3rd URSI AT-AP-RASC, Gran Canaria, 29 May – 3 June 2022

Time-Domain Science Pipelines for the OVRO-LWA

Yuping Huang*(1)(2), Mei-Ling Laures*(1)(2), Marin M. Anderson(3)(1)(2), Casey J. Law(1)(2), and Gregg Hallinan(1)(2)
(1) California Institute of Technology, Pasadena, CA 91125, USA
(2) Owens Valley Radio Observatory, Big Pine, CA 93513-0968, USA
(3) Jet Propulsion Laboratory, California Institute of Technology, Pasadena, CA 91109, USA

Abstract

The Owens Valley Radio Observatory Long Wavelength Array is a low radio frequency all-sky imaging radio interferometer. The full 352-element array will generate more than 2 TB of visibility data per hour of observation. One of the array's primary science cases, the search for variable radio emission from exoplanets and for transients, require fast and high dynamic range interferometric imaging. Here we detail the design and implementation of a two-pipeline infrastructure that minimizes development cost: an offline pipeline that facilitates experimentation with existing packages, and a real-time pipeline that minimizes overhead.

1 Introduction

Modern radio interferometric arrays produce vast amounts of data. Progress in the last decade in calibration and imaging algorithms has enabled higher dynamic range and wider field of view, especially at low radio frequencies (< 400 MHz). Given the large space of available algorithms, data processing pipeline infrastructures that facilitate experimentation and rapid deployment at scale are key to maximizing the scientific potential of telescope arrays. After the correct software and algorithm solutions are identified, a real-time pipeline integrating these components can drastically increase the scientific output of a telescope.

This paper details the data processing pipeline design and implementation for the time-domain science for the Owens Valley Radio Observatory Long Wavelength Array (OVRO-LWA), an interferometric array at Caltech's Owens Valley Radio Observatory (OVRO) near Big Pine, CA. OVRO-LWA serves as a software demonstrator for the radio camera concept[1], where a streaming pipeline produces science-ready images without the need for visibility-based deconvolution. We summarize the background and requirements for the project in § 2, describe our offline and real-time pipelines respectively in § 3 and § 4, and conclude in § 5.

2 Design Considerations

The OVRO-LWA is currently going through its final stage (Stage III) of upgrade funded by NSF MSRI. The upgrade

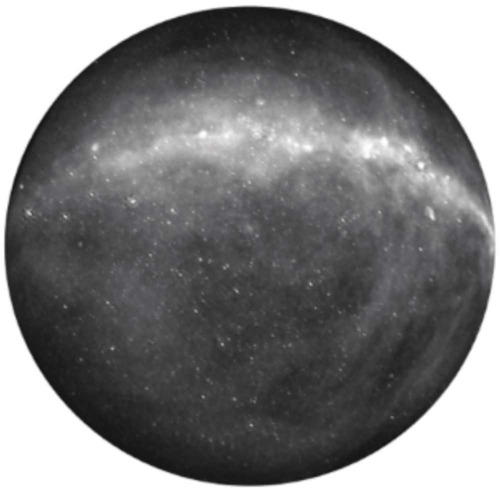

Figure 1. A 13 s snapshot image with the Stage II OVRO-LWA. There are more than a thousand stellar systems within 25 pc that are monitored simultaneously within the field of view

will increase the number of antennas from 288 to 352, spanning a maximum baseline length of 2.4 km, and feature redesigned analog, digital, and compute backends. For all stages of the array, the correlator produces full cross correlations across all baselines every 10 s and thus enjoys an all-sky field of view. The observing frequencies span from 12 MHz to 85 MHz with the band below 25 MHz usable at night only. The angular resolution at 25 MHz is $\sim 15'$. With 352 antennas and 2944 frequency channels across the band, the OVRO-LWA will produce 2 TB of visibility data per hour.

