SDP Memo 017: Dataflow Choice

Document number……………………………...………………………..SKA-TEL-SDP-0000066
Context……………………………………..……………………………………………..…ARCH
Revision………………………………………………………………………………………………1
Author……………………………………..……………………………………….M. Dolensky
Release Date………………………………………………………………………..…. 2015-05-29
Document Classification…………………………………………………………..……. Unrestricted
Status………………………………………………………………………………………………. Draft

Name	Designation	Affiliation
Signature & Date:		

Version	**Date of Issue**	**Prepared by**	**Comments**
1	2015-05-29	M. Dolensky	PDR close-out action

ORGANISATION DETAILS

Name	Science Data Processor Consortium

1 Table of Contents

2 List of Figures

3 List of Tables

4 Introduction

The action MC1-133 in the PDR Report [AD1] reads as follows:

"SDP is requested to produce a brief comparative study of dataflow versus the obvious alternatives and to have this material inserted in the next revision of the PDR data pack."

With the data rate being the dominant system parameter the SDP software architecture is focused on this very aspect. Consequently, it has a data layer which is a middleware supporting data-driven coarse-grained workflows and provides a clear-cut place where COTS and customized industry solutions can be incorporated. After briefly touching on the basics of the dataflow approach, it gives a synopsis of the scalability analysis of precursors systems comparable in scope to SDP and then discusses the merits of the chosen dataflow approach in comparison to prevalent high performance compute architectures.

5 References

5.1 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.

Reference Number	**Reference**
AD1	SDP Preliminary Design Review Panel Report rev. 1, SKA-TEL-SKO-00000169
AD2	SDP Glossary, SKA-TEL-SDP-0000056

5.2 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**
RD01	PDR.01 SDP Architecture, SKA-TEL-SDP-0000013
RD02	PDR.01.02 SDP Dataflow Environment, SKA-TEL-SDP-0000015
RD03	PDR02.02 Data Sub-Element Design Report, SKA-TEL-SDP-0000023
RD04	PDR02.02.01 Data Challenge Supplement, SKA-TEL-SDP-0000024
RD05	SDP PDR Guidance, SKA-TEL-SDP-0000059
RD06	PDR13, Risk Register, SKA-TEL-SDP-0000006
RD07	PDR14.01, Prototyping Plan, SKA-TEL-SDP-0000054
RD08	ASKAPsoft Overview and Thoughts on Scaling, SKA-TEL-SDP-MEMO-009
RD09	LOFAR Software overview and scaling, SKA-TEL-SDP-MEMO-010
RD10	LMC Study: Going from MWA to SKA, SKA-TEL-SDP-MEMO-011
RD11	MPI: A Message-Passing Interface Standard v3.0, http://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf
RD12	MapReduce: Simplified Data Processing on Large Clusters, Jeffrey Dean and Sanjay Ghemawat
RD13	The Fourth Paradigm, Data-Intensive Scientific Discovery, Microsoft Research, ISBN 978-0-98225442-0-4

6 Existing Dataflow Architectures

This section is a summary of prevalent dataflow approaches, memory models and programming paradigms.

6.1 Dataflow Approaches

A dataflow architecture means that the program execution is determined based on the availability of input arguments to the instructions (i.e. Processing Components). There is no static control flow at the level of programming code.

A static dataflow approach means that the dataflow graph is a collection of activity templates for inputs, outputs and programs. It is simple and loops are executed sequentially. In contrast, with the dynamic approach each loop iteration or subprogram can be invoked in parallel. There are further models - but only one is mentioned here - the explicit token store approach which does efficient matching.

6.2 Memory Models

We consider the three basic memory models:

1. distributed: each processor has its local memory space; comms via network
2. shared: processors (e.g. CPUs on a board) sharing memory
3. hybrid: interconnected nodes with shared memory

For the sake of brevity the three models are introduced together with their de facto industry standard product, that is MPI, OpenMP and their combination.

