An Example RADPS Workflow Decomposition

PREPARED BY	ORGANIZATION	DATE
Urvashi Rau	NRAO-DMS	02 Oct 2024

Change Record

VERSION	DATE	REASON
0.1	10/02/2024	Initial release.

Introduction

This document illustrates one possible RADPS workflow for end-to-end data reduction. It decomposes a relatively-standard sequence of calibration and imaging algorithms into pipeline tasks and applications, to highlight the atomicity of operations and the levels at which processing may occur concurrently either within a single workflow, or across multiple workflow instances. A mapping of workflows to a pseudo-software stack illustrates details such as where core compute-intensive components lie, the separability of concurrency management software, where state and control are maintained, and how a user interface may connect to the workflow.

The purpose of this document is to get the conversation started on how best to partition the parallelism for a realistic pipeline workflow, and how to realize such workflows in the RADPS operational environment. This document is meant as input to an FY25 Q1 RADPS program increment.

- **Walking Skeleton** : Prototype the structure of the entire software stack by defining dummy (empty) domain functions packaged as Applications, assembling Tasks and Stage graphs using Dask/Airflow/Prefect, etc and then running Workflows with realistic parallelism breadths on a local machine as well as a Kubernetes cluster.

- **Questions** : Design choices that may be evaluated using such a prototype may include (but are not limited to) the ease of use and scaleability of different workflow and workload management frameworks, the ideal granularity of a TaskWorkload, mechanisms for data caching/locality and whether or not they are needed, operational choices of on-the-fly versus to-disk operations in situations where it is algorithmically possible to do either, potentially disjoint timescales of operating different sections of the pipeline and the related checkpointing needs, local reproducibility of the operational parallelism environment for debugging purposes, and accessibility of the software interface for heuristic development.

Reference documents

- SDP Conceptual Architecture (Mark Whitehead)

    - Nomenclature : A **Workflow** may be expressed as a graph. Each graph node is run as an **Application** within a container. Pipelines are workflows/graphs with a prescribed sequence of **Tasks**, grouped into **Stages**. Each Task implements algorithms to achieve one logical step of a pipeline Workflow. A Task that needs to be sent to a resource manager is called a **Workload**. Each Task graph is assembled from a collection of Applications (Domain, Data I/O and Operations functions) using Map/Reduce/Shuffle where needed. Multi-tenancy is achieved when multiple Workflows are run concurrently.

- RADPS User Interaction Document (Joseph Masters)
    - Types of users : **PL Operator**, **QA reviewers** and **External users** will run a pipeline, look at weblogs, restart the pipeline at predetermined Stages/Tasks and may adjust Stage/Task parameters. **Developers** and **Commissioning Scientists** will also step inside each Task and edit the algorithms (including parallelism graphs).
- Algorithm Architecture Document (Sanjay Bhatnagar)
    - Applications : Tasks (or steps within Tasks) that implement iterative solvers have domain-layer components that may be classified into **UpdateDirection**, **UpdateModel**, UpdateStep, **ExitCriteria**, PreSolvePrep, InLoopPrep, DataTransform, ImageTransform. Each component is assembled into an application that represents a single graph node (and may be run within a container in (say) a Kubernetes pod).
- ALMA Pipeline User's Guide (Todd Hunter and ALMA PLWG)
    - A specific sequence of pipeline stages (or Tasks) that illustrate what a typical Workflow instance would look like. Each Task may itself contain a collection of science-domain algorithms assembled into Task graphs by combining domain Applications with Map/Reduce/Shuffle. Parallelism choices at this level depend on the algorithms being run, such as whether adjacent steps may be parallelized together or separately.

An Example Pipeline

A typical pipeline for (say) spectral line imaging contains a series of Tasks that may be grouped into Stages. Each Task may have distinct resource requirements and parallelization choices. User interaction for quality assurance typically occurs soon after the QA steps, after which the sequence may be resumed at a few predetermined places.

An example pipeline workflow is shown in this drawing :Diagram_RADPS_Workflow_Example.pdf / Diagram_RADPS_Workflow_Example and in Figure 1 below.

The example sequence used in this memo is adapted from the ALMA Pipeline Users Guide. It is a highly simplified version of the current pipeline, purely for the purpose of illustrating the mapping of high level Architectural constructs to a specific example. Very specifically, the steps that were left out from the ALMA pipeline are wvrgcal, renorm, polarized source steps, mitigation, spwphaseup (phase coherence checks), flaglowgains, gfluxcal, image_precheck, flagtargets(2nd time), and representative bandwidth cubes. Additionally, some structural features from the VLA pipeline were included, to illustrate variety (e.g. making cube or continuum images before vs after self-cal). *This workflow is therefore not expected to exactly match any of the known pipelines, but is intended to illustrate the variety of usage modes that architectural design choices must support.*

Requirements

1. Show all main stages of a typical pipeline from archive to archive. Show what data subsets can be operated upon independently, how the stages depend on each other (dependency graph), and illustrate the multiple start and terminate nodes of an execution graph (i.e. multiple input data sources, and multiple outputs).
    1. Include examples of where ‘state’ or ‘context’ may need to be saved/restored for checkpointing or scheduling.
    2. Show (via dependencies) how intermediate data products (such as calibration tables) must be managed in-between Stages.

2. Show calibration, flagging, analysis and imaging examples with iterative solver loops that require different partitioning/grouping of data or parallelism types. Include trivial parallelism, as well as in-loop parallelism.
    1. Include an example where two adjacent (calibration) solvers require different partition and combination axes. Or, self-cal where calibration and imaging operate in a loop with different parallelism.
    2. Include examples of solver loops within a Task as well as across multiple Tasks.

