Reference software implementation for GIFTS ground data processing

R. K. Garcia*, H. B. Howell, R. O. Knuteson, G. D. Martin, E. R. Olson, M. J. Smuga-Otto
University of Wisconsin-Madison Space Science & Engineering Center,
1225 W. Dayton Street, Madison, WI, USA 53706

ABSTRACT

Future satellite weather instruments such as high spectral resolution imaging interferometers pose a challenge to the atmospheric science and software development communities due to the immense data volumes they will generate. An open-source, scalable reference software implementation demonstrating the calibration of radiance products from an imaging interferometer, the Geosynchronous Imaging Fourier Transform Spectrometer[1] (GIFTS), is presented. This paper covers essential design principles laid out in summary system diagrams, lessons learned during implementation and preliminary test results from the GIFTS Information Processing System (GIPS) prototype.

Keywords: Data Processing, Scalable Software, Fourier Transform Spectrometer, GIFTS, Cluster Computing

1. INTRODUCTION

Work at the University of Wisconsin Space Science and Engineering Center (UW SSEC) continues toward building high-performance prototype algorithms and processing systems for candidate meteorological observation platforms such as the Advanced Baseline Imager (ABI) and Hyperspectral Environmental Suite (HES). Current work focuses on support of laboratory tests and inter-comparisons for the GIFTS instrument currently being evaluated at the Utah Space Dynamics Laboratory. This paper documents the rationale, design and technologies used in a prototype implementation of a scalable test data processing system for meteorological satellite imaging interferometers such as GIFTS.

Hyperspectral imaging instruments are distinguished by their high throughput rates, often on the order of 100 megabits per second. The GIFTS system design specifies the capability to produce images of 128 by 128 pixels, each having 3072 complex interferogram sample points, once every 10 seconds. This translates to 1.5 Terabytes of data a day, assuming continuous operation.

The processing of raw data produced by such instruments is usually spatially independent, but has temporal dependencies stemming from the need to maintain a dynamic calibration context. These dependencies exist because the instrument is unable to observe its target (an 'earth' scene) and its calibration sources (blackbodies) simultaneously. For ground-based and atmospheric interferometers, there are typically two such sources, each held at a preset stable temperature. In the GIFTS design, a third "deep space background" view is included to complement the two small internal blackbodies, a design that is more compact than the alternative – a large external calibration target.

Latency constraints govern the generation environmental data records such as temperature and water vapor profiles, trace gas analyses, and extraction of wind vectors. In the case of GIFTS, this time limit is on the order of 15 minutes.

Developing a systems approach to processing such a massive quantity of data with near real-time constraints requires an eye to efficiency at every layer of the system.

1.1 Requirements

A multitude of sometimes conflicting requirements need to be reconciled in order to proceed with design and implementation of a processing system. A variety of experimental algorithms and algorithm variants must be tested, refined, and re-tested. Ground testing and characterization algorithms for the GIFTS instrument need large ensemble studies to test for sensitivity to various tunable parameters. Candidate algorithms for operational use need to be demonstrated, verified and validated in a form showing scalability. This includes algorithms to generate sensor data records (e.g. absolute earth radiances) as well as environmental data records (e.g. retrieved temperature profiles). The algorithms must be accurate to within the program goals, and must fit within computing system memory and time budgets.

Furthermore, these algorithms may be prototyped in a variety of programming languages, and may consist of a mix of "read-compute-write" applications and DSP-style in-memory operations. Integrating these various kinds of code in testing is desirable.

It can be expected that algorithms will contain substantial amounts of control logic. There will be temporal dependencies, mutable context information, and context selection from large databases. It is critical to coherently handle this while recording exactly which data and code entities were used to derive a given product. Implementation of past systems and test algorithms for GIFTS has shown that flow-of-control quickly becomes at least as complex as the actual numerical details of the algorithms. With this in mind, separation of control logic from data processing is desirable, as well as migration of state information to high-level control logic where possible.

Recording data provenance is a priority in order to retain provably valid data. A complete description of processing history contained in metadata enables repeatable reprocessing and ensemble testing is needed.

1.2 Constraints and Assumptions

Several constraints and simplifying assumptions on the system design were also included in order to keep the system design tractable.

The system must be implementable on existing architectures available to a university research center such as UW SSEC. These include small- to medium-scale multiprocessor shared memory systems, or small-to-medium scale cluster computing systems. A multiprocessor system design will be needed in order to solve the problem given the throughput and latency constraints. Furthermore, the granularity of the data processing must be sufficiently small to meet latency requirements when scaled to a production system.

