The LSST Pipeline Framework: the General Design

Abstract

The Pipeline Framework is a set of classes for assembling pipelines out of reusable scientific processing modules. An important purpose of the framework is to separate application concerns--that is, how an algorithm is applied to some data--from middleware concerns, like how the processing is distributed over multiple processors and how data is read from and written to storage, as these latter concerns can depend greatly on how the pipeline is deployed on a particular platform. The framework provides application developers a software container for hosting an algorithm (primarily, the Stage class), creating a module that can be connected with other modules to build a pipeline. The framework should maximize flexibility in how modules can be assembled as well as in how pipelines are optimized for parallel processing and data I/O. Finally, we recognize that a single pipeline model may not work well for all application cases; thus, we must allow for implementations that "break out" of the standard pipeline model and take more control over parallel processing concerns and I/O. This document describes the design of the LSST Pipeline Framework and how it executes astronomical processing modules in a pipeline. It emphasizes how the framework is *designed* to work--not how it currently works (as implementation is on-going as of this writing).

Background: Design Origins

The origins of the pipeline framework design comes from a collaborative, UML-based modeling effort (with contributions from across the LSST DM team). This effort is captured in a UML model (available via UMLModel in PDF format). Two versions of the model are currently maintained: a ?general LSST model (LSSTModel) that captures the reference design, and a ?DC2-specific model (DC2Model) that captures the DC2 implementation of the design. The origins of the design are found in the Use Cases for pipeline processing (?LSSTModel, Middleware Use Cases, starting around p. 308). A few Use Cases in particular explore how a pipeline operates:

- Run a Pipeline (?LSSTModel, p. 369),
- Pipeline Execution and Monitoring (?LSSTModel, p. 367),
- Execute Processing Stage (?LSSTModel, p. 364)

The Use Case modeling led eventually to a logical class model for a pipeline framework (Processing Framework Summary, ?LSSTModel, p. 453) and refined for implementation in DC2 (Pipelines and Stages Data Model, (?DC2Model, p. 362).

Some general concepts that came out of this modeling effort include:

- A *Pipeline* is made up of one or more *stages* in which each stage executes one logical step in the pipeline.
- *Pipeline Construction* refers to the process of building a pipeline out of pluggable components (stages).
- Pipelines are configured to operate on a particular set and/or stream of input on a particular set of hardware platforms.
- The primary mechanism for configuring a pipeline is through a *Policy*--a set of parameters that can be captured in flat files or are built up from a library of Policies. Policies provide a broad range of configuration settings from parameters that control a Pipeline as a whole down to parameters that control fine details of the behavior of some astronomical algorithm. A policy library captures default values for these parameters which can be overridden for a pipeline configured for a particular

execution.

- *Pipeline Orchestration* refers to the scheduling, execution, monitoring, and clean up of a Pipeline.
- A pipeline stage will typically be parallelized by having each processor operate on a different (preferably independent) chunk of the data.
- The prototypical stage executes in three steps (?LSSTModel, p. 364):
    1. a serial setup step
    2. a parallel processing step where the main computation is done
    3. a serial post-processing or clean-up step. A common use of this step would be to record the *provenance* (the metadata describing what was done) of the stage.

Another concept to come out of this modeling effort is the notion of a *slice*. An important concern for optimizing a pipeline is minimizing I/O costs. One way to save on I/O costs is to keep data that is used by multiple stages in memory. For example, the image processing pipeline might be parallelized by processing each CCD from the focal plane array on a different processor. On one particular node, a CCD image can be kept in memory for as long as it is needed by the chain of pipeline stages. This implies then that a single processor will host a sequence of stages that operate on the same CCD (or set of CCDs). A *slice* of the pipeline, therefore, is that set of stage instances (running inside a worker process) that will operate on the same data-parallel chunk of data.