3. Show examples of map, reduce and shuffle to illustrate options for graph optimization (within Tasks or Stages) and potential trade-offs with code-reuse.
    1. Provide examples that can be used to test data locality and caching mechanisms in a distributed operations environment vs a local instance with shared file system access.

4. Show examples of heuristics generation steps, context management, QA, and web service queries. These are meant to be examples of tasks that are not workloads, and may or may not represent synchronization points (i.e.a serial step, or a shuffle).
    1. Include an example of data-driven decision making where an algorithm choice depends on the outcome of the previous step.

5. Show where different types of users may want to interact with the workflow
    1. Monitoring progress, intervening between Stages to change sequence/parameters, or even within the low level solver loops (for interactive clean).

Diagram

Figure 1 shows an example pipeline workflow with input data consisting of one (or more) bandpass calibrator(s), one (or more) time-variable-gain calibrator(s) and one (or more) target fields that must be imaged separately. Calibration includes the application of antenna position corrections (via a web-service query), solving for bandpass gains, using them in an initial cal_apply step and then solving for time-variable gains, and then applying all gain solutions to the calibrator and target data. The calibrator data are the imaged (per spectral window), and the target source data are further processed with flagging, the identification of line-filled and line-free frequency ranges. Spectral line cubes are then imaged after performing continuum subtraction, and continuum images are made (per spw as well as aggregate) along with an optional self-calibration sequence. Data are read in from the archive just prior to the first stage that needs to operate on it, and output images are saved to the archive once they are ready.