Algorithms can be broken down into software “capsules” which adhere to functional programming principles, i.e. given the same set of input data blocks, any copy of a given pipeline capsule will provide the same output. In order to meet a real-time requirement, there will be limits on iteration and feedback within the science pipelines.

Not all intermediate data need be archived so long as reproducibility of the calculation is maintained. Finally, an archiving and distribution system of sufficient volume and capability will accept the output of the system and make it available for analysis.

2. DESIGN OVERVIEW

As presented in previous papers[2,3], the design concept we have adopted has the following major characteristics.

• The input service is responsible for accepting external data such as a satellite data feed, performing any necessary and minimal preprocessing in order to segment the data into well-sized pieces, and extracting sufficient metadata regarding those pieces in order for control logic to determine its routing through the computing system.

• The master is responsible for manipulating metadata on the input data, taking into account the known capabilities of the slave node array, as well as any system and application specific information such as load levels and cost estimates. It constructs work orders that provide simple instructions for the rest of the system to follow in order to process and export one or more chunks of input data.

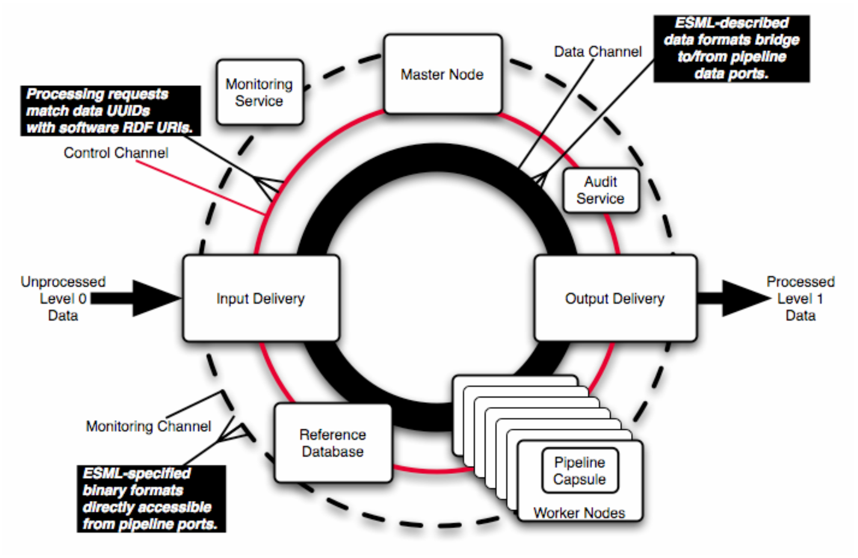

Figure 1. The GIPS target architecture diagram

• The reference service provides a persistent clearinghouse of algorithmic seed data as well as any context data that is likely to be reused within the system. Examples of seed reference data are offline calibration files, projection matrices for atmospheric state retrieval, pixel masks, lookup tables, and design characterizations such as Monte Carlo simulation generated emissivity profiles. Context data includes such things as periodically updated numerical fits and predictors for calibration sources.

• The audit service records the history of processing, providing reports that can be embedded in output files.

• The monitoring service accepts real-time processing system feedback, showing pertinent summaries of engineering performance and science summaries useful for system operators.

These elements are connected by a group of abstracted communication channels:

- The control channel manages the transference of data properties, localities, and work orders throughout the system. It is principally managed by the master.
- The data channel is responsible for bulk data transfers and is optimized to throw simple blocks of data far and fast.
- The audit channel funnels processing history information into an audit database.
- The monitoring channel takes tagged values and feeds them to the monitoring service for display.

2.1 Theory of Operation

At startup, the slave, input and output nodes determine which application pipelines they will be supporting, then load and initialize them. The reference service provides any initialization data needed. The master loads the application strategy module responsible for planning data movement and computations, retrieving any stored state from the reference or audit databases.

When ingesting data, The input service cycles the application input pipeline. A property list describing the important features of the data, including the UUID names assigned to its components, is forwarded to the master.