Data processing for the OVRO-LWA takes place on a dedicated compute and storage cluster housed near the telescope and connected to the correlator via a dedicated Ethernet switch. Currently, a small compute cluster with 10 compute nodes runs the data reduction. Each compute node possesses 16 CPU cores and 64 GB of RAM. These compute nodes share a `Lustre`[2] distributed storage system with 5 Object Storage Servers (OSSs) and 1 Management Server (MGS), totaling 560 TB of usable space. The shared `Lustre` file system significantly simplifies the pipeline de-

[1] https://www.radiocamera.io/

[2] https://www.lustre.org/

sign because the pipeline sees a unified file system. The planned Stage III upgrade will expand the cluster: the new cluster will consist of 11 compute nodes, each with 48 CPU cores, 512 GB of RAM, and the ability to accommodate GPUs; 9 storage nodes will form the `Lustre` storage cluster with 4.3 PB of usable space. Benchmarking on the current compute and storage system informed the parameters of the upgraded cluster.

Time-domain science with the OVRO-LWA includes all-sky searches for transients and emission from exoplanetary magnetospheres (Fig. 1). Theorized transients at low radio frequencies include stellar radio bursts that may trace the plasma environments around stars, and prompt counterpart to binary neutron star mergers. Bright radio emission has been observed from all magnetized planets in the solar system and is one of the most promising means of detecting magnetospheres of planets outside our solar system. Characteristics of planetary magnetospheric radio emissions include high variability and high degree of circular polarization. A competitive time-domain survey for transients and exoplanetary emission requires thermal noise limited, time resolved, and all sky imaging between timescales of seconds to hours in Stokes I (for transients) and in Stokes V (for exoplanets). The baseline requirement for the Stage III array is to process a 1000-hour survey offline within reasonable time, with a stretch goal of real-time operations. We are developing two different pipeline infrastructures to satisfy these requirements while minimizing overall development cost:

1. A distributed, file-based, offline processing framework (detailed in § 3) allows for rapid experimentation and iteration with existing software packages that operate on measurement sets[3]. The offline framework will help us identify the appropriate combination of software packages and algorithms that balance performance and dynamic range requirements.
2. A planned real-time pipeline framework (detailed in § 4) will incorporate algorithms and software packages validated via the offline pipeline and minimize overhead by passing data in-memory between processing steps. Some parts of the sky are computationally cheaper to process than others due to the absence of bright sources. The real-time pipeline is a focused effort to achieve real-time time-domain science for as much of the sky as possible.

3 Offline Pipeline: From Prototype to Production

The current processing infrastructure for the OVRO-LWA exoplanet and transient science is an offline pipeline.[4] The goal of the offline pipeline is to reduce the friction between

[3]https://casacore.github.io/casacore-notes/229.html

[4]Hosted at https://github.com/ovro-lwa/distributed-pipeline

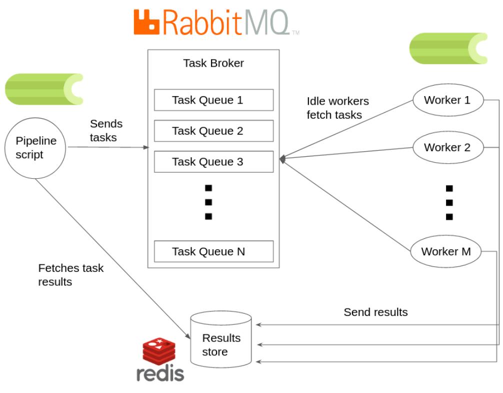

Figure 2. An illustration for the distributed queue-backed pipeline based on `Celery`.

experimentation on a small set of data and batched processing on a distributed cluster on a large amount of data. It leverages existing software packages and input/output conventions. We opted to build a pipeline framework based on Python packages, as opposed to using a general-purpose task scheduler like `Slurm`[5], because the cluster has a dedicated purpose and a relatively small number of users. The added structure of Python scripts also allows users to share and reuse pipeline scripts with ease. This infrastructure was used for a recent transient survey with the OVRO-LWA [1], reusing algorithms developed in previous work [2] and other existing radio interferometry calibration and imaging packages.