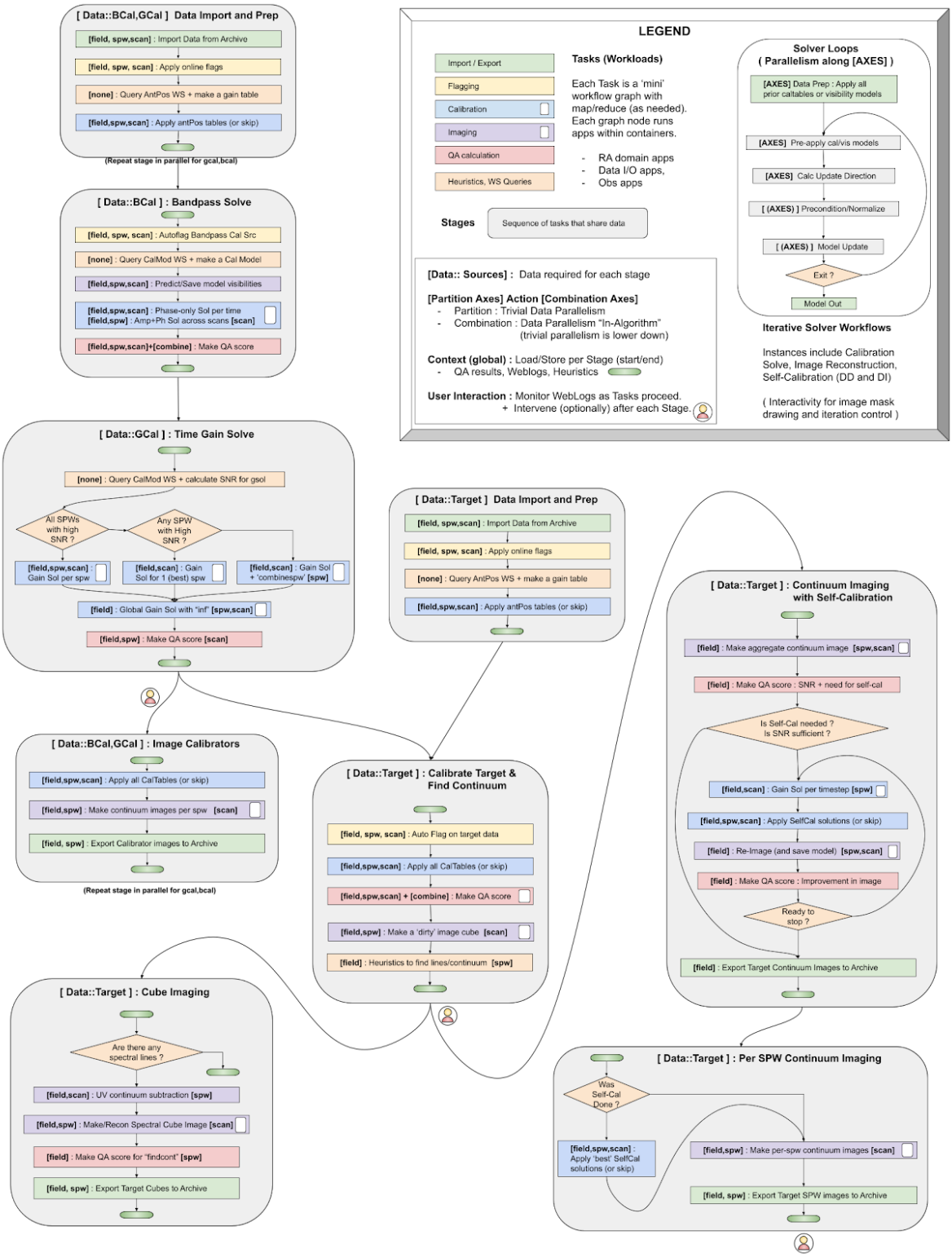

[Data::BCal,GCal] Data Import and Prep
[field, spw,scan] : Import Data from Archive
[field, spw, scan] : Apply online flags
[none] : Query AntPos WS + make a gain table
[field,spw,scan] : Apply antPos tables (or skip)
(Repeat stage in parallel for gcal,bcal)
[Data::BCal] : Bandpass Solve
[field, spw, scan] : Autoflag Bandpass Cal Src
[none] : Query CalMod WS + make a Cal Model
[field,spw,scan] : Predict/Save model visibilities
[field,spw,scan] : Phase-only Sol per time
[field,spw] : Amp+Ph Sol across scans [scan]
[field,spw,scan]+[combine] : Make QA score
[Data::GCal] : Time Gain Solve
[none] : Query CalMod WS + calculate SNR for gsol
All SPWs with high SNR ?
Any SPW with High SNR ?
[field,spw,scan] : Gain Sol per spw
[field,scan] : Gain Sol for 1 (best) spw
[field,scan] : Gain Sol + 'combinespw' [spw]
[field] : Global Gain Sol with "inf" [spw,scan]
[field,spw] : Make QA score [scan]
[Data::BCal,GCal] : Image Calibrators
[field,spw,scan] : Apply all CalTables (or skip)
[field,spw] : Make continuum images per spw [scan]
[field, spw] : Export Calibrator images to Archive
(Repeat stage in parallel for gcal,bcal)
[Data::Target] Data Import and Prep
[field, spw,scan] : Import Data from Archive
[field, spw, scan] : Apply online flags
[none] : Query AntPos WS + make a gain table
[field,spw,scan] : Apply antPos tables (or skip)
[Data::Target] : Calibrate Target & Find Continuum
[field, spw, scan] : Auto Flag on target data
[field,spw,scan] : Apply all CalTables (or skip)
[field,spw,scan] + [combine] : Make QA score
[field,spw] : Make a 'dirty' image cube [scan]
[field] : Heuristics to find lines/continuum [spw]
[Data::Target] : Cube Imaging
Are there any spectral lines ?
[field,scan] : UV continuum subtraction [spw]
[field,spw] : Make/Recon Spectral Cube Image [scan]
[field] : Make QA score for "findcont" [spw]
[field, spw] : Export Target Cubes to Archive
[Data::Target] : Continuum Imaging with Self-Calibration
[field] : Make aggregate continuum image [spw,scan]
[field] : Make QA score : SNR + need for self-cal
Is Self-Cal needed ? Is SNR sufficient ?
[field,scan] : Gain Sol per timestep [spw]
[field,spw,scan] : Apply SelfCal solutions (or skip)
[field] : Re-Image (and save model) [spw,scan]
[field] : Make QA score : Improvement in image
Ready to stop ?
[field] : Export Target Continuum Images to Archive
[Data::Target] : Per SPW Continuum Imaging
Was Self-Cal Done ?
[field,spw,scan] : Apply 'best' SelfCal solutions (or skip)
[field,spw] : Make per-spw continuum images [scan]
[field, spw] : Export Target SPW images to Archive
LEGEND
Import / Export
Flagging
Calibration
Imaging
QA calculation
Heuristics, WS Queries
Tasks (Workloads)
Each Task is a 'mini' workflow graph with map/reduce (as needed). Each graph node runs apps within containers.
- RA domain apps
- Data I/O apps,
- Obs apps
Stages
Sequence of tasks that share data
[Data:: Sources] : Data required for each stage
[Partition Axes] Action [Combination Axes]
- Partition : Trivial Data Parallelism
- Combination : Data Parallelism "In-Algorithm" (trivial parallelism is lower down)
Context (global) : Load/Store per Stage (start/end)
- QA results, Weblogs, Heuristics
User Interaction : Monitor WebLogs as Tasks proceed.
+ Intervene (optionally) after each Stage.
Solver Loops (Parallelism along [AXES])
[AXES] Data Prep : Apply all prior caltables or visibility models
[AXES] Pre-apply cal/vis models
[AXES] Calc Update Direction
[(AXES)] Precondition/Normalize
[(AXES)] Model Update
Exit ?
Model Out
Iterative Solver Workflows
Instances include Calibration Solve, Image Reconstruction, Self-Calibration (DD and DI)
(Interactivity for image mask drawing and iteration control)

Figure 1

In Figure 1, gray boxes indicate Stages, the coloured rectangles indicate Tasks/Workloads, the white vertical rectangles inside Tasks represent solver loops (also shown with its components at the top right in the legend), the green ovals represent checkpoints where state or context may be saved and restored and where readiness for the execution of each stage is evaluated during scheduling. The 'human' figures indicate points in the workflow where a human might want to intervene to check status and prior outputs and optionally re-run a prior stage (or set of stages). Decision boxes are drawn to illustrate data-driven decision points that bring in runtime flexibility and unpredictability.

We assume that the data are pre-partitioned along field, spw, scan axes. In this simplified example, "scan" is a proxy for partitioning along the time axis and could be as fine as a single timestep (or it could represent a single "EB" in ALMA nomenclature), "spw" is a proxy for partitioning along the frequency axis and could be as fine as a single channel, and "field" is a proxy for sources or targets that may be treated independently (i.e. not fields of a mosaic). Of course, these are simply choices of convention purely for the purpose of illustration using a simple example.

For each Task listed below, we use [----] to signal the axes of parallelization allowed. Parallelism may be trivial, or be included inside an algorithm when needed (such as a calibration solver that might want to combine data across 'spw' or an imaging solver that combines across 'scan'). A functional decomposition is shown down to the level where all parallelism is trivial (i.e. the functions being parallelized may run independently), and examples are included where adjacent/subsequent Tasks may or may note share parallelism axes.

Parallelism Breadth

This information is approximate (for ngVLA and ALMA-WSU) and is meant to be used only to evaluate scalability of the chosen workflow/workload managers and schedulers, as well as to inform choices of where it may not be worth to deploy parallelism across graph nodes (and opt instead for local compute within a single graph node).

If more refined numbers are required, especially to understand the different tradeoffs and compute-to-I/O ratios of ALMA-WSU and ngVLA, they must be obtained from ALMA-WSU and ngVLA "size of compute" documents. Tradeoffs for single-dish pipelines will also be different and must be included in a formal analysis.

