14]
- DALiuGE has been run in an HPC environment with ~10 million tasks, whilst providing visualisation of progress. [RD08] This demonstrates scaling of DALiuGE in terms of overhead (>0.1ms for >0.25m drops). Its performance is comparable to MPI. This reduced SDPRISK-390.
- DALiuGE has successfully tested streaming data modes to support the Receive function of the Ingest pipeline [RD08].
- To look at supporting efficient processing components, the GPU support provided by different frameworks was investigated. It is possible to make DALiuGe and Spark support GPUs, but this is not natively available [RD05]. StarPU does provide native GPU support [RD20], so there are options available.
- Spark scaling tests were conducted on the ICAL pipeline. Initial memory issues were somewhat alleviated by using Alluxio to deal with the negative effects of Spark shuffles [RD-23]. PySpark tests in SIP incurred a large overhead because of the extra serialisation required to get the data into the JVM [RD-17].
- As a test of the streaming workflows, Rob Lyon and the Manchester team have a prototype of Pulsar Search via Data Stream Processing. Thus, we have successfully built and tested a prototype pulsar search pipeline using Apache Storm. This aimed to: i) establish the feasibility of processing pulsar data in real-time (or close to), ii) establish if there are any pipeline issues relevant for SDP, iii) produce optimised versions of pulsar search algorithms that could be deployed if necessary, iv) understand the computational requirements of a

[7] This is a very simple imaging pipeline, with a very simple data distribution.

minimal, yet viable pulsar search workflow deployed using COTS tools, and v) determine which areas of the search pipeline are trivially parallelisable and which are not. The overall results achieved by the prototype were promising. It obtained 81% pulsar recall, and an overall filtering accuracy of 99%. This has not yet been integrated with the SDP control mechanisms. However, this does show the use of a streaming framework for SDP is feasible. [RD25]

- Dask has been run at moderate scale (~137,000 tasks on 500 workers on a single node; 80GB working set for image size ~1GB) [RD-14]. Further tests before CDR closeout are planned.
- Dask graphs have also been created and then run in DALiuGE, showing the interoperability of the Python-based frameworks [RD-15].
- The SDP has worked with frameworks of acceptable maturity: both Spark and Dask, have acceptable or good communities surrounding them, thus reducing SDPRISK-390.

4.2 Difficulties

- Neither the current architecture documentation nor the way we explained the architecture was sufficiently clear and concise for members of the COTS EF team to understand enough to be able to work on the prototyping. We only made progress by a number of iterations of refactoring the ARL [RD14] which then demonstrated the module architecture to team members in a way they could understand.
  Two of the most difficult architectural concepts were: the distinction between workflows, wrappers and processing components; and: the granularity of processing components. Firstly, we had processing components that could modify the processing graph, by extending it. While that is compatible with Execution Frameworks such as Dask, it is not compatible with StarPU or Spark, which do not admit such modification of the graph. (StarPU, for instance, needs to know all dependencies of a task before running it, which means that you cannot modify or extend the graph during processing, as this would destroy its understanding of the dependencies.)
  There were then further issues with the granularity and level of processing components. In initial prototypes, we had processing components which could call other processing components, without modifying the graph. While that made them superficially more compatible with other Execution Frameworks, we found that these meta-components made it very easy to write a serial workflow, but very hard to expose the parallelism inherent in the problem, upon which we rely. This made it hard to then write the code in the Execution Framework, as it was not clear where the parallelism or data dependencies were to be found.
- There was a steep software engineering curve for new members of the COTS EF team, with some important technicalities only understood by a few members of the team (examples are the Python Global Interpreter Lock (GIL) and dynamic closures generated using `libcffi`).
- We encountered a number of defects in critical Python modules (`cffi` and SciPy) related to multi-threading, which required debugging by the technical lead as no other members had experience in Python-source-code level debugging. The first defect (in `cffi`) was present only in a specific range of versions which we used, illustrating the need for a strictly reproducible yet easily varied execution environment.
- We have significant issues dealing with the metadata for processing, owing to the differences between Python and C. Python has in-built support for highly dynamic typing and data organisation which does not exist in C. For these reasons, the COTS EF team resorted to using Serialization to pass some metadata between different processing components.
  For example:
  `dict` (a dictionary, a mappable object in Python) allows one to create a data object whose size can be altered during processing. In C, however, such objects can not be handled by

language-level constructs, but must be done through use of complex function calls and conventions that must be followed, making the source code unapproachable to inexperienced C programmers and making it hard to write error-free code
The current work-around in the APIs is difficult to maintain and unworkable in long term. It is possible that using protocol buffers may provide a better interface for serialising structured data, if we wish to continue to support using Python components in any Execution Framework [RD21].

