EPJ Web of Conferences **119**, 25003 (2016) DOI: 10.1051/epjconf/201611925003
ILRC 27

DATA FLOW OF A MULTIPLE INSTRUMENT ON-DEMAND PROCESSING ENGINE WITH PYTHON AND DPLKIT

Joseph P. Garcia[1]*, Edwin Eloranta[1] , Raymond K. Garcia[1]

[1]*The University of Wisconsin – Madison, 1225 W. Dayton St., Madison, WI 53706, USA,*

**Email: jpgarcia@lidar.ssec.wisc.edu*

ABSTRACT

The University of Wisconsin LIDAR Group's High Spectral Resolution LIDAR on-demand data website and processing codebase uses Python to explore coding techniques which facilitate a flexible codebase that is reusable for various outputs, cooperative multi-instrument products, and retains stability and maintainability without hindering dynamic experimentation.

1. INTRODUCTION

The University of Wisconsin LIDAR Group has implemented the High Spectral Resolution LIDAR on-demand data website and processing codebase using Python to be more easily deployable, extensible, and reusable. The processing engine uses a data flow programming toolkit and conventions, known as DPLKit, developed together with other atmospheric research programmers at the University of Wisconsin - Madison. These techniques introduce a natural data flow end-to-end, moving incrementally toward encapsulated data actors with simple interfaces which can be independently tested. The objects created can be rearranged and reused as needed, and allow a researcher to focus on coding just the module he/she wants, unencumbered by data intake or outflow concerns. The system already includes shared input code layers for ease in supporting multiple instrument sources in different formats, resampling filters for gridding multiple sources onto a single coordinate system, image creation, processed file creation, and can be plugged into an interactive UI for active data sources. These demonstrate ways to create modules to replace what would otherwise be single-use ancillary code. The website, hsrl.ssec.wisc.edu, uses networks of actors to process data on-demand from HSRL deployments since 2003. It also includes an expanding codebase for advancements in instrumentation and processing, such as integrating products from co-located instruments in order to create new innovative measurements.

2. UNDERSTANDING FRAME STREAMS.

Many traditional scientific processing engines operate on files directly, neglecting proper handling of file borders and I/O optimization until it has already infiltrated the science code. In order to separate these tasks while addressing both real-time and retrospective batch processing, a more generally versatile approach to the task interfaces is necessary. The fundamental idea of data flow with DPLKit (DPL stands for "Data Pot-Luck", after pot-luck parties at which guests each bring a dish to share) involves thinking of data as a unidirectional flow of measurement frames in time. I/O levels need only retrieve data in chronological order, presenting every time step once and only once. File borders disappear from functional code, leaving only more approachable and repeatable framing boundaries. The specific content of the frames themselves is flexible, and only relevant to the narrow requirements of the task consuming them. Any repeatable task operating on these frames may be broken out into its own "actor," which also permits the frames be from any actor capable of making compatible structure, whether or not the two actors are familiar to each other. Each actor in the pipeline advertises what its output frame will contain as a dictionary attribute, available to the immediate downstream actor at initialization time. In adapting existing code, the manner in which actors are chained together may start out as fixed, but will evolve into a more shared interface as new actors are dynamically added that plug in anywhere along the pipeline, without other actors' awareness.

The only necessary actors for any new data source are one to discover and catalog data URIs from a storage source, referred to as a Librarian, and one to convert URIs into frames of ordered data,

referred to as a Zookeeper. The separation of these two tasks simplify both, since it allows them to focus on the filesystem and file content separately, and either can change without impacting the other.

3. USING PYTHON: ACTORS AS PYTHONIC GENERATORS

The natural implementation of repetitive processing is a for-loop, which for our purposes means using iterable objects. The fundamental idea here is that the object will read or synthesize a frame with each step, tracking its own state along the way. Far more intriguing, Python generators add elegance by creating an iterable object with little code overhead in what deceivingly looks like a stateless function. In Python, the simplest actor will simply iterate upon its source (upstream) actor, complete its task on each frame, and "yield" the product for the downstream actor. All state of the loop is maintained by simply being a generator.

Any parameters needed by the actor should be provided at initialization time. When an actor is initialized, its parameters and source actors' frame descriptions allow it to configure itself completely, as well as present expected errors at that moment instead of mid-operation. All runtime state is maintained within the actor as necessary, or within the actor's __iter__ function scope when operating as a generator.

Because iterators are allowed to operate in a disjoint rhythm, more complex actors may accumulate multiple frames to operate on each frame in the context of those in close proximity in time and space. These actors would appear to yield a time-deferred product when compared to the input frames, but this is not at all a problem so long as chronological order is maintained. Furthermore, an actor can accumulate and assemble multiple frames into a compound-frame or a single high-level observation as its product. For example, a scanning instrument can have one stream of native resolution data frames, usable for status content, while also feeding into a separate actor that stitches together whole scans into complete images.

4. MODULAR DESIGN BENEFITS

By separating tasks into different actors that flow data in one direction, operating in its own context, we have also effectively created an efficient computational pipeline. Minimization of side effects from cross talk and backwards flow between actors makes it more feasible to parallelize their tasks. Output of one actor can be diverted to another thread or process, or entirely separated within an actor. By integrating a method of transport and resource management, this becomes a clear direction for improved efficiency.

With each actor doing a very deliberate and finite task, generalized modules can be written with full focus on the purpose of that module. This avoids duplication and allows you to benefit from robust implementations, having only to provide the proper initialization to get correct results.

Building upon this, the actor can also operate by calling any separately developed code. Heritage code, library functions, and scientific code written by an expert can be interfaced with, as Python already has many community libraries and language bridges. Actors can serve as a sandbox for any environment or scientific workspace.