6.2.1 Distributed Memory: MPI Cluster

Distributed memory refers to a multiple-processor system in which each processor has its own private memory and computations operate on local data. Other data need to be communicated from remote processors.

MPI is a portable communication protocol [RD11] with defined language bindings in Fortran and C and implementations in several other programming languages and from various vendors. MPI provides communication and synchronisation between multiple processes. Typically, a job scheduler (e.g. SLURM) is employed which in turn uses MPI to start a single process on each allocated CPU core at time. It supports several logical process topologies and collective communication (broadcast, scatter, gather, etc.).

6.2.2 Shared Memory: OpenMP

Shared memory is a multiprocessing design allowing several processors global memory access. It can be implemented in hardware and software (inter-process communication). OpenMP is a de facto software standard for parallel programming on shared memory systems and comes in the form of libraries for various platforms and requires compiler support. Application code includes compiler directives to parallelize algorithms.

6.2.3 Hybrid Architectures

MPI and OpenMP can be used in combination to support hybrids consisting of interconnected shared memory systems. Some hardware architectures include accelerators such as GPUs which come with their own programming interface (e.g. CUDA). Exploiting the potential of hybrid hardware and software architectures adds considerable complexity to application code.

6.3 Programming Paradigms

6.3.1 MapReduce

MapReduce [RD12] is a programming paradigm for processing large data sets with a parallel, distributed algorithm on a cluster. MapReduce libraries exist in many programming languages including open source implementations. MapReduce is the processing part of Hadoop (the other part being storage).

In a nutshell, the idea is to organize data in {key_1, value_1} pairs and then to sort and group (i.e. partition) the tuples such that each partition can be processed independent of each other and therefore in parallel. Following this *Map()* function the data are processed and a result set consisting of {key_2, value_2} tuples is generated on each compute node. The *Reduce()* function aggregates the result sets. *Map()* and *Reduce()* functions are user provided. In short:

$$map(k_1, v_1) \rightarrow list(k_2, v_2)$$
$$reduce(k_2, list(v_2)) \rightarrow list(v_2)$$

6.3.2 Dataflow Programming

Dataflow programming is a paradigm that models a program as a directed graph (i.e. Physical Deployment Graph provided by LMC) of the data flowing between operations. [RD02] gives a detailed introduction plus references and discusses the potential of domain specific languages.

6.3.3 Data-driven Programming

Data-driven programming is a paradigm in which the code describes the data to be matched as well as the required processing. This is very similar to event-driven programming. (The SDP data layer triggers actions when a datum changes state.)

7 The Problem

SDPCMT-428: *“Define the scale of the problem faced in SDP and the (without question) need for scalability in the implementation. The document should then comment on the scalability of the existing solutions and explain how (for example the frequency of data throughput) this has led to SDP pursuing a data parallel approach (i.e. it should record the steps undertaken).”*

7.1 Scale of the Problem

The fundamental challenge of the SDP is the data rate from CSP. The raw visibility data (excluding metadata) arrive at a rate of 7.3 TByte/s for SKA1-low, 3.3 TByte/s for SKA1-mid and 4.6 TByte/s for SKA1-survey [RD05]. These numbers are not cast in stone and the re-baselining has already changed them. For two reasons it has no bearing on this document though: Firstly, the PDR considers the configuration prior to rebaselining. Secondly, and more importantly, the software architecture was done with the aim of scalability to the full SKA, essentially leaving the timing and phasing of its implementation open.

[RD01] estimates that the dataflow system needs to cope with 500 million tasks when assuming an average chunk size of 10 GB and 50 stages for processing a dataset of 100 PB. This corresponds to about 50000 dataflow actions/s.

The data layer has to support two fundamentally different scenarios. For one, let’s label it the imaging pipeline case, there is a frequency partitioned parallel workflow with some challenges to minimize re-ordering and aggregations along the processing flow. And then there is, let’s call it the transient pipeline case, which requires a low latency streaming workflow, albeit with a much smaller data rate. The architecture embraces both cases, whereby the imaging pipeline is clearly the main design driver and the requirement for the transient pipeline came known only a couple of months ago.