A typical data set (all data required for a PI project) may range from a few 100 GBs (for testing or early commissioning) to upto 5-10 PB for the largest ngVLA datasets. Based on the general assumption that in each hour of observing, a calibrator is observed for 10 minutes and the target for 50 minutes, we can say that calibrator data will typically be about 20% of the target data volume along the time axis. Observing 'scans' may be 5 -30 minutes long. The frequency setups are typically shared between calibrators and targets, and can range from a few 100 data channels up to a few 100000 data channels.

Therefore, very approximately,

- Calibrators :
  - n_field may range from 1 to 10,
  - n_spw may range from 10-100 with each spw containing between 1000 to 8000 data channels
  - n_scan may range from 10 - 100 scans spread throughout the observation.
- Targets
  - n_field may range from 1 to 10 (depending on the science case. For the most part, with the definition of ‘field’ used this document, they may be treated independently)
  - n_spw may range from 10-100 with each spw containing between 1000 to 8000 data channels
  - n_scan may range from 10 - 1000 scans (assuming ~10minute scans and observations ranging from a couple of hours long to a few 100 hours long). Number of target scans is roughly 5-10 times more than the number of calibrator scans.
- Image Cubes :
  - n_field is same as above.
  - n_spw and channels are also same as above (in the highest resolution cube imaging cases). For continuum imaging, single plane images may be made per spw or across all spws (resulting in just one image plane)
  - Number of pixels on a side ranges from a few 1000 to 20000.

Walking Skeleton v0.1

A very simple Dask-based Skeleton implementation has been done to test if the above pipeline sequence (a) holds together (logically) when actually coded up, (b) illustrates an appropriate level of domain complexity with outer/inner loops and trivial/in-algorithm parallelism and (c) can be decomposed into Domain functions, Tasks/Workloads with parallelism and Workflow Stages. There is no actual data flowing through the example, but Processing Sets are represented as dictionaries with ‘n_field, n_spw, n_scan’ values to indicate data partitions along which parallelism may occur (depending on algorithm choices).

Code / Movie

Code

A Jupyter notebook : ipynb & HTML (you may have to download the HTML and open it in a browser)

The skeleton implementation illustrates the separation of code into Domain Functions (currently, empty Python functions with ‘sleep(0.5)’ statements to simulate ‘work’), Task functions that construct graphs for each algorithm and which may be scheduled independently as a Workload, and Stages that group the Tasks into blocks that fit together for each logical subset of data (e.g. calibrators vs target sources). Currently, domain functions represent only “RA domain” applications only, but Data and Observer sidecar applications may be added later. The notebook also illustrates the graphs that result for each Stage, the high level workflow showing only the Stages, and a full workflow graph that combines all Stages with the detail.

Graphs are assembled using dask.delayed() and then visualized using dask.visualize(). For the purpose of illustrating task graphs a fixed number of iterations are set via a parameter offered at the Task level. But, in a real application, an arbitrary/adaptive number of iterations can be achieved simply when the compute is actually performed and the outputs are used to decide the next step in the workflow.

An example code snippet is shown in Figure 2, for a (mock) Processing Set, a flagging Task containing two algorithmic steps (with different parallelism), two domain Applications used in the Task, and the use of the Task within a sequence that forms the import and data preparation Stage of the Workflow. For this simple example implementation, this illustrates the level of coding complexity/simplicity that would be required during heuristic development and prototyping. When realized in a production system, perhaps via the use of graph-viper to provide abstraction for parallelism and metadata management, it will be important to evaluate whether the coding complexity remains similarly straightforward or not.

```
datashape = {'bcal':{'n_field':1, 'n_spw':3, 'n_scan':1},
             'gcal':{'n_field':1, 'n_spw':3, 'n_scan':4},
             'target':{'n_field':1, 'n_spw':3, 'n_scan':5} }

def task_flag(inp,datashape,src='bcal'):
    """
    Run a flagging algorithm (metadata flags, autoflag, etc.. )
    Parallelism : Depends on algorithm. Here, we pick separate runs per field,spw and scan
    """
    n_field = datashape[src]['n_field']
    n_spw = datashape[src]['n_spw']
    n_scan = datashape[src]['n_scan']

    setup = dask.delayed(calc_heuristics)(inp)

    flag_par=[]
    for i in range(0,n_field*n_spw*n_scan):
        flag_par.append( dask.delayed(flag_data)(setup,i) )

    return flag_par

def calc_heuristics(inp,id=0):
    time.sleep(ns)
    return

def flag_data(inp,id=0):
    time.sleep(ns)
    return

def stage_import_prep_data(inp,src=''):
    '''Import Data
       Apply Online flags
       Antpos WS Query and make antpos table
       Apply antpos cal table
    '''
    res1 = task_archive_import(datashape,src=src)
    res2 = task_flag(res1,datashape,src=src)
    res4 = task_applymodel(res2,datashape,src=src)
    res5 = task_store_context(res4)
    return res5;
```

Movie

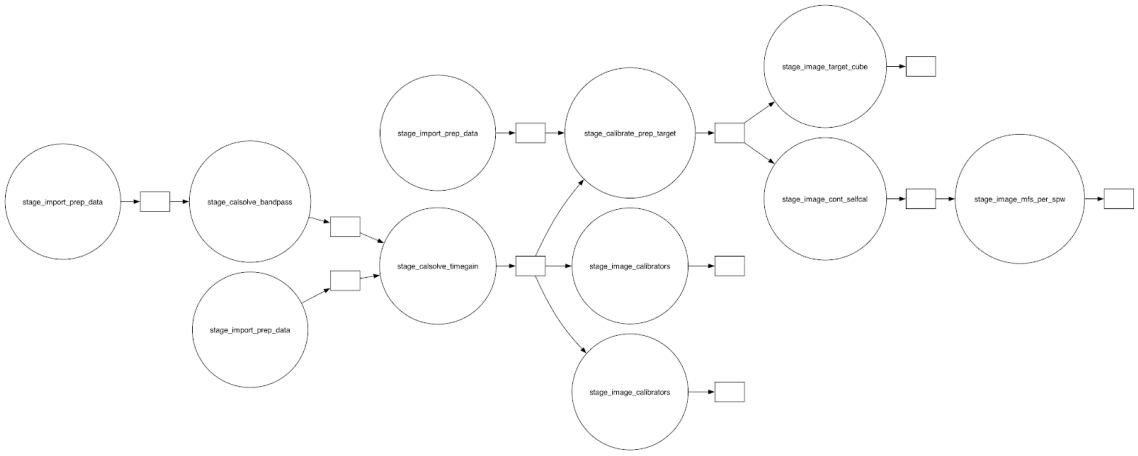
[Execution graphs for two workflows being executed in tandem]