The pipeline infrastructure consists of three layers:

1. The top execution layer should be an off-the-shelf package within the Python ecosystem that schedules and executes tasks across the cluster. The choice of the execution layer may change as the scale of the project evolves.
2. A middle “adapter” layer bridges between the top execution layer and the bottom functional interface layer, typically converting types or filling in necessary metadata for the execution layer. This layer should be thin and decoupled from the other layers, so that switching the execution engine requires less effort. Currently, the adapter layer takes the form of a functional decorator.
3. We stipulate that the bottom layer consists of Python functional interfaces to routines that do work on a minimal unit of input (often a measurement set with a single time integration). The routine can be a native Python function, a wrapper function to code written in a different language, or a `subprocess` call to a compiled program with a command-line interface. Each function must take a path to the input file and returns the path to the output. Each function thus reads

[5]https://slurm.schedmd.com/

from and writes to files, because files are the common input/output of existing packages. A user can use these functions for exploratory analyses in an interactive Python shell or a notebook, and tune parameters accordingly in preparation for batched processing with the pipeline.

Decoupling the execution layer from the processing layer allows us to iterate on both processing algorithm and execution framework quickly. The current distributed execution layer is `Celery`,[6] a Python distributed task processing framework. The pipeline code puts tasks in queues (backed by `RabbitMQ`[7]) through `Celery`, and each idle worker process on every node of the cluster picks up tasks from the queues (Fig. 2). We find `Celery` well matched to our needs with its simplicity, native Python support, tolerance to power outage, and monitoring capability. `Celery` allows a user to specify a pipeline's steps and input/output with straightforward Python semantics. For example, the following code snippet asks the pipeline to chain the output of the first task to the input of the second task and apply the chain of tasks to all elements in `ms_list`:

```
for ms in ms_list:
    (task_1.s(ms, par_1) | task_2.s(par_2))()
```

The non-blocking `.s()` method call queues the task. The pipe operation (`|`) is a shorthand for specifying a chain of tasks, where the output of the first task serves as the input for the second task. Since `Celery` sends all the requisite tasks to the task queue to be executed by the cluster, the user does not have to manage the execution in the pipeline code. The above snippet can be shortened with list comprehension and `Celery`'s `group` class. Specifying the pipeline processing in Python allows pipeline-level code reuse via converting pipeline code into a parameterized function. It also allows pipeline mock testing, which ensures that each step in the pipeline receives the correct input. One can also build higher-level abstractions (with configuration files or the Common Workflow Language[8]) on top of the pipeline execution code. A `PathManager` class indexes spectral windows, observing time, and file types to file paths on the shared file system. `PathManager` enforces consistent file-naming conventions across the project. New naming schemes can be added by subclassing `PathManager` and pipeline code can be reused with new file naming schemes by switching to the new subclass.

The file-based framework with extensive code reuse accelerates deployment for batched processing after prototyping on a small set of data. However, it becomes inefficient for tasks with high file I/O to compute ratios. To mitigate the issue, we merge tasks with high file I/O to compute ratios with an adjacent step so that the Linux buffer cache can significantly speed up file I/O.

[6]https://docs.celeryproject.org/

[7]https://www.rabbitmq.com/

[8]https://www.commonwl.org/

4 Real-time Pipeline

Although offline pipelines may offer flexibility, they typically make heavy use of disk-backed data, where reading and writing is an order of magnitude slower than data kept in cache and RAM. A real-time pipeline that keeps intermediate data products in RAM will be vastly more efficient. This section starts with a comparison of the old and new memory management paradigms; we will then describe how the new Memory Lender framework will work in the Recycling library, and its planned usage with the OVRO-LWA.

4.1 Memory Alternatives and Comparison

Before deciding on a new framework, we undertook an in-depth exploration of `Bifrost` [3]. `Bifrost`, as well as `HASHPIPE`[9] and `PSRDADA` [4], uses a ring-buffer system. Memory buffers are instantiated before starting operational steps and placed between them, such that the steps can write to and read from the buffers to move data along the pipeline. `Bifrost` pre-allocates multiple buffers between steps to minimize run-time latency.