The Components of a Pipeline

The Pipeline Framework is characterized by three main components (represented in software as software classes):

- *Pipeline*: a manager of a sequence of processing stages that run over multiple, parallel processes. It is instantiated only in a "master process", and it controls the flow of data and executes the stages in proper order.
- *Slice*: a host for executing the stages in a single parallel "worker process", operating on a data-parallel chunk of data. One Slice object is instantiated in each worker process, which are distributed across multiple processors (one differenct machines).
- *Stage*: a container for a processing algorithm that handles one logical step in a pipeline. The Stage class itself is abstract; it is sub-classed and implemented to apply a specific algorithm to data. The Pipeline instance contains instances of each Stage class that makes up the pipeline. Each Slice also contains instances of each Stage class.

Figure 1 shows *one way* these classes might be deployed on in a 2-way pipeline. A single master process hosts a Pipeline object which has references to instances of each of the stage objects handle each stage of the processing. Each worker process hosts a Slice object with also has references to its own set of stage objects. The master communicates with each of the Slice objects (which may be on different machines), telling it when to execute each stage; in our design, that communication is done using MPI messages.

Figure 1. The deployment of the pipeline classes into processes. The master process hosts the Pipeline object, while each worker process hosts a Slice object.

The Stage class API includes three methods representing the three steps of stage execution:

- *preprocess()*: a serial set-up step
- *process()*: the parallel computation step. This is implemented to operate on one data-parallel unit of data (e.g. one CCD from the focal plane array).

- *postprocess()*: a serial clean-up step

The serial methods are executed on the Stage instance in the master process, and the parallel `process()` method is executed on the Stage instances in each of the worker processes. In particular, the Pipeline and Slice objects can use the Stage API to execute the stage in the following way (ignoring for the moment data handling issues):

1. When new data is available for the first stage, the master Pipeline object will call the `preprocess()` on its instance of the first Stage.
2. After the `preprocess()` is complete, the Pipeline object sends an MPI message out to all of the worker processes telling them to execute the parallel portion of the first stage.
3. The Slice object in each worker process receives the message. It then calls the `process()` method on the first Stage object.
4. After the `process()` method completes, each Slice object sends an MPI message back to the master process indicating that the parallel processing is done.
5. When the Pipeline object has received the messages from all the worker nodes, it can execute the `postprocess()` step on its instance of the first Stage.
6. When the `postprocess()` is done, the Pipeline object can repeat this sequence for the second stage, and so on, until the entire chain of stages is complete.
7. The Pipeline starts the sequence again beginning with the first stage for the next set of available input data.

Note that a Stage implementation need not provide implementations of all three processing methods. In fact, it is likely that most stage implementations in DC2 will not include `preprocess()` and `postprocess()` implementations. Also note that the above sequence need not be the only one implemented by the Pipeline and Slice objects. For example, if it is known that the serial steps are completely independent of the parallel steps (i.e. without any information sharing), the Pipeline may make certain optimizations. For example, it could make all of the `preprocess()` calls for all the stages all at once at the beginning of a pass of data through the pipeline and all of the `postprocess()` calls at the end of the pass after all of the parallel executions for all of the stages. In this way, this would allow the synchronization barrier at the beginning and end of each stage to be removed: a Slice could process its data-parallel unit of data through all of its stages without waiting for the corresponding stages in the other worker nodes to complete.

Finally, we note that it is not required that a slice include all of the stages of the pipeline. That is, the stages of a pipeline could be split over multiple worker processes. This will be useful when the axis or granularity of parallelization needs to change between stages (e.g. cross-talk correction).

Passing Data and Information

Figure 1 shows that Stages are stitched together via *Queue*s which provide the channel for passing information and data from one stage to another. As illustrated in Figure 2, a Queue has two halves to its interface: an *InputQueue* and an *OutputQueue.* A Queue connects two Stages together by attaching the OutputQueue half to the preceding Stage and the InputQueue half to the following Stage; this is done via the Stages' `initialize()` function in which references to the InputQueue and OutputQueue that the Stage should use.

Figure 2. The Pipeline Framework Class Model, highlighting the class relationships and the most important parts of the API.