The parallelism is parameterized along dataset axes of n_field, n_spw, n_scan and the graphs can be re-generated for different Processing Set shapes. As an initial example, two workflows were generated with different ‘processing sets’ parameterized simply by n_field, n_spw, n_scan to illustrate how data shapes can change the parallelism structures. The two workflows were then run within a single Dask cluster with 20 cores and a video was recorded to show the dask execution graphs. With each workflow running a maximum of “n_field x n_spw x n_scan” nodes at any time, this illustrates the most basic form of **multi-tenancy**.

Workflow Graphs

The examples below illustrate the structure of a few graphs assembled in Tasks and Stages. The purpose is to show how such a framework may be used to evaluate some design choices.

- A Stage-level workflow diagram (graph) is shown here : fig_stage_workflow.png.
    - A dataset with a "bandpass gain calibrator source", "time-variable-gain calibrator source", and one "target source" has multiple data import stages (3 starting nodes) and multiple final images products that are separate for each source and image type (2 terminal nodes for the calibrator images, 1 for target cubes, and 1 for target continuum).
        - Note to the reader : The circles in this diagram represent the large Gray boxes in Figure 1. Within each circle is a sequence of Tasks as depicted by the colored rectangles within the Gray boxes in Figure 1.



- The graph shows dependencies between stages, highlighting (obvious) facts such as independent data subsets may be pre-processed independently and concurrently for all Stages prior to one that requires outputs from multiple data subsets. For example, the “stage_calsolve_timegain” stage begins with the application of calibration solutions that were derived in the “stage_calsolve_bandpass” Stage.

- Similarly, after all calibration solutions have been applied to the data (calibrators and target), imaging of the calibrators and targets may proceed concurrently. For the imaging of the target source, once the ‘find_continuum’ step runs within the “stage_calibrate_prep_target” Stage (in our example), the Cube and Continuum imaging may also proceed concurrently.

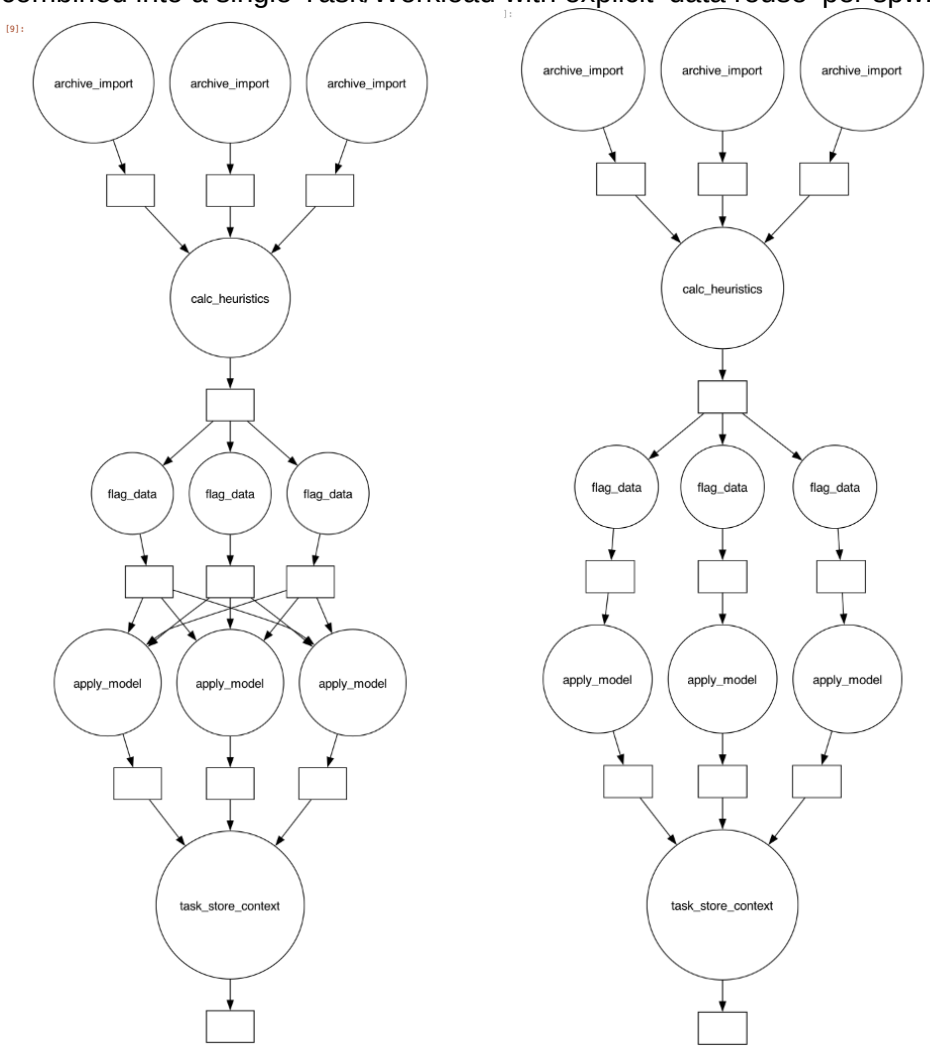
- Following the convention used with current ALMA pipeline operations, the entire workflow is split into three ‘mini workflows’. One purpose is to accommodate imaging projects that require multi-configuration data that may be taken several weeks to months apart. Joint imaging is possible only after all the observations are complete, but calibration must be performed within (say) 12 hours of the observation itself to allow for the option of repeating an observation (if needed) to meet QA standards for “calibratability”. In our simple example, this partitioning could look as follows
    - The “import_prep_data” for the bandpass and gaincal calibrator sources, the “calsolve_bandpass”, the “calsolve_timegain” and the “image_calibrators” (for both cals) form a calibration mini-workflow.
    - The “import_prep_data” for the target, and “calibrate_prep_target” may form a mini workflow for imaging-preparation.
    - The “image_target_cube”, “image_cont_selfcal” and “image_mfs_per_spw” would form a third mini-workflow