7.2 Tackling the Problem

The architectural approach outlined in the PDR Executive Summary, p. 6 [RD05] is the result of an intense collaborative effort of the SDP architecture group involving all task leaders with design responsibility for operational software elements and the SDP management team.

Technical documentation of (at least) the following projects was considered during the design process: ALMA, ASKAP, LOFAR, LSST, LHC, MegaPipe, MWA, Pan-STARRS and Sloan (more in section 7.3). A number of potential bottlenecks such as global file systems and limited compute efficiency were identified and quantified [RDnn].

The architectural principles described in the system architecture [RD01] formed the basis for the subelement design reports. Risk items were identified [RD06] and the goals in the Prototyping Plan [RD07] (e.g. 5.5 Data Flow Management System) were set.

7.3 Precursor Solutions

For the precursors ASKAP, LOFAR, MWA a scalability analysis of their respective SDP equivalents was performed and documented in memos [RD08-RD10] as part of the rebaselining input, milestone M7. Here is a synopsis.

7.3.1 ASKAP

[RD08] describes the experience on scaling the MPI based ASKAPsoft Central Processor. In short:

ASKAPsoft uses an MPI-everywhere model for parallelism. Due to third-party libraries such as casacore and wcslib not being thread-safe shared-memory parallelism cannot be fully exploited. Furthermore, using OpenMP for shared memory parallelism scales very much sub-linear and plateaues at only a few cores due to the synchronisation overhead. For the same reason an MPI trial implementation of the deconvolution step does not scale.

Visibilities are stored on a single global Lustre filesystem leading to the usual benefit of simplified management at the cost of creating a bottleneck. Visibility I/O is limited by a combination of the Lustre metadata server being a bottleneck, inability to align Measurement Set tile I/O to Lustre stripes, CASA table caching behavior and lack of parallel MPI I/O.

A newly developed prototype parallel/MPI-IO FITS writer allows all MPI processes to write to a single file in parallel, thereby working around the CASA image aggregation bottleneck.

7.3.2 LOFAR

The scaling analysis of the CEP component of the LOFAR software system is described in [RD09]. Similar to SDP it performs the processing downstream of the correlator. It achieves parallelisation by subband partitioning. OpenMP is used to distribute the load to available cores on a single node thereby balancing varying processing times of individual data chunks.

The lack of thread-safety[1] limits scalability of some functions, for instance, continuum imaging with the FFTW library. Other current system limitations include shared file access on head nodes and sensitivity to load balancing and (central) master logging.

7.3.3 MWA

[RD10] discusses MWA and is focused on LMC scalability because the processing starting from raw visibilities (400 MB/s) happens off-site. The current system is centralized and can cope without a comprehensive message passing middleware layer to coordinate distributed/parallel tasks. Centrally collecting, reordering and processing telemetry happens at a cadence of 1 s and doesn’t scale well, especially for metadata produced from the transforming data stream: For example, the creation of cross-correlation power metric plots for all 8128 baselines based on 1 s data dumps (400 MB) goes with $O(N_{bl}^2)$ and therefore is on an operator’s request only. The analysis states the need for a dedicated network for a decentralized LMC and its inherent timing constraints.

8 How the SDP Solution Compares

Increasing compute intensity in scientific computing and a scale-out architecture as a solution are not only Gray’s first and second law [RD13] but at the heart of the SDP architecture. The term CyberBrick coined in that context is analog to the Compute Island with its local storage space. The third law hardly needs any mentioning, it is the mantra of bringing computation to the data and to concentrate on data locality.

[1] Update 05/2015: Casacore is fully thread-safe w.r.t. the statics being used. In general, access to the objects (e.g. an Array object) is not thread-safe because of the penalty involved. Where needed the LOFAR code handles the locking.