- PySpark has limitations - one can't access the full functionality of Spark when using it, as it lags behind Scala (the native Spark language), so one advantage of Spark (the Python interface) is negated by the loss of functionality.
- StarPU requires writing C functions in a particular way, so that StarPU can recognise and manage the dependencies. When using the StarPU standard C API, all C functions that should run as tasks must adhere to a particular structure (in which your function may only be passed two parameters: an array with your parameters for the task, and a StarPU internal construct which contains control data including data dependencies. In addition, a codelet must be defined with the input and outputs of the function defined. The codelet in addition supports implementation of the same kernel for different architectures (i.e. CUDA or x86). Finally, one must define whether data is a copy, or whether it is a pointer, as this allows StarPU to automatically compute the data dependencies. None of this is immediately natural for a C programmer; if GCC plug-in support is available, StarPU defines a C extension to make it easier to write StarPU code, however we would be depending on a particular tool chain so we chose to prototype with the standard C API.
- Dask may have overhead issues - the scheduler seems to slow down at ~256 workers, which would not guarantee reaching the scale required for SKA-1, but which would provide adequate support for early Construction [RD24]. Dask may also not manage to meet the tasks/second requirements for the largest workflows in SKA-1. [RD-14]
- Early experiments with Spark had serious memory issues, which required a lot of effort to overcome. Even then, Spark was unable to reach 1% of SKA scale for the ICAL pipeline [RD23].
- Similarly, Dask had memory issues [RD16], though these could be partially mitigated by causing the graph to do more serial work on the same data. It is not clear whether the memory issues seen in Spark and Dask are specific to the Execution Frameworks in question, or a more general problem for Execution Frameworks.
- In most cases, extra effort was required to prepare programs for use in an HPC environment. This can be seen in the early CASA+Swift/T issues [RD11], running Dask on HPC [RD-14], and the modifications required to run DALiuGE on Tianhe-2 [RD-13].
- The SWIFT/T prototyping shows that it is possible to have a modular, easy to read, language with pure dataflow semantics which looks familiar even to astronomers [RD-06].
- Most current COTS execution frameworks are unable to deal with repeatable failures of tasks and often they can not deal with even intermittent failures.

5 Lessons Learned

5.1 Metadata structures are a key part of the design

This finding emerged from the COTS EF prototyping; it was not an issue that we had considered when architecting the Execution Frameworks and Processing Components. However, setting stable metadata constructs (which contain stable data objects - no `dict`s) for data objects allows us to

move between languages with and without explicit memory management, which allows us to use multiple Execution Frameworks. However, this is not a natural mode of programming in Python.

5.2 Performance risks not completely reduced

Initially, the SDP were concerned about the performance of an Execution Framework. However, ICRAR and the SDP China group have conducted high performance tests of Spark and DALiuGE, and there have been more limited tests of Dask using the ARL. While high-performance processing components have been tested, they have not been tested in Execution Frameworks [RD27]. We would still like to do more high performance prototyping, to explore the limits of currently available Execution Frameworks, and the overheads imposed by wrapped components, either before CDR closeout, or during Bridging.

Pulsar search best done with batch processing

Pulsar search is best undertaken using batch processing where possible, as it is simpler to implement and for pulsar search, this permits fundamentally higher levels of filtering accuracy [RD-25].

5.3 Cross-disciplinary teams are essential

As we saw in our Swift/T prototyping ([RD06][RD11]), execution frameworks prototyping relies heavily on having cross-disciplinary teams, covering computational radio astronomy and high performance parallel computing. In particular, having radio astronomers who understand the key algorithms used in processing, and how/whether they can be parallelised is essential to make substantial progress with Execution Frameworks coding. HPC experts are also helpful, as running workflows on HPC systems is also a considerable coding effort. The SAFe process that will be used in SKA Construction should allow the creation of appropriate teams, and thus ensure that progress is not blocked by lack of appropriate expertise.

5.4 It is hard to communicate the architecture

One of the persistent issues we uncovered during prototyping was that people were frequently misunderstanding the architecture and finding it difficult to apply the architecture to the programming problem in front of them. It tended to require frequent small group meetings to communicate the essential concepts. This also led to the issues noted above with defining what exactly a processing component is, and hence where the interface to that component is defined. This has obvious implications for getting new staff on the project up to speed - we estimate that it will be around 6 months before most people will be capable of making a substantial contribution to the Execution Frameworks area, although the example code we now have should help a little with this. There is also now more comprehensive architecture documentation, with better coverage of this area. However, we still note that this difficulty should be factored in to SKA planning in construction and operations.

5.5 Risks Associated with the component interface are substantially reduced

By defining a C API to the ARL components, and using that C interface to run ARL components in another Execution Framework, the risk associated with the interface has been substantially reduced. We have shown that this is architecturally feasible for a simple workflow, and have a working prototype. However, we have also shown that agreeing the level at which that interface operates is

very important. In order to support composition by the workflows, processing components must be very low level; any composition must happen at the workflow level. Having a working prototype should make understanding this easier in future.

5.6 Risks Associated with the control interface are substantially reduced

This was demonstrated by the work undertaken by SIP, using P3-AlaSKA [RD17].

6 Suggestions/Recommendations

- Proceed with Python-based frameworks.
    - This allows use of C or Python pipeline components, and makes the metadata issues easier.
    - Python is the official SKA programming language, so this may make staff recruitment and code maintenance easier
    - The frameworks tests so far show acceptable performance; if processing components become the bottleneck, we can investigate replacing parts of the ARL with Numba code. [RD-14]
    - However, this leaves it unclear what to do about real-time processing (for Receive, Real-Time Calibration and Real-Time Imaging for Slow Transients, all of which have stringent performance requirements described in AD01), as we have not conducted much testing in these areas.
    - We recommend that large scale tests be carried out with e.g. CSD3
- We believe we can proceed with any or all of DALiuGE, MPI, and Dask. There are pros and cons to all of these:
    - DALiuGE - has been shown at scale, but needs to be entirely maintained by SKA. Real-time processing has not been investigated beyond testing the Receive portion of the Ingest pipeline.
    - Dask - has wider adoption, but has some scaling issues [rd24], and again, streaming has not been investigated.
    - MPI - `mpi4py` has very wide adoption, but it is a lot of effort to write pipelines as all the dependencies/communications have to be managed by hand.
- In order to allow for using any Execution Framework with Python components (e.g.from the ARL), work will need to continue on the Processing Component interface, looking at Protocol Buffers.