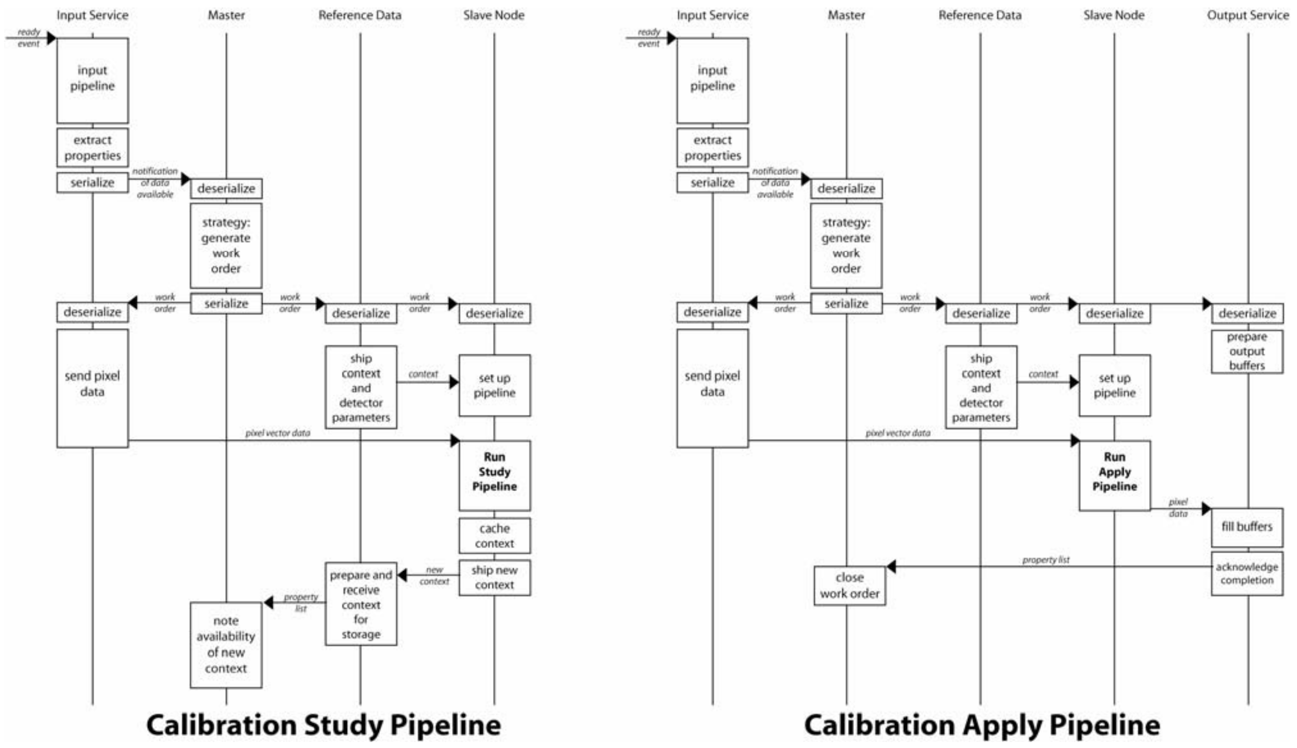

Figure 2. Sequence diagrams for calibration study and apply pipelines.

To generate work orders, the master invokes its strategy capsule on the property list. The strategy uses its known system state and performs any necessary cost estimation on alternatives for processing the data. It then creates any number of work orders, which are queued to the control channel for broadcast.

By managing data movement through the system in a prescribed fashion, direct interaction between slave nodes is minimized. Each node in the system is effectively following a script that the master has optimized to the available resources.

The bulk of the system's time is spent marshalling data and performing computations. The nodes within the system walk through the work order, extracting for their own use the processes and data elements that are local. Data objects that are already present locally and represent dependencies of processes to be performed elsewhere are queued for transfer via the data channel. Data objects that are available remotely and are needed by a local pipeline are queued to the data channel as incoming transfers. Once the dependent data are available locally for a given local work order step, the appropriate processing pipeline is invoked. Locally created outputs named in later work order steps are queued to the node or nodes needing them. This sequence of events is represented as a UML sequence diagram in Figure 2.

The final steps of a work order specify sending data to the output service, and composing audit report information with final data formatting for export to an archive and distribution facility.

3. TECHNOLOGY CHOICE

A variety of technological choices are available to realize the above design. First off, we must be able to aggregate and configure the system as a whole from a set of available computing facilities. Within that system, we must be able to shuttle control information, the data itself, as well as monitoring and auditing records from point to point. Several technologies that fill the needs of different parts of the system were evaluated at the UW SSEC.