Before concluding with a short discussion there are some textbook arguments about the merits of the various approaches. The bullet items inlcude pros (+), cons (-) and it depends (~) for each of MPI, OpenMP, MapReduce and Dataflow Approach:

8.1 MPI

+ runs on either shared or distributed memory architectures
+ can be used on a wider range of problems than OpenMP
+ each process has its own local variables
+ distributed memory computers are less expensive than large shared memory computers
- no fault tolerance
- when combined with shared file system not compatible with architecture
- no data placement awareness
- demanding programming changes to go from serial to parallel version
- can be harder to debug
- performance is limited by the communication network between the nodes

8.2 OpenMP

+ easier to program and debug than MPI
+ incremental, gradual parallelization possible
+ program is still functional in serial code
+ serial code statements can usually stay unmodified
- code easier to read and therefore maintain
- requires shared memory computers
- requires compiler support
- mostly used for loop parallelization

8.3 MapReduce

(with Hadoop/HDFS)

+ fault tolerance
+ highly scalable
- high latency; batch system
- single fixed dataflow; single input/output; not suited for loops
- not suitable for stream processing
- not suitable for in-memory processing
- not suitable for all algorithms

- I/O for complex data structures requires system tweaking
- some native optimizations not available through Java

8.4 Dataflow Approach

+ control of data locality
+ exploitation of data parallelism
+ works with commodity hardware
+ no global file system needed just a GUID server like CEPH
+ control flow of sub-graph not limited by a master node
+ load-balancing
~ fault tolerance: depends on underlying storage manager
- idle times when waiting for matching data

8.5 Concluding Discussion

MPI communication constitutes a serious performance bottleneck (see example of ASKAPSoft). Shared systems and hybrid systems add considerable software complexity. In the timescale of SKA1 hardware solutions (such as SGI UV2000) combined with programming languages like UPC may actually scale up to Compute Island scale and make for a viable competitor to clusters of commodity hardware. MapReduce based systems require additional layers of software (HADOOP/HDFS, Apache Spark etc.) to offset some of their inherent drawbacks.
Fault-tolerance comes at the price of a lot of hardware redundancy, which is problematic within the given limited power budget.

In contrast to all of above, the SDP architecture prescribes a system that maximises throughput by optimising data locality on a distributed system. The proposed data layer is not comprised of a single system in terms of an existing product, but requires the combination of existing products. This literally forced the architecture group to look for alternatives to more classic approaches (most notably MPI-everywhere) which usually have no explicit data layer. For illustrative purposes only and without actually suggesting a concrete implementation, this could involve a hierarchical storage management system, a data management tool optimising data placement, a software defined network implementing a subscriber pattern, and a management DB plus GUID server providing consistent DataObject [AD2] journalling.

MapReduce requires serious restructuring of code. The dataflow system, on the other hand, treats Processing Components (i.e. Linux Containers) as black boxes whereby strongly typed inputs and outputs are defined once when building a logical graph which in turn is derived from a processing pipeline recipe for a given instrument configuration.

The data layer glues together domain specific processing recipes - that is the orchestration of Processing Components expressed as a graph - and employs (commercial) middleware solutions to deal with consistency and availability. Especially when combined with Compute Islands built from commodity hardware there is, as a side-effect, minimised vendor lock-in, because the data layer slices through deep stacks of proprietary inflexible (or costly to change) management frameworks.

Again, given that no known data management system copes with the performance requirements (at least not within perceivable cost constraints) there is no alternative than to divert to a dataflow approach that allows tailoring to SDP specific combination of requirements.

Exposing the SDP design hotspot as a design component, namely the data layer, has drawn a lot of attention and sometimes criticism. This should be seen as an opportunity to focus on the challenge. Prototyping will continue to help improve the understanding of how to maximise throughput with moderate changes to domain-specific code. The associated risk of the dataflow approach is not a result of the architectural choice, but of the performance requirement and the ultimate user expectation on observatory capabilities. The dataflow choice gives the system the flexibility to incorporate and swap in and out third party components throughout the system life cycle and will allow to perform system integration and tuning at a much higher fidelity than with a deep proprietary software stack. We are convinced that the dataflow approach has currently the best potential for meeting the challenge of sustained data throughput.

❇