5. STEERING AND CONTAINING DIVERGENT CODE EVOLUTION

In the process of adapting and separating existing code into DPL constructs, discrete tasks will have a tendency to evolve in place. When these segments are found to be useful to more than one role, they can be extracted into library functions and used where needed. This allows existing modules to maintain the same stable and mature functionality, while new development code can reuse these functions, extending the API minimally to be more use-agnostic. This approach keeps dramatic changes localized without compromising core functionality. Properly allowing development and experimentation to independently diverge from stable code produces a much more approachable and versatile system as a whole. In time, the experimental code can stabilize and mature to contribute its own reusable functionality back into the stable system.

6. SEPARATION OF TASKS IN PRACTICE: HSRL

HSRL calibration routines evolved into four distinct actors: instrumentation constants,

instrument calibration tables, atmospheric profiles, and computed calibration vectors. Instrumentation constants consist of simple values that describe the instrument, which can be necessary for proper handling of both raw and inverted HSRL data. Calibration tables are more complex tables generated from properties of the instrument. These are combined with the measured atmospheric profiles to compute the calibration vectors necessary for processing HSRL raw data. Atmospheric profiles initially came from RAOB radiosonde files, which have a specific actor for reading them into a stream of individual profiles. Later, a virtual radiosonde system (VRS) using the GFS archive and forecast data became available, so an actor that streams and adapts these profiles to resemble the radiosonde source was written. For a mobile platform HSRL, the VRS profiles need to track the instrument location. This location information is stored in the HSRL raw data files, so the HSRL's raw data stream actors are reused here, adapted to serve as location input for the VRS stream. From this, the code generating the calibration vectors gets a stream of atmospheric profiles at semi-regular intervals, determined by the VRS from either significant time or position changes, without any modifications in the calibration synthesis code.

In a lab environment, overriding the production default calibration task for experimental purposes is sometimes necessary. To facilitate this, an additional actor can be injected on demand between the deployable instrumentation calibration tables and the calibration vector computation. This actor can be configured to alter or completely replace tables in any way the scientist desires. Because this filter is simply omitted from the production data flow, the system doesn't suffer any additional complexity from unused non-production code, while allowing any means of modification useful to the scientist.

Downstream from the calibration vector calculation, an actor receives these vectors and resampled raw HSRL data, and is solely responsible for the inversion task.

After the HSRL processing itself, output frames can be streamed into an actor designed for batch plotting, storage, or a live UI. For housekeeping plots, only the raw data is needed, so processing actors can be omitted entirely in that case. The same actors can also be used for custom extraction tasks, selecting specific fields to be exported to a CSV file for one-off analysis, for example. With a live updating UI, the actors can be configured to continuously loop instead of terminate on the last slice. The processing code can operate on simply the newest slice of data, allowing it to be combined in another actor tasked with caching old data. The result is a rolling window of compound data, which can be connected to a constantly updating display. Accuracy of the content is ensured since all methods use the same common tested backend.

7. COOPERATIVE PRODUCTS

With multiple HSRLs deployed in different platforms, often with other instruments, it becomes extremely appealing to be able to integrate other datasets into the data flow. Already, the processing platform includes the ability to read processed data from several co-located radar systems for the purpose of re-gridding to a common set of axes, and creating convenient runtime observations that don't already exist in the source files (radar backscatter, for example). These can be used in combination with HSRL data to produce cooperative observations. The computation tasks are compartmentalized into actors that focus on only that task, without complications from disk access, data flow, resampling, or making any assumption about the where the data is to be used down-stream.

Currently, actors exist that combine HSRL and radar data to calculate particle measurements and rainfall, as well as actors that compare these results to measured observations from other sources. As more datasets are introduced to the processing system, additional actors can be written to produce new cooperative results in much the same style.

8. GRAPH DISPATCH ENGINE

As processing engines grow in size and complexity, connecting actors in an ad-hoc fashion becomes inefficient. Structural duplication of actors and resource use become a serious problem to scalability. To help control this, a reusable many-to-many graph construct is being

developed, where actors can be added and reused via branching in order to avoid multiple instances of the same actor. With further research, this implementation can use its awareness of the entire graph structure to configure, optimize, and dispatch a multi-core pipeline, allowing actors to operate in parallel, with data-flow between processing contexts managed intelligently.

In the process of adapting our actors to work robustly a graph solution, we did discover that actors should adhere to a set of expectations if the output is to be reliable. The most important is that frames and actor attributes must be immutable. What this means is that once an object is created, the exposed content and structure stays the same for the life of the object. Immutable frames allow any single frame object to be shared across many actors and contexts, without the ability of another actor to alter it unexpectedly. Immutable actor attributes are useful to actors downstream, as it allows them to acquire some parameters from upstream actors, making their initialization simpler and consistent. For example, our cooperative actor that operates with both HSRL and radar needs system parameters and constants that aren't part of the data stream itself. Actors tasked to each instrument exposes these parameters as attributes, becoming available down stream to any actor. Hypothetically, if these attributes are allowed to change, a parallel processing system would require complex synchronization between actors or suffer from unpredictability. Any actor's attributes that do need to change as the task progresses must be provided within its frame stream, or via a separate sparse frame stream and actor. The latter is more appealing, as the attributes usually constitute useful meta-data that shouldn't depend upon other complex tasks to acquire. For HSRL, this is why the instrumentation constants became it's own stream, available to the raw data and the calibration tables, without depending upon either. With this layout, dynamic parameters are available to any actor, allowing the parameters to be used independently, or synchronized properly to another dependent data stream.

ACKNOWLEDGEMENT

This work was partially supported by NSF Grant# ARC-0946359.