- These mini-workflows also mark boundaries where user-interaction may be required. A DA, for example, might need to inspect weblogs at these boundaries and assess whether or not to re-run a previous mini-workflow with edited parameters.

- A Task-level workflow diagram, showing parallelism at and within each Task, as well as with a sequence of Tasks strung together to form a Stage. Examples of ‘map’, ‘reduce’ and ‘shuffle’ are shown visually.

    - These graphs illustrate two options for the “stage_import_prep_data” shown in Figure 1. The steps (Tasks) include the
        - Importing of Data from the Archive (parallelized across 3 spws),
        - The application of online flags (parallelized across 3 spws, but with a serial QA step at the start of the Task)
        - ~~The query of an external Web Service to create an antenna position correction gain table,~~
        - The application of the antpos gain table (multiply/divide visibilities with appropriate phase corrections), and the saving ot context/state for checkpointing purposes.

- The graph on the left is with the flagging and cal_apply steps implemented in separate Tasks (Workloads), showing an instance of a possibly unnecessary data shuffle between subsequent Tasks that happen to share a parallelism axis. The graph on the right is an example where the flagging and apply_cal steps are combined into a single Task/Workload with explicit ‘data reuse’ per spw.



- This example is meant only to illustrate how programming choices can affect graph structure. In practice, optimization choices will depend on the chosen parallelism framework and data transport/caching mechanisms, and one may or may not need to worry about such optimizations during programming (and heuristic development/prototyping).

    - For example, Dask contains graph optimization algorithms that might be able to detect instances where data may be shared across subsequent nodes and treat them as a ‘single node’. Or, there may be a data caching mechanism (such as Pelican, used in ARDG prototyping) which recognizes that data may be locally available after its first access. Or, it may suffice to ensure data locality at the Stage level, allowing Tasks to freely use different access patterns as needed within their algorithm graphs, with no data-transfer overheads.

- A Task-level graph showing an example of an iterative solver with both trivial and in-algorithm parallelism: fig_image_calibrators_stage.png

    - Note : Not all solvers require in-algorithm parallelism. Solvers that operate on small amounts of data may complete their iterative loops within a single node in such a Task-level graph. Examples include some calibration solve steps and per-channel image reconstruction. This example here, though, uses both types of parallelism and is most representative of continuum imaging of the target source where large amounts of data must be processed (and parallelized over) for the update_residual step of the iterative solver.

    - This graph shows the 'image the calibrators' stage with multiple Tasks/Workloads in it, with both trivial as well as in-algorithm parallelism within the Imaging Task. The graph goes all the way down to the domain function calls that run "per data chunk". This graph represents a dataset with 1 field, 3 spws and 4 scans for the calibrator.

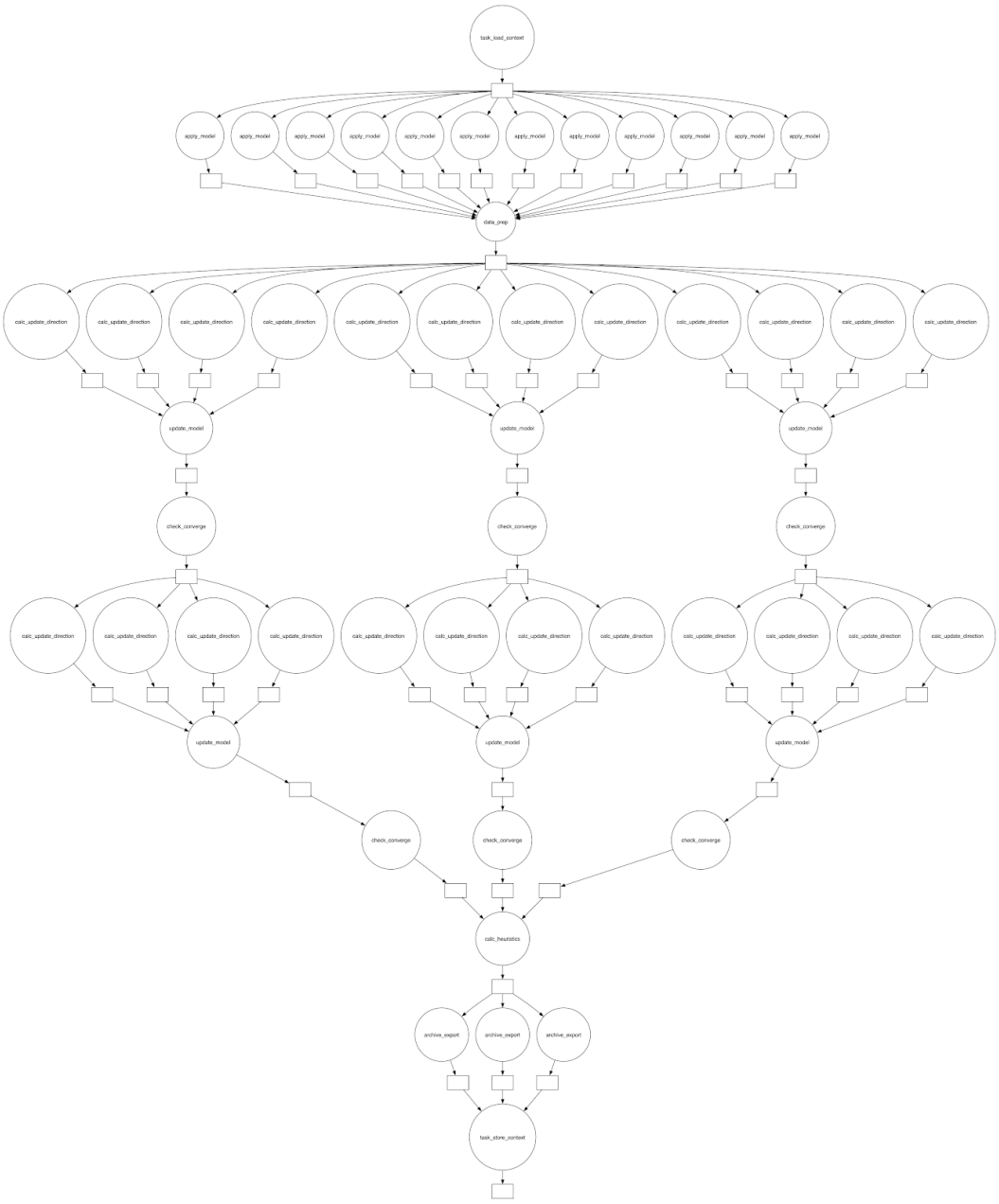
        - The first Task is to load the state or context and retrieve data products from prior Stages that are required as input to this one.