3.1 The Control Channel

The control channel sustains comparatively low traffic, shuttling informative metadata to the master and work orders from the master. Candidate technologies for a control channel include:

• Message Passing Interface (MPI): A scientific computing middleware library that allows asynchronous message passing between nodes of a multiprocessor system. It lends itself marginally to this purpose, since the data structures – property lists and work orders - that are sent along the control channel must be serialized in order to move through MPI.
• TCP sockets: Raw POSIX sockets have the benefit of having well-specified and well-understood behavior, efficient implementations, and high portability. However software to provide system aggregation, fault tolerance, and messaging protocols must be built from scratch.
• Queuing system: A queuing system could be used to distribute work orders and property lists through the system in the form of scripts. However, since multiple nodes may collaborate in executing a single work order, each work order would have to be queued to at least the nodes involved in that work order. This implies that a work order would translate to multiple jobs with dependencies coordinated by the queue manager. The attendant protocol overhead may be mitigated if the processing tasks are broken up into sufficiently coarse-grained pieces.
• SQL database server with signaling: The use of an SQL database server to manage and distribute work orders invites the blending of function between the audit database and the control channel, since work order steps represented as table rows can be marked up with completion information as the work order proceeds. However, server-side scripting or an external notification agency is needed in order to avoid busy-waiting on SQL queries. System aggregation using this type of control channel becomes more a matter of launching applications and pointing them at the central database facility.

3.2 The Data Channel

The data channel's purpose is to distribute large data objects (order of megabytes) between nodes in an efficient and reliable manner.

• File-based, remote access; e.g. Network File System (NFS) or Storage Area Network (SAN): All data channel transfers are accomplished by writing the data to a file and using a remote access mechanism. This induces substantial overhead on cluster systems, but may be acceptable on large multiprocessors mated to a well-tuned SAN.
• File-based, remote copy: rsync or equivalent. This involves data transfers through the use of an agent service such as rcp, rsync or rdist. This is more efficient than remote access scenarios, but also engenders substantial overhead due to application launching and initialization, as well as local filesystem operations.
• Queuing system: Some queuing systems permit the specification of dependent data file transfers, as well as job dependencies. Using them has the benefit of leveraging pre-existing software, but as with the control channel, the mechanism of a queuing system would require a serialization facility. Interdependent job scripts could be generated and handled within the confines of the queuing system.
• MPI: Designed to move large medium-to-large data blocks between multiprocessor nodes efficiently, it is an attractive candidate technology for a data channel.
• TCP sockets: Another “roll your own” option which implies both aggregating the computing system and managing the connection and disconnection of sockets between any two nodes of the system.

3.3 The Audit Channel

The audit channel must be very reliable, ensure that transactions complete before proceeding, and provide a straightforward interface for reporting complete or aborted work orders.

• Formatted log file: This is the simplest audit recording system. On either a local or remote filesystem, formatted (e.g. XML) log files record the history of processing for a given UUID, allowing reconstruction of the computation and traceability back to standards. This approach becomes more expensive when the records are queried in order to make decisions on reprocessing; however, it does not preclude the possibility of moving audit files to searchable database tables at a later time.

• SQL database: A more advanced option is for the nodes to log completion of work orders and work order steps as transactions to a central SQL-based database engine.
• MPI: Audit data records could be formatted to fit an MPI transport, similarly to control channel traffic, with guarantees of arrival and synchronization. Archiving and reporting audit information would not be serviced by using an MPI audit channel.

3.4 The Monitoring Channel

The monitoring channel conveys real-time information on system status and performance – both science and engineering level – to the operator. This being the case, it should be concise, relatively low bandwidth, and emphasize responsiveness and flexibility.

• TCP or UDP sockets: Information can be provided to an external monitoring system by creating a simple key-value protocol using sockets or datagrams.
• SNMP: The Simple Network Monitoring Protocol was designed for tasks like this, but it presents the complication of mapping potentially science-centric values (e.g. data vectors) onto the SNMP “tree-structure” schema.

3.5 Technology Choice for this Prototype

In this iteration of the processing system, we chose MPI for system aggregation, control channel and data channel. MPI is available with facilities for system aggregation, fault tolerance, and efficient point-to-point, broadcast, and asynchronous messaging between nodes in a multiprocessor system. It is also relatively interconnect-agnostic: Implementations exist for IP over switched ethernet, NUMA systems with specialized interconnect hardware, and clusters using Myrinet and Infiniband interconnects.

We used locally formatted log files with offline reporting for the audit channel. In the interest of simplicity, log files written by individual nodes include work order information sufficient to backtrack the computation inputs and outputs. While not sufficient for a production system, this solution is adequate for testing purposes.

We deferred implementing a solution for the monitoring channel. As of this writing, our monitoring functionality did not have a GUI, and performance metrics were largely being logged as screen output. In the longer term, science and engineering performance monitoring will be accomplished by streaming tagged output to a display application.

3.6 Underlying Technologies

Choices of underlying platform and support technologies were made to assure flexibility, longevity of the design and overall performance.