The `Bifrost` exploration consisted of a prototype pipeline with 2 different ring buffers: a buffer for visibilities from a measurement set and one for visibilities after flagging and calibration. The size and shape of these buffers were identical, containing all polarizations and as many channels as possible within one time integration. Through this we discovered two drawbacks of using ring buffers for memory. The first applies to when data is moved from one buffer to another. When an operational step moves modified data of the same shape from one ring buffer to the next, this is less efficient than in-place modification. The second concern with using ring buffers are challenges associated with multi-threading: lock contention and memory ownership. With multiple blocks, each with at least one thread, reading from and writing to the same ring buffer, there will inevitably be lock contention. This will lead to slowdowns as both the reading/writing threads wait for the other to finish with the ring buffer.

The new concept for memory storage is a memory lender containing buffers for all data and metadata types, with an API that allows users to call for the sizes and data types needed through a unified call to the memory lender. The API will return a pointer to a physical location in memory of a certain size. By giving the pointer, the framework encourages in-place modification for data refinement steps instead of movement between buffers. This method also lends itself well to singular memory ownership, encouraging only one thread to do all write operations on the buffer - useful to ward off headaches in a multi-threaded environment. This custom method improves upon the pitfalls that are present in the ring buffer framework.

[9]https://casper.astro.berkeley.edu/wiki/HASHPIPE

4.2 Memory Lender Features

The lender is written in C++, a low-level language with custom memory management that has pre-existing software packages that are widely used by the radio astronomy community. As the framework is based on a new idea for a memory paradigm, the custom memory management that C++ allows for is necessary, even though this makes it riskier for the user. C++ can also be used for GPU code, allowing for easier communication between the framework and the software steps it would be linking together.

The memory lender is initialized with a user-configured number of all types and sizes of memory needed, before the execution of the pipeline begins. This will be all the lender-managed memory that is available in the lifetime of the pipeline. This memory is split into data buffers that the user is allowed to fill with new data. During execution, the user calls a lender function, specifying the type of memory that is needed, which is returned from the lender buffers. The lender also provides functionality for the user to keep a list of "filled" buffers: the operate list. These are buffers that have been filled with received or calculated data, but are yet to be consumed by the pipeline. There is, however, only one of these lists available for each type of data to encourage the batching of data modifications.

All buffers from the memory lender framework are managed memory, meaning that users do not need to make or destroy the memory. The buffer keeps track of its own number of references with the `shared_ptr` object from the C++ standard library. Each time a copy of the buffer is passed to another thread or given to the operate list, the reference count is incremented. When the buffer goes go out of scope, the reference is decremented. When the reference count reaches zero, the buffer object will pass the memory back to the memory lender as "free" memory, without the need for user intervention.

When using pipeline memory in other software packages, the user might need the raw pointer to the managed memory. The raw pointer should be used expressly for the purpose of converting buffer memory into other objects for use in radio astronomy software. Using raw pointers in this way will be safe for the user as long as the buffer/raw pointer object go out of scope at the same time. This avoids the memory being freed by the buffer while its still in use by the raw pointer and its object. Calls to functions in external packages should block within each thread so that the `shared_ptr` does not go out of scope before the external function call terminates.

As the framework and system of memory is new, there will be a debugging mode with two features for all users to ensure that they correctly are managing and using the buffers. First, the debugging mode will be zeroing and filling of all data buffers at initialization and free. This will allow the user to catch the use-after-free bug, and understand the life cycle of buffers. Second, the debugging mode will provide the memory lender the set of its own buffer pointers. The lender will then refuse any user-made buffers.

4.3 Pipeline and Orchestration

The real-time data reduction pipeline will be built using this framework to link together different software developed by the community that suit the purpose of reducing the visibilities to image form. The pipeline will receive visibilities, flag them and apply calibration solutions. The calibration solutions will be calculated during the execution of the pipeline as well. The calibrated visibilities will then be put through gridding, imaging, and reprojection. The final product of the pipeline includes a HEALPix [5] image for exoplanet science, a measurement set containing the calibrated visibilities, and the calibration solutions and flags. The output measurement sets can be used for other science cases.