        - The second Task is to run apply_cal to apply calibration solutions. This may be parallelized over n_spw x n_scan = 12 graph nodes.

        - The third Task runs imaging solvers separately per spw (3 of them), but combines data from all 4 scans to make each image. Therefore, the calc_update_direction (a.k.a. ‘major cycle’) step is parallelized across 4 scans, but the ‘update_model’ (a.k.a deconvolver or minor cycle) step is run within a single graph node and includes the appropriate gather and normalization for forming the residual images that go into the deconvolver. In this example, there are 2 solver iterations. There are three output images (one per spw).

        - The last Task sends the output images (3 of them, one per spw) to the archive. This step may also be parallelized across spw.

        - The final step is the saving of state/context for checkpointing.



- The Stage code for our simplified example is shown here :

```
def stage_image_calibrators(inp,src='bcal'):
    """
    Apply caltables to cal sources
    Reconstruct images separately per field and spw (combine across scan)
    Export to archive.

    Note : This stage can be run in parallel for 'gcal' and 'bcal' sources, from the higher level workflow dag.
    """
    res1 = task_load_context(inp)
    res2 = task_applymodel(res1,datashape,src=src) ## Apply caltables.
    res3 = task_solve(res2,datashape,src=src,combine='scan') ## Image each field and spw separately. Combine-scan. Do 2 iteration loops.
    res4 = task_archive_export(res3,datashape,src=src,par=1) ## Export images, parallelize by field and spw
    res5 = task_store_context(res4)
    return res5
```

- Finally, we have an example that shows all Stages andTasks, showing parallelism down to the level where domain methods are trivially parallelized : fig_full_workflow.png (13MB). This is purely for illustration purposes as such a detailed graph can quickly become difficult to read/parse visually.
    - (You may have to download the png file and open it locally. Or, download the notebook HTML and open in a browser.)

Other Ideas to Test

Below are some additional notes to consider when making a to-do list of things to evaluate using such a prototype “skeleton”.

1. **Data partitioning** : We assume that the MSv4 format is being used, with data pre-partitioned along field, spw, scan axes. If needed, a granularity may be picked such that one will never need to partition a single MSv4 for parallelism. This will help with data locality/caching across steps/tasks that may need different parallelism axes. Also, the workflow decomposition is implementation-agnostic and applies to non-RADPS code and data formats too. This connects to prototyping efforts currently ongoing within the ARDG, as well as options of using CASA6 methods with modern workflow mgmt options.

2. **OTF vs to-disk Caltable/ModelVis Apply** : All Caltable applications and model visibility predictions are listed as separate Tasks. But, they can both be done “on the fly” in the “apply_model” step of the solvers. A choice between ‘to-disk’ and ‘OTF’ caltable and modelvis application could be driven by the processing environment. For example with a shared filesystem we may want to save to disk and reduce repeated compute. For more distributed execution, we may want to minimize data transport and pay the cost of extra computing (within each node/application). However, if such a choice is provided, one must keep in mind the debugging use-case where it is important to be able to replicate the steps used in a production environment, on a local machine (or laptop).

3. **Algorithm optimizations** are possible with a new and scalable parallelism framework, and one does not have to be restricted to the protocols followed by existing pipelines. One idea for continuum imaging could be to calculate per-spw mfs images and construct aggregate images in the image domain. This would avoid needing to make continuum images twice just to get both per-spw and aggregate images. This is an example where algorithms of existing pipelines could be modified to take advantage of improved parallelism capabilities, and to reduce task/stage repetition (and therefore compute cost).

4. **Ordering of Stages** : Workflow Stages may operate in different orders, depending on the needs of each pipeline or telescope. Code re-use with the decomposition shown in this diagram reflects that already implemented in the ALMA vs VLA pipelines (e.g. each telescope makes different choices about whether to run cube and continuum imaging steps before or after self-calibration or both). Flexibility of Stage-level re-ordering should be allowed.