• UML-RT[4] UML-RT provides a mechanism for representing real-time systems as recursive compositions of cooperating capsules, which communicate by exchanging asynchronous messages over predefined protocols. A subset of these concepts was used in the development of the science software being used in our test environment, with science capsules having limited and specific input and output ports to accept and return data. Designing the system first as UML-RT diagrams helps isolate control logic from science computations and input/output concerns, and facilitates separable testing and validation.

• C++: The C++ language inherits the performance characteristics of C, adding to it many powerful modern programming idioms. It provides high-level libraries, comprehensive compile time type checking, and can interact directly with FORTRAN, C, and systems services with minimal loss of efficiency. It is, however, difficult to master and subject to much variance in usage conventions.

• Boost: The Boost library set is a developmental wing of the C++ standard library, adding capabilities typically available in other high level languages. It includes libraries for regular expressions, process, thread and concurrency control, object serialization, and compile-time containers, algorithms and compile-time interface contracts. Similarly to C++ itself, the power of the Boost libraries does not come without a price of sometimes obscure syntax.

• MPI: The Message Passing Interface is a library that is used to build distributed applications, which can be implemented across shared memory, NUMA and distributed (e.g. cluster) systems. It provides an interface standard with many implementations for a variety of systems and interconnect technologies. MPI-2 includes further functionality such as run-time removal and aggregation of cluster nodes, handling of node failure, and one-sided data transfers. MPICH-2 implements this standard in an open-source and portable reference implementation.

4. TEST APPLICATION

The test application chosen to deploy in this architecture, SimCal4, is a simplified set of processing pipelines for GIFTS Level 0 (raw interferograms) to Level 1 (absolute earth radiance spectra) processing.

The input data for this pipeline consists of simulated instrument interferogram images derived from atmospheric model profiles[5]. Using simulated data provides a verifiable loop-back before instrument launch: the simulation dataset includes atmospheric profiles, absolute earth radiances as seen by the GIFTS instrument. The model atmospheric profiles can be checked against GIPS-generated profiles when Level 2 retrieval pipelines become available for integration into the system.

The components for performing Level 1 processing were readily available and tested, and the input data was available in quantity. The application consists principally of two science pipelines, SimCal4Study and SimCal4Apply.

SimCal4Study takes clumps of simulated blackbody interferograms and previous context data, and creates updated temporal context models for each detector pixel in the instrument. It is expected that self-calibration using blackbody views will be scheduled to occur approximately hourly for an orbiting GIFTS-like instrument. It can also be expected that the algorithm that fits calibration curves to data will evolve to include spacecraft temperature probes, diurnal temperature models, and some predictive capability in order to maximize the accuracy of the calibration of the instrument in real-time.

SimCal4Apply takes the calibration context generated periodically by SimCal4Study, along with earth observation interferogram images, and generates absolute earth radiance spectra for each pixel in the observation cube.

SimCal4Input is an input pipeline residing on one or more input service nodes. It accepts data cube files from an input repository, breaks them into 256 "clumps" of pixels, and forwards metadata in the form of a property list to the Master node. The clumped data is cached locally on the input server until it is used by a work order.

SimCal4Strategy accepts property list events, creating work orders and passing them to the control channel for broadcast. For this limited demonstration, the orders prescribe either SimCal4Study or SimCal4Apply for a given clump, based on the input data type (blackbody or earth) represented in the property list.

The work order also includes the URLs of the input data (as presented in the property list), as well as URLs of reference data on the detectors (such as off-axis angle), and the URL of the calibration context needed in order to process the data. In the case of SimCal4Study, this is the antecedent calibration context; in the case of SimCal4Apply, it is the relevant context, which is determined temporally as well as by pixel clump identity and interferometer scan direction.

The strategy also maintains "affinity" information on which pixel clumps are best processed on a given node. The calibration context needed to process a given earth radiance clump is several times the size of the data itself; therefore bandwidth waste and load on the reference service can be minimized by processing a given clump on a node that already has the relevant context cached locally.

5. IMPLEMENTATION

The relations between the basic datatypes of the system are represented in Figure 3 as a UML class diagram.