Around half of the operations will be done using community software. One planned use is of Image Domain Gridding (IDG) [6, 7]. As the memory framework and pipeline is within the CPU, this software will copy said memory to the GPU before gridding and applying an FFT. This image will be reprojected into HEALPix, with custom software. Another planned software use is `CASA` [8] for applying calibration. `casacore`[10] will also be used to write to measurement sets.

The memory lender is used in the same threads as the pipeline operations. There will be a thread to listen and receive visibilities to put into the operate list provided by the framework. A function to flag and apply calibrations to the visibilities will be run by multiple threads to parallelize processing of buffers. These flag/application threads will give copies of data - modified visibilities, flag masks, calibration solutions - to different threads to write to disk. The final steps of the pipeline, being gridding, imaging and reprojection, are run by one thread per pixel grid. The thread will own the pixel grid memory and form it into a final HEALPix image.

5 Conclusion and Future Work

We describe the design and implementations of the time-domain science data processing pipelines for the OVRO-LWA. Given the need to experiment with different existing software and parameters and the need for performance, we pursued a two-pipeline approach that prioritizes using existing software packages and minimizes development cost. The offline pipeline is operational, with a load-balancing feature planned using multiple task queues. Implementation of the real-time infrastructure and integration with existing software packages are in progress.

[10]https://github.com/casacore/casacore

6 Acknowledgements

This material is based in part upon work supported by the National Science Foundation under Grant Nos. AST-1654815, AST-1212226, and AST-1828784. This work was supported by a grant from the Simons Foundation (668346, JPG). We are grateful to Schmidt Futures for supporting the Radio Camera Initiative, under which part of this work was carried out. The OVRO-LWA project was initiated through the kind donation of Deborah Castleman and Harold Rosen. Y.H. thanks the LSSTC Data Science Fellowship Program, which is funded by LSSTC, NSF Cybertraining Grant #1829740, the Brinson Foundation, and the Moore Foundation; his participation in the program has benefited this work.

References

[1] Huang, Y. et al., "A Matched Survey for the Enigmatic Low Radio Frequency Transient ILT J225347+862146," *The Astrophysical Journal*, in press, 2022, arxiv:2112.03301.

[2] Anderson, M. M. et al., "New Limits on the Low-frequency Radio Transient Sky Using 31 hr of All-sky Data with the OVRO-LWA," *The Astrophysical Journal*, **886**, 2, 2019, doi:10.3847/1538-4357/ab4f87.

[3] Cranmer, M. D. et al., "Bifrost: A Python/C++ Framework for High-Throughput Stream Processing in Astronomy," *Journal of Astronomical Instrumentation*, **6**, 4, 2017, doi:10.1142/S2251171717500076.

[4] van Straten, W., Jameson, A., and Osłowski, S., "PSR-DADA: Distributed Acquisition and Data Analysis for Radio Astronomy," *Astrophysics Source Code Library*, 2021, ascl:2110.003.

[5] Górski, K. M. et al., "HEALPix: A Framework for High-Resolution Discretization and Fast Analysis of Data Distributed on the Sphere," *The Astrophysical Journal*, **622**, 2, 2005, doi:10.1086/427976.

[6] van der Tol, S., Veenboer, B., and Offringa, A. R., "Image Domain Gridding: a fast method for convolutional resampling of visibilities," *Astronomy and Astrophysics*, **616**, 2018, doi:10.1051/0004-6361/201832858.

[7] Veenboer, B. and Romein, J. W., "Radio-astronomical imaging on graphics processors," *Astronomy and Computing*, **32**, 2020, doi:10.1016/j.ascom.2020.100386.

[8] McMullin, J. P., Waters, B., Schiebel, D., Young, W., and Golap, K., "CASA Architecture and Applications", in *Astronomical Data Analysis Software and Systems XVI*, **376**, 2007.