5. **QA vs Heuristics** : There is a minor difference to point out between QA Tasks and Heuristics Tasks. Although they may both use the same domain layer Applications, QA tasks may only be used for 'inspection of intermediate data states and products' and are not critical to the actual operation of the workflow. They may therefore be run asynchronously, if there is any advantage to doing so. It may also be an option to run QA diagnostics and weblog generation code only at the end of each 'mini workflow' and not within every Stage/Task. Heuristics Tasks on the other hand are an integral part of the workflow itself.

6. **Solver Loops and Interactivity** : Solver loops may be implemented in three ways, depending on data parallelism and user-interaction needs. W.r.to user-interaction, it is important to note that operations versus commissioning use-cases may require different levels of user-interaction, but will expect the exact same numeric or algorithmic building blocks (for numerical reproducibility of the results).

    1. **Within an Application** (i.e. one graph node) : The entire solver runs independently within a single graph node. This use-case is when there is no need for data parallelism that is managed by the Task/Workload scheduler. For example, a calibration solve that operates independently per scan or spw. Or, a cube imaging that can run separately per data/image channel.

        - For such a solver, there will be no user-interaction option within the solver loops.

    2. **Within a Task** (i.e. across graph nodes) : The steps of the solver run as separate Applications, but they are tied together within a single Task/Workload. This use-case is when there is a need for data parallelism for one or more of the steps of the solver, especially when there is a 'reduce' operation in-between. One example is continuum imaging where the calc_update_direction (or major cycle) may be trivially partitioned, but the residual images from each partition must be combined prior to the model_update step (minor cycle). Another example is a calibration solve with "solint='inf'" where (say) data from all scans of a calibrator feed into a single set of bandpass gain solutions.

        - For such an implementation, there will still be no user-interaction option, as the entire solver is still within a single Workload.

    3. **Across Tasks** : Interactive Imaging is a use-case where each step of a solver (or groups of solver steps) may be treated as a separate Workload, to allow for *asynchronous user-input* to draw masks, monitor progress and perhaps edit iteration control parameters.

        - One detail to note here is that in current pipeline operations, interactive mask drawing is actually very rarely used (it is more for the end-user use-case). For pipeline operation, a DA or scientist may at most want to (i) make a dirty/residual image (ii) draw a mask externally/asynchronously using CARTA, for example, and then (iii) run imaging solver loops non-interactively.

7. **Context and Intermediate Data Products**
    1. **Context** : A record of what has already been done, some QA metrics (and weblogs), and results of heuristics calculations from one Stage that may be supplied as input to another Stage (e.g. find_continuum results, or a mapping of calibration solutions to data selections that may need to be re-used for every caltable application (if done OTF)).
    2. **Intermediate data products** : The equivalent of Caltables, Flagversions, Data column versions (if cal_apply and modelvis_prediction are done to-disk and not OTF). The definition, storage and re-distribution of these intermediate products need to be considered as part of an architecture evaluation. For the use-case of different pipeline Stages being run at different times,management of these intermediate products becomes part of checkpointing. An example use-case is multi-config ALMA data where calibration happens within 12 hrs of each observation but imaging can happen only after all data blocks have been observed (after a few months). The ESO Data Processing System could be a useful reference for ways to structure data context handling for such a system.

8. **Commissioning Scientist Use-Case** : Commissioning scientists will need access to the code at a level where they can experiment with algorithms and heuristics. This might mean adjusting the order of “domain calls” within a Task (which might require changing the parallelism setup), inserting new domain calls to change heuristics, and also editing the parameters that go into each domain call. It is one of the requirements (from ALMA) to enable such a programming interface that is not too difficult for a non-programmer to learn.
    1. With this prototype, there is an opportunity to evaluate whether a commissioning scientist or DA finds the Task-level programming interface accessible enough or not. With the use of graph-viper as a programming API, the actual graph building and MSv4 metadata management is abstracted away, and it might be possible to craft a few examples that illustrate that one can focus on editing the algorithms and heuristics.
    2. **One suggestion is to include a small ‘user survey’ as part of the prototyping effort** (with just the skeleton, but perhaps including graphviper to showcase the abstraction).
        1. A focus group of DAs and commissioning scientists could be invited to use the Task level programming interface to mock up how they would want to fiddle with heuristics and parameters.
        2. They could also try out running the skeleton on multiple computing resources, to mimic the use-case of running operations on (say) a Kubernetes cluster but needing to reproduce the same run and debug it locally on one’s laptop.
        3. Including instances of “failure modes” may also be useful here, to ensure that the various user-types listed in the RADPS User Interaction documents are adequately served.
    3. While this does not strictly influence operational architecture, it is a critical factor to get right. If not, developers may be asked to provide complicated intermediate level interfaces with a large number of control parameters that would enable ease of use for commissioning scientists but would greatly increase development and maintenance complexity and cost. A middle ground that would benefit everyone would be to showcase an “easy to use” programming interface that developers and commissioning scientists can simply share.

More references

- Draft API specs - CASA : NCS-213 CASA Science Specification and API
- Draft API specs - PIPE : pipeline abstraction
- Calibration Math : Calibration Framework Model (VIPER)
- Adaptive Data Reduction Workflows for Astronomy
    - The ESO Data Processing System :
- Size of Compute documents
    - ALMA : ALMA Memo 626 Estimates of ALMA WSU Data Properties
    - ngVLA : ngVLA Data Rates and Computational Loads (Update) ngVLA Computing Memo 11

Acknowledgements

The author would like to thank NRAO DMS and NAASC colleagues Mark Whitehead, Jeff Kern, Jan-Willem Steeb, Rui Xue, Remy Indebetouw and others within the CASA and ARDG teams for ideas and conversations that have guided the contents of this memo.