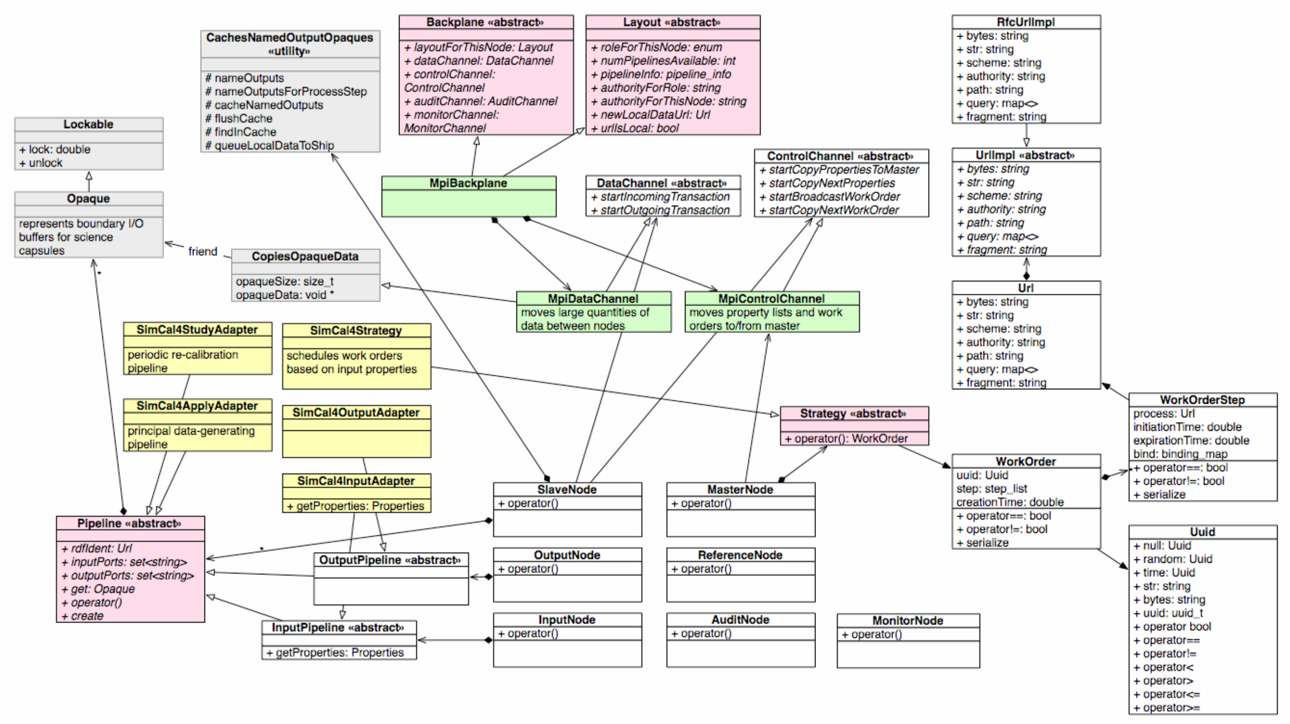

Figure 3: Class diagram of C++ GIPS prototype.

The implementation of this prototype system was started with a "bootstrap" element, the Backplane. This abstraction provides the nodes of the system with information on system layout, roles and configuration, and acts as a factory for the channels used by the node control logic.

Universal Unique IDentifiers (UUIDs) provide the naming functionality throughout the system, guaranteeing that each resource (data or code) within the system can be uniquely identified[6]. Uniform Resource Locators (URLs) are central to the control system, as they are used to localize and identify the code and data resources used through the system. Work orders are largely just compositions of URLs. UUIDs are used in composing URLs which identify not only the unique name of the data or process, but also the node on which to find it.

URLs are implemented as handles, with the URL class being a reference-counted pointer to an immutable UrlImpl, whose exact class can vary by URL scheme. This allows URLs to be declared and treated as simple and intuitive value variables. It also opens the possibility of shorthand representation in the future, as well as validation and manipulation of URLs which is specific to the class (i.e. scheme) of the URL. The URL implementation may grow to include functionality for external URLs to be access by way of libcurl or equivalent. For now, it is sufficient (albeit inefficient) that the two main UrlImpl classes are NullUrlImpl (an empty URL) and RfcUrlImpl (working from RFC 3986 URL spec). The RfcUrlImpl uses the boost::regex library to do on-the-fly breakdown of the URL into useful substrings. URLs also obey the boost::serializable conventions to allow them to be stored and transferred as part of a work order moving over the control channel.

In order to move data between nodes and pipeline boundary buffers, the concept of an opaque data object is created. Opaques are simply handles for blocks of memory, so called because the framework has no knowledge of the data contained in those blocks. An example of an opaque is a boundary buffer for the actual science processing capsules. The data channel can push and pull data to these boundary buffers once they're declared. opaque objects have a locking capability which permits write permission for the data they envelop to be mediated between the processing capsule and the data channel.