A Stage implementation uses the InputQueue interface to get its input data. In particular, it uses the `getNextDataset()` to retreive a Clipboard object. A Clipboard provides a by-name access to references to data objects of arbitrary type. This can include primitive type values (e.g. a floating point number called "searchRadius") or complex objects (e.g. an Image type named "biasImage" or a DataProperty type named "fieldOfView"). The Stage pulls the data it needs off the Clipboard (via its `get()` function), it applies the target algorithm to the data, and then it posts the output data back to the Clipboard, giving each item a name. Finally, it posts the Clipboard to the OutputQueue. The Queue implementation is responsible for getting the data on a Clipboard posted to an OutputQueue to the InputQueue of the next stage. Note that the Clipboard may contain data posted to it by a previous stage that is never touched by the current stage. Note also that no assumption is made about who put a piece of data onto the Clipboard, whether it was the immediately preceding stage, some earlier stage, or the managing Pipeline object. Thus, any stage is able to make its outputs available to any downstream stage, where ever it appears.

Obviously, if the one stage is going to use as input data that was output from the previous stage, the two stages must "agree" on the name to give to that data. We envision this aggreement coming about in one of two ways. First, one stage may be developed assuming that another stage was already executed previously. The stage developer will consult the documentation for the prerequisite stage to get the names of the output data items and will implement the new stage to look for the data under those names. In other cases, it's not necessary to make any assumptions about what has happened previously. Thus, we expect that we will need to define a small set of standard names--a data model for input and output data--that should be used for posting data playing a common semantic role in a pipeline. One example might be "primaryImage" which could be the name that points the current target image being operated on. For early image processing stages this Image might in fact be a raw, uncorrected image; however, stages may replace the Image object with corrected versions. As a result, the Image held under this name in the latter stages may be highly processed. Another standard name might be "fieldOfView" which may give a description of the observational footprint of the current observation being processed.

As mentioned previously, how the data on the clipboard gets moved from an OutputQueue to an InputQueue depends on the Queue implementation. Figure 2 shows that a Queue can be implemented in different ways, and the choice of which Queue to use depends on how the Stages being connected are distributed. In the simple case where the two Stages will be executed on the same node in the same address space, a simple Queue (e.g. InMemoryQueue) can keep the Clipboard in memory and pass the reference from the output side to the input side. If the Stages are on different machines, a different implementation can be used that either caches the data to shared disk (e.g. CachingQueue) or passes the data via MPI communication (e.g. MPIQueue).

Note that each process, whether it is hosting a Pipeline or a Slice, will have its own instances of Queues and the Clipboard that will pass through them. The Pipeline and Slice objects are responsible for populating the initial contents of the Clipboard. For some of the data, identical copies will be placed on all of the Clipboards that will be handed to the first stage. For some of the data, however--namely, the image data--each Slice may place a different data-parallel chunk the data on the Clipboard. Note that by default, the Clipboards held by the Pipeline object and the Slices are all independent. In some cases, however, a Stage implementation will want to share values across Clipboards. In particular, data generated during the `preprocess()` execution may be needed in the `process()` step (which will be run in a different worker process). In this case, when the `preprocess()` function posts the data to Clipboard, it can tag it as being *sharable* (via the `sharable` parameter to the `put()` function). After the `preprocess()` function completes, the Pipeline object will look for data marked as sharable and broadcast it to the Slice objects (as an MPI scatter message, incorporated into its message telling them to execute the `process()` function). The Slice object will then put the data onto the Clipboard before calling the Stage's `process()` function. Similarly, if the parallel `process()` execution produces data needed by the serial `postprocess()`, that data is also marked sharable, and it will get sent back to the Pipeline object via an MPI gather message.

Note that there is no provision in the model (as of yet) that allows parallel Slices to share data with each other (other than via a combined gather-scatter operation described above) as this breaks the data-parallel processing model and require implementing the MPI communication patterns. If this is necessary, a Stage implementation would be given access to the MPI capabilities more directly.

Pipeline and Slice objects are responsible for initializing the Clipboard at the beginning of the Pipeline. They are also responsible for taking any data remaining on a Clipboard when it arrives on the OutputQueue of the last stage and persisting it according to the Pipeline's policy. (In general, data on a clipboard could be persisted between any of the stages.) Some of the data on Clipboard will be destroyed while other data can remain to be available on the InputQueue of the first stage again. In this way, the Clipboard can keep some information from previous traverses through the Pipeline. The work of cleaning up a Clipboard at the end of a pipeline and its re-initialization for use at the beginning will be delegated to a special implementation of a Queue. In particular, the special Queue implementation that will handle the end points of the Nightly Processing Pipeline will be responsibility for receiving the new images from the telescope, instantiating them as Image objects and placing them on the Clipboard as input to the first stage. How the data is received is not specified in the framwork design but is rather encapsulated in the special Queue implementation (it may monitor some directory or it may respond to an event that points it to new data on disk).

Thus, we envision the Clipboard as an object that is initialized for the first stage and then travels through stages (via the Queues), picking up new data items along the way. At the end of the pipeline, output products are saved and the Clipboard is re-initialized for the next set of data to be processed.

Inter-pipeline Communication with Events

We envision that Pipelines will occasionally need to communicate asyncronously with external processes. In particular, we have a use case in which nightly processing is handled as a set of separate and loosely coupled pipelines, and that these pipelines will neede to pass information between them to, say, coordinate their activities. This type of communication is handled via the Event system. Like other aspects of the middleware functionality, we would like to minimize a Stage's responsibility for handling the passing of this information via events; nevertheless, if the Stage needs to take more control it can access the Event API directly.

In most cases, it will not be necessary for a Stage to interact with the event system to get data that is delivered by another system via an event. If a Stage needs some event data, the Pipeline can be configured to listen for the particular event; when the Pipeline receives the event it can put the data onto the Clipboard prior to executing the Stage's `preprocess()`. Furthermore, the Pipeline can be configured to wait until an event arrives before executing the Stage, if desired. If this data is needed by all of the worker copies of the Stage, the data item can be tagged as *sharable* and distributed to the Slices like any other item on the Clipboard. By making the Pipeline responsible for receiving the event and extracting the data, we can ensure that all of the copies of the Stage have a consistant set of input data and, therefore, behavior.

At the moment, we do not have a general approach sending out data via an Event from a Pipeline. Thus, for now, we would allow a Stage to use the Event API to publish events directly. In most cases, this would be done from one of the serial steps (`preprocess()` or postprocess()`) so that only one Event is issued by the Stage via the master process.

The Full Pipeline Sequence

Now that we have discussed how the pipeline framework manages the data flow, it is worthwhile to revisit the operation sequence presented in the introduction to the pipeline components above.

1. The Pipeline and the Slice objects are created and configured according to a given Policy. As part of the configuration, the Pipeline notes the events that it needs to receive and when. The Pipeline initializes a stage position counter, *N*, that indicates which Stage is to be executed next, setting it to first stage.
2. The Pipeline object tells the Queue between stage *N* and *N-1* to transfer the Clipboard from the output side to the input side. For the first stage (when the *N=1*), this means initializing the Clipboard with the new data for the Pipeline.
3. The Pipeline object receives any events it is configured to listen for prior to the execution of the stage *N*. If so configured, it waits until that event has been received. Upon receipt, it places any data contained in it on the Clipboard for stage *N*.
4. The Pipeline object calls the `preprocess()` on its instance of the Stage *N*.
5. After the `preprocess()` is complete, the Pipeline object inspects the Clipboard on the Stage's OutputQueue for sharable data. It then sends an MPI scatter message out to all of the worker processes telling them to execute the parallel portion of stage N; the message is followed by a serialization of any sharable data.
6. The Slice object in each worker process receives the message. It tells the Queue between stage *N* and *N-1* to transfer the Clipboard from the output side to the input side. It then unserializes any sharable data items it receives from the master process and places them on the InputQueue for the stage *N* (with the sharable tag turned off). It then calls the `process()` method on its instance of the Stage *N* object.
7. After the `process()` method completes, each Slice object inspects the Clipboard on the Stage's OutputQueue for sharable data. It sends an MPI gather message back to the master process indicating that the parallel processing is done; the message is followed by a serialization of the any sharable data.
8. When the Pipeline object has received the messages from all the worker nodes, it aggregates any received sharable data items and places them back on the master Clipboard and posts it to the InputQueue of stage N. It then executes the `postprocess()` step stage *N*.
9. When the `postprocess()` is done, the Pipeline object can repeat this sequence beginning with step 2 for stage *N+1*, and so on, until the entire chain of stages is complete.
10. When the `postprocess()` of the last stage is done, the Pipeline resets its stage position counter *N* to the first stage, and starts the sequence again (at step 2) beginning with the first stage for the next set of available input data. Encapsulated in the transfer of the Clipboard from the end of the pipeline to the start is the persisting of any data items left on the Clipboard, according to the Pipeline policy.

Configuring a Pipeline and its Components

A Policy Object will be used to configure the Pipeline. As described in PolicyDesign, the Pipeline Policy will contain the configuration for the Pipeline as a whole as well as for all the component Stages and the components internal to the Stages; this information is organized into a hierarchical tree. It is a function of *Pipeline Construction* to assemble the comprehensive Policy out of default policies for each of the Stages; that is the pipeline construction system would pull default policies as flat files from a policy library, override their paramters as desired, and assemble them into one comprehensive policy file. It is a function of *Pipeline Orchestration* to deploy that comprehensive policy file on each node that will run a pipeline process.

The kinds of information that would be configured at the Pipeline level include:

- the stages to be run
- the topology of the pipeline: how stages are distributed across processors
- the Queue implentations to use; in particular, the Queue class that handles the input at the beginning of the pipeline and the output at the end.

Queues would have policies associated with them that control how they pass the Clipboard data from the output side to the input side. The Queue handling that start and end of the Pipeline in particular would require policy parameters that control:

- how to receive and instantiate data for placement on the InputQueue of the first stage
- what data from the OutputQueue to persist

The individual Stages would each have policies associated with them. These would mostly be specialized to the particular behavior of that Stage; however, some parameters could provide hints to the Pipeline about how the Stage is implemented (e.g. does it provide `preprocess()` and `postprocess()` implementations, will there be sharable data) which the Pipeline can use to optimize the execution of the Stage.

Implementing a Stage

Application-level classes that encode some algorithm are made available to a pipeline by wrapping them in a Stage implementation. That is, the application developer subclasses the Stage class to make a Stage that applies a particular algorithm. For example, the developer may create a class called ImageSubtractionStage?. The developer must provide implementations of the virtual functions, include `preprocess()`, `process()`, and `postprocess()`. (In DC2, many Stage implementations will not need to implement the serial steps). The `process()` implementation encodes the logic for applying the algorithm to the input data retrieved from the Clipboard on the InputQueue. This could be a "thin" implementation that mainly wraps a single class that does the work or a "thicker" implementation that glues together several classes together to apply the algorithm.

In DC2, Stages are implemented in Python.

In practice, one of the first things the stage developer will need to do is determine what data it expects to find on it InputQueue and what data it will provide on its OutputQueue. Each data item will be given a name according to the heuristics described above. The Stage documentation must include a full enumeration of these data items.

The Stage developer will also need to determine what configuration information it will need via a Policy. This will be documented by creating a Policy Dictionary file (see PolicyDesign). The Stage developer implements the Stage's `configure(Policy)` method to pull data out of the Policy by the names defined in the Dictionary. Some of those names may return to Policy objects that can be passed to instances of classes the Stage uses internally.

Testing a Stage

Supporting Different Pipeline Topologies