The relation between URLs, UUIDs and opaques is that for a given work order step on a slave node, each boundary buffer opaque represents a data object having a given UUID. Upon completion of that work order step, the UUID names are dissociated from the opaques, once a copy of any locally generated data is made to a most-recently-used node data cache.

Work orders are created by application strategy logic and distributed through the control channel. In the case of the current implementation, they are literally broadcast throughout the system. Minimally, however, they need to be broadcast to all the nodes named in the work order. A work order is composed of a UUID, and a listing of steps. Each step identifies (through URLs) the slave and the pipeline to run on it, followed by a set of Port – URL pairs assigning the input and output ports of the pipeline to their data names.

The control channel's principal responsibilities are broadcasting work orders from the master and permitting the master to receive property lists (effectively as events) from the rest of the system. The MPI implementation of the control channel uses its own MPI communicator, and handles the sending and receiving of property lists with asynchronous buffered transfers. Property lists and work orders are serialized to byte buffers prior to transfer. As dynamic message sizes are not easily handled by MPI, transfers of these buffers are handled by one nominally sized message which includes (when necessary) sizing and identification information on a follow-up message. It is intended that the nominally sized message have its size tuned to handle all normal operational traffic. While property lists are transferred using point-to-point messaging, currently work orders are simply broadcast to all nodes on the control channel communicator.

The data channel is the data transfer engine of the overall system, acting similar to a DMA (Direct Memory Access) controller in a computer architecture. Transfer requests are specified in terms of URL and opaque pairs, and a destination in the case of an outgoing transfer. The opaque data buffer is locked until the transfer completes. The MPI implementation of the data channel transfers three messages per send - a small "vanguard" block of uniform size which identifies the subsequent transfers and their sizes; a metadata block which contains the URL of the data and may include data properties in the future; and finally the data message itself. This permits the data channel to receive the bytes directly into the endpoint opaque, in the likely case that the receiving side has opened the receiving transaction in advance of the sending side initiating the send.

Pipelines are abstracted to allow the driving logic of the input, output and slave nodes to handle most of the details of servicing a work order. Their principal features are a URL identifying the module and providing a link to any metadata about its implementation and design; an invocation call which cycles the input data through the pipeline and results in new outputs; and access functions for obtaining named boundary opaque handles for the ports which feed (or are fed by) the pipeline. With this "simple-as-we-dare" interface, the slave control logic can walk through a work order, identify which pipeline to use from a URL-to-Pipeline table, form data channel transactions binding incoming and outgoing data to memory locations, wait for transfers to complete, cycle the pipeline capsule, and notify the data channel that the output is available to be moved to remote nodes for potential further processing.

A strategy is effectively an event handler, with the content of an event represented as a property list. The implementation of a strategy is application-specific, and is intended to separate the metadata-manipulating planning element from the data manipulation happening in the rest of the system. For a simple application, it may suffice for the strategy to stamp out a boilerplate work order to broadcast. For more complex algorithms, the strategy may write work orders including reference data updates, using cache affinities to minimize data transfers, and dynamically decide which pipelines to invoke based on data properties. In the case of the SimCal4Strategy, its primary decision point is whether to use the Study, or the Apply pipeline. It must also maintain state information on what contexts have been calculated, and make sure to route the correct context to the node performing the computation. The strategy's context cache is simply a set of URL lookup tables keyed on time, pixel information and view type. Furthermore, it must accept property lists informing it that a context calculation has been completed, and properly update its state so that later clumps use the most current context information. Thus, the strategy quickly becomes a complex set of algorithms. However, in being separate and metadata-centered, it is intended that many algorithms for families of instruments will evolve into reusable control patterns, analogous to the reusable science capsules.

6. OBSERVATIONS AND REFINEMENTS

During the course of implementing this prototype, many difficulties were encountered and circumvented, and observations made. The principal concern in developing this code quickly became MPI profiling and debugging.

Defensive programming techniques helped simplify debugging. Heavy use of assertions throughout development led to the system failing more often during development testing on assertion failures than on crashes or miscalculations. However, this complicated unit and integration testing as the scenarios painted by the unit tests were sometimes incomplete in line of the overall system assertions.

Multithreaded MPI presents difficulty in implementation and debugging. Debug statements often arrive out of order while running the code. Also, debugger, debug output, and performance profiling can impact the communications performance. More work on a variety of differently shaped test applications is needed to stress-test the framework fully. Further integration testing prior to running complete applications will be helpful.

Diagnosis of MPI bottlenecks is difficult, especially when asynchronous transfers are in progress. Monitoring of interconnect traffic using a tool such as Ethereal would be useful in diagnosing abrupt and lengthy delays. Correlating this diagnostic capability with load and activity of the individual nodes is needed to complete profiling of the system.

While MPI is excellent for scatter-gather parallel computations, it does not lend itself as well to support of distributed event-driven systems. This makes it more desirable to try strategies geared more toward large work orders in line with scatter-gather, rather than working with fine-grained work orders. A move to coarse-grained task division can also simplify strategy logic, but is more likely to waste compute cycles while waiting for transfers to complete.

Like many middleware solutions, committing to MPI presents an all-or-nothing conundrum; a program is straightforward only if it stays within the facilities offered by the middleware, however incomplete they be. An example complication in the case of GIPS was using the MPI_Waitany / MPI_Waitall select loop with POSIX mutex-based locks. Typically this is accomplished using threads, which complicates the implementation and makes it dependent upon fully thread-safe versions of MPI (e.g. MPICH2). A re-implementation which focuses all MPI operations on one event-loop thread may be required in order to simplify debugging; however, this reduces the role of MPI to aggregating the system and performing the duties of a socket library.

Better use of caching is needed, especially in the presence of large reference data objects. One of the benefits of the GIPS architecture is that it enables the master's strategy to prefer node-local data over data transfers. This requires that the nodes cache recently received and generated data, and balance the use available memory against the expense of copying the data from a remote reference server. If the UUID of a given data object matches cache content, the processing nodes can use the local copy instead of the original, which engenders either aborting or never initiating a transfer from another machine. Furthermore, the Master will be better able to schedule work orders if has inventory information on individual node caches. This becomes a tracking complication if cache discards are implicitly performed by node control logic, or a management complication if cache content is managed by the master.

Prototyping such a design variant on this architecture, however, assumes that the substantially larger overhead of such a design is tenable, with the benefit being reduced programmer time doing maintenance and development.

7. CONCLUSIONS AND FUTURE WORK

This prototype implementation has successfully distributed the SimCal4 science algorithms across an 80-processor Opteron cluster and demonstrated speed-up. However, profiling, bottleneck elimination, and caching improvements are currently underway in order to realize its full capability. It is very encouraging that in a relatively short amount of time it was possible to implement a critical subset of the functionality described in our previous design papers. Future refinements to this code will include modification of existing science capsules for processing real GIFTS data. Also, we will prototype distributed environmental data record (EDR) processing algorithms which await testing, validation and performance optimization. Eventually, experiments with software system metadata using Resource Description Framework (RDF) tools can be added to this framework. Our intention is to extend the existing GIPS to be the engine for a flexible, web-driven mass data processing portal**.**

REFERENCES

1. W. L. Smith, F. W. Harrison, D. E. Hinton, H. E. Revercomb, G. E. Bingham, R. Petersen, and J. C. Dodge, “GIFTS—The precursor geo-stationary satellite component of the future earth observing system,” in *Proc. IGARSS*, Toronto, ON, Canada, June 24–28, 2002, pp. 357–361.
2. Raymond K. Garcia, Maciej J. Smuga-Otto, "Component-oriented design studies for efficient processing of hyperspectral infrared imager data." in *Atmospheric and Environmental Remote Sensing Data Processing and Utilization: an End-to-End System Perspective* edited by Hung-Lung Allen Huang, Hal J. Bloom, Proceedings of SPIE Vol. 5548 (SPIE, Bellingham, WA, 2004) pp 444-454.
3. Ray Garcia, Steven A. Ackerman, Paolo Antonelli, Ralph G. Dedecker, Steven Dutcher, H. Ben Howell, Hung-Lung Huang, Maciej Smuga-Otto and David Tobin, “A Prototype for the GIFTS Information Processing System”, *21st International Conference on Interactive Information and Processing Systems for Meteorology, Oceanography, and Hydrology,* 2005.
4. Bruce Powel Douglass, *Real-Time UML: Developing Efficient Objects for Embedded Systems,* (2nd Edition), Addison-Wesley, New York, 1999.
5. Erik R. Olson, Jason Otkin, Robert Knuteson, Maciek Smuga-Otto, Raymond K. Garcia, Wayne Feltz, Hung-Lung Huang, and Christopher Velden, Leslie Moy, "Geostationary Interferometer 24-Hour Simulated Dataset for Test Processing and Calibration", *22nd International Conference on Interactive Information and Processing Systems for Meteorology, Oceanography, and Hydrology*, 2006.
6. Leach, P., M. Mealling, R. Salz, 2005: "A Universally Unique IDentifier (UUID) URN Namespace" *http://www.ietf.org/rfc/rfc